Fast verification algorithms in Matlab [1]

Siegfried M. Rump

1 Introduction

For the toolbox INTLAB, entirely written in Matlab, new concepts have been developed for very fast execution of interval operations to be used together with the operator concept in Matlab. The new implementation of interval arithmetic is strongly based on the use of BLAS routines. The operator concept of Matlab offers the possibility of easy and user-friendly access to interval operations, real and complex interval elementary functions, automatic differentiation, slopes, multiple-precision interval arithmetic and much more. Some of the new concepts are presented. The paper focusses on implementation and mainly on performance issues.

Hardware requirement for our approach is the IEEE 754 arithmetic standard (1985), which is implemented on many computers. If a special hardware supporting interval arithmetic or even elementary interval standard functions would be available, much of the below would be simpler and faster. However, our challenge was to design fast and easy to use algorithms running on standard computers, without additional hardware requirements. Furthermore, the algorithm should still be fast when written in Matlab.

Matlab (1997) is a widely used interactive programming environment for scientific computations. At a first glance it seems to be impossible to realize an operator concept in Matlab. This is because one of the main principles in Matlab is that *no* type declarations of variables are necessary but, by the interpretation principle, variables are automatically declared when used. Also, there is no distinction between scalars, vectors and matrices, whether they are real or complex; and a variable may frequently change its type.

The identification of new data types in Matlab works as follows. To define a new type, say TYPE, together with operators working on it, a subdirectory with name @TYPE is to be defined. This subdirectory has to be adjacent to the search path, i.e. the parent directory of @TYPE is in the search path of Matlab, whereas the subdirectory @TYPE itself is not in the search path.

Within the subdirectory @TYPE there has to be a routine named TYPE. This is the constructor for the new data type. The core of that routine, e.g. for TYPE being `intval`, could look as follows:

```
function A = intval(a)
  A.inf = a;
  A.sup = a;
  A = class(A,'intval')
```

The main statement is the last one, the "class-constructor", which tells the Matlab system that the output is a variable of "type" `intval`. From now

[1]published in G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto, editors, Symbolic Algebraic Methods and Verification Methods, pages 209-226. Springer Mathematics, 2001

on things are standard. Every operation involving a variable of the new data type calls a corresponding function in the subdirectory @TYPE, in our example `intval`, with fixed naming conventions. For example, names of operators are

```
+   plus
.*      times
*   mtimes
^   mpower
[ ... ]   horzcat
```

and many more. The operator concepts also includes a user-defined display routine. For example, the statement

```
A = intval(3.5);
5+A
```

calls the `intval` constructor in the first line: The variable `A` is now of type `intval`. The second statement calls the function `plus` for arguments `5` and `A`, and subsequently the `intval` display routine is called because the result of `5+A` is of type `intval`.

Summarizing this is a really nice and easy way to define and to use new operators in Matlab. For further information see Matlab (1997).

2 Performance issues

The nice working in Matlab is, there may be a severe interpretation time penalty when using low-level operators. Matlab is a *Mat*rix *Lab*oratory, and extensive use of scalar operators causes much interpretation overhead. Consider the following four ways of writing a matrix multiplication in Matlab, timing for multiplication of two randomly generated 200×200 matrices included.

```
n = 200; A = rand(n); B = rand(n);

C = zeros(n);
tic
  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
toc
```

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

C = zeros(n);
tic
  for i=1:n
    C(i,:) = C(i,:) + A(i,:)*B;
  end
toc

C = zeros(n);
tic
  C = A*B;
toc
```

The following table for a 300 MHz Pentium I Laptop shows the interpretation overhead in Mflop.

	3 loops	2 loops	1 loop	no loop
Mflop	0.05	1.9	36	44

Table 1. Interpretation overhead

The table clearly shows that minimization of interpretation overhead is mandatory if the system shall not be restricted to toy problems. In the following, consider interval matrix multiplication as our model problem.

Current implementations of interval matrix operations mostly use a top-down 3-loop approach, similar to the first one presented before. According to the above table this is much too slow in an interpretative system. However, even when using programming languages with highly optimized compilers such as Fortran or C, this top-down approach is very expensive in terms of computing time: The most inner loop is an interval operation, thus containing if-statements and case-distinctions. Such code can hardly be optimized by a compiler.

The effect of lack of optimization, of different sequencing of the loops and more subtle methods like unrolled loops shall be demonstrated in the following. For the moment, we restrict ourselve to pure floating point computations, no additional slow-down by interval computations present.

The first experiment is a scalar product $c = x^T y$ for $x, y \in \mathbb{R}^n$ for dimension $n = 1000$. The traditional loop is compared to an unrolled loop with five terms:

```
for ( i=0, c=0; i<n; i++)
  c += x[i]*y[i] + x[i+1]*y[i+1] + x[i+2]*y[i+2]
       + x[i+3]*y[i+3] + x[i+4]*y[i+4];
```

The performance rates in Mflops for the standard loop and the unrolled loop, both without and with compiler optimization on a RS 6000 workstation are as follows.

Performance [Mflop]	standard loop	unrolled loop
w/o optimization	4.3	6.9
with optimization	12.7	19.9

We see quite some increase in performance by the unrolled loops and, as we expect, by the optimization of the compiler. Both is *not* possible for the standard implementation of interval matrix multiplication; in other words, performance is basically limited to the smallest number in the above table.

Another standard method for improving performance is the sequence of loops in matrix multiplication. Any of the six possibilities ijk, ikj, ..., kji computes the correct matrix product, however, with quite different performance. The following table shows performance for the six possibilities, both without and with compiler optimization.

Performance [Mflop]	jki	kji	ikj	kij	ijk	jik
w/o optimization	2.1	2.1	2.9	2.9	2.9	2.9
with optimization	5.8	5.4	27	25	62	62
BLAS 3	100					

The big differences in performance are mainly due to memory access and cache optimization. The last line, with another improvement of a factor 1.5 compared to the best possible achieved so far, gives the Basic Linear Algebra Subroutines (BLAS) performance. The BLAS library (Dongarra et al. 1990) is available for almost every computer today. The ingenious idea of BLAS was that only the function headers are specified, whereas the implementation for each individual computer is performed by the manufacturer.

The numbers presented show how high-level routines and, whereever possible, using BLAS improves performance significantly.

3 Interval arithmetic

The product of two point matrices is easy to implement using BLAS. We use a routine `setround(m)` with the property that after the call `setround(-1)` the rounding mode is permanently switched downwards, i.e. every following operation is performed with rounding downwards according to the IEEE 754 standard (1985). This remains true until the next call of `setround`. Accordingly, `setround(m)` shall switch the rounding upwards for m=1, and to nearest for m=0.

Then the product of two point matrices $A \in M_{n,k}(\mathbb{F}), B \in M_{k,m}(\mathbb{F}), \mathbb{F}$ denoting the set of double precision floating point numbers, can be realized as follows.

```
setround(-1)
C.inf = A*B;
setround(1)
C.sup = A*B;
setround(0)
```

It follows

```
C.inf <= A*B <= C.sup
```

for comparison in a componentwise sense. Note that the proof is a successive use of the fact that, in case rounding is switched downwards, the sum and the product of two floating point numbers yields a floating point result definitely being less than or equal to the correct (real) result, whereas the floating point result is definitely greater than or equal to the exact result for rounding switched upwards.

Note that this approach does not necessarily work for other composed operations. For example, it is not correct for triple matrix products. Similarly, the product of a point matrix $A \in M_{n,k}(\mathbb{F})$ and an interval matrix $\mathbf{B} \in \mathbb{I}M_{k,m}(\mathbb{F})$ cannot be computed by `A*B.inf` and `A*B.sup`. In Rump (1999a) a method was proposed for fast computation of $A * \mathbf{B}$ using BLAS. The idea is the intermediate use of midpoint-radius representation.

```
setround(1)
Bmid = B.inf + 0.5*(B.sup-B.inf);
Brad = Bmid - B.inf;
setround(-1)
C1 = A * Bmid;
setround(1)
C2 = A * Bmid;
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 1. Point matrix times interval matrix

We mention that Algorithm 1 requires quite some additional memory. This could be reduced - at the expense of readability. If memory is a major issue, we would suggest to call a C or Fortran implementation, which may also avoid copying of matrices.

The first three lines calculate a matrix pair `<Bmid,Brad>` with the property

$$\forall B \in \mathbf{B} : \texttt{Bmid} - \texttt{Brad} \leq B \leq \texttt{Bmid} + \texttt{Brad}$$

for the comparision in the componentwise sense. This elegant way of transforming infimum-supremum representation into midpoint-radius representation in lines 1...3 is due to Oishi (1998). The next statements compute first an inclusion `[C1,C2]` of `A*Bmid`, and then take care of the radius `Brad`. The number of operations adds to $3n^3$ additions and $3n^3$ multiplications. This is 1.5 times more operations than necessary by the traditional implementation because the product of a floating point number and an interval can be performed with two multiplications (and a case distinction), and the interval addition requires two additions anyway. However, the following timings will demonstrate the vast improvement in performance of the new approach.

To our knowledge, there was only attempt to improve on the traditional implementation of interval matrix operations, namely the BIAS approach (Knüppel 1994). The following table gives the performance in "Miops" for multiplication of an $n \times n$ point times an $n \times n$ interval matrix for the traditional approach, for the BIAS approach and the above Algorithm 1. Timing is on a Convex SPP 200. Here the the matrix multiplication is counted as $2n^3$ interval operations, and a Miop are 1 Million interval operations.

Performance [Miops]	n=100	n=200	n=500	n=1000
traditional	6.4	6.4	3.5	3.5
BIAS	51	49	19	19
Algorithm 1	95	219	142	162

Table 2. Performance for point matrix times interval matrix

Obviously there is an immense improvement of performance by the new approach using BLAS routines. The decrease of performance of the BIAS library is due to cache misses. It could be improved by implementation of blocked algorithms, whereas the new Algorithm 1 uses BLAS, and therefore it uses blocked algorithms without effort on the part of the user. The varying performance of the new Algorithm 1 for different dimensions is also due to favourable and less favourable block sizes.

Another advantage of Algorithm 1 is that parallelization comes free of work, linking the parallel BLAS does the job. The BIAS approach can be parallelized as well, however, this has to be done by the user. The Convex SPP 200 allows us to use 4 processors, and performance data is as follows (for the traditional approach and the BIAS approach performance does not change unless special algorithms would be implemented).

Performance [Miops]	n=100	n=200	n=500	n=1000
Algorithm 1	142	551	397	526

Table 3. Parallel performance for point matrix times interval matrix using 4 processors

Comparing with the last row of Table 2 shows that the gain in performance is, for larger values of n, not too far from the magic factor 4. Note that this was achieved by merely linking the parallel BLAS library.

For the product of two interval matrices we also use an intermediate midpoint-radius representation. The performance numbers are even more impressing than before. However, there is a drawback to this, namely, that midpoint-radius product causes an overestimation of the true result whereas the infimum-supremum representation yields the sharp inclusion of the product of two interval matrices.

This would be a showstopper if overestimation could not be bounded. However, it can be shown that overestimation is globally bounded by a constant (Rump 1999a), and it is small if the input intervals are small. More precisely, define the relative precision prec($\mathbf{A}$) of an interval $\mathbf{A}$ by

$$\operatorname{prec}(\mathbf{A}) := \min\left(\frac{\operatorname{rad}(\mathbf{A})}{|\operatorname{mid}(\mathbf{A})|}, 1\right).$$

Let interval matrices $\mathbf{A}$ and $\mathbf{B}$ be given such that $\operatorname{prec}(\mathbf{A}_{ij}) \le e$ and $\operatorname{prec}(\mathbf{B}_{ij}) \le f$ for all i, j. Let $\mathbf{C}$ denote the result obtained by midpoint-radius arithmetic, and $\mathbf{A} {*} \mathbf{B}$ denote the narrowest interval matrix containing the power set product. Then the overestimation by midpoint-radius arithmetic satisfies

$$\frac{\operatorname{rad}(\mathbf{C})_{ij}}{\operatorname{rad}(\mathbf{A} * \mathbf{B})_{ij}} \le 1 + \frac{e \cdot f}{e + f} \le 1.5.$$

for all indices i, j. For example, input intervals with relative precision 1% suffer an overestimation of not more than 0.5% in radius.

If this overestimation is critical, the traditional interval matrix multiplication or some variant (Rump 1999a) may be used. Otherwise the following performance data apply. Again, matrix multiplication is counted as $2n^3$ interval operations.

Performance [Miops]	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
adapted Algorithm 1	91	94	76	99
adapted Algorithm 1 parallel	95	145	269	334

Table 4. Performance for interval matrix times interval matrix

Implementation of complex vector and matrix operations is not difficult. For various reasons we use circular arithmetic (midpoint-radius representation) in INTLAB. The implementation in Matlab is straightforward because real vector and matrix operations are already available. Thus the new approach also solves the problem of interpretation overhead.

Moreover, the above approach also applies to sparse matrices. As sparse matrices are already an intrinsic data type in Matlab, an implementation for sparse interval matrices comes without additional work.

A first simple application example is the check of nonsingularity of a given interval matrix $\mathbf{A} \in \mathbb{I}M_n(\mathbb{F})$. A well-known sufficient criterion for $\mathbf{A} \in \mathbb{I}M_n(\mathbb{F})$ being nonsingular is that for some matrix R and some interval vector $\mathbf{X}$

$$(I - R\mathbf{A}) \cdot \mathbf{X} \subseteq \operatorname{int}(\mathbf{X})$$

is satisfied. A simple implementation of this criterion is

```
R = inv(A.mid);
C = eye(n) - R*A;
X = infsup(-1,1)*ones(n,1);
Y = C*X;
res = all( ( X.inf<Y.inf ) & ( Y.sup<X.sup ) );
```

If `res`=1 after execution, every real matrix enclosed in the interval matrix **A** is proved to be nonsingular. The above is only an example for ease of use.

4 Nonlinear problems

For application of verification methods to nonlinear problems especially the inclusion of derivatives of functions over a range is needed as well as interval elementary functions. Gradients can be computed using an operator concept and automatic differentiation (Rall 1981). The implementation is simplified by the vector and matrix operations in Matlab.

When defining a function, a user friendly way would use the same source code for evaluation of the function at some real or complex floating point number, for the evaluation of the range of the function, for gradient information or for the gradient of the function over a certain range. This causes a specific problem. Consider the sample function

$$f(x) = \sin(\pi x).$$

A Matlab implementation is

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

There are no problems when inserting a (real or complex) floating point number `x`, a gradient value or a slope value. Problems occur when inserting an interval `X`. In this case the user may want to use an inclusion of the irrational number π in the definition of the function. Otherwise a call `y = f(intval(1))` may produce an interval not containing zero because the (intrinsic) floating point approximation `pi` of π is used instead. A redefinition

```
function y = f(x)
  Pi = midrad(pi,1e-16);
  y = sin(Pi*x);
```

would calculate a correct inclusion `Pi` for π and deliver a correct inclusion for zero when calling `f(intval(1))`. However, the simple call `f(1)` would yield an unexpected interval answer. Obviously, the type of the result shall depend on the type of the input argument `x`: For interval input the computation should be performed in interval arithmetic with correct interval data for π, for floating point input the Matlab internal approximation `pi` is sufficient and a floating point approximate result should be delivered.

The solution to the dilemma are two functions to adjust the type of a constant. Consider

```
function y = f(x)
  Pi = typeadj( midrad(pi,1e-16) , typeof(x) );
  y = sin(Pi*x);
```

In this implementation `typeof(x)` returns type information about the input parameter x. Especially, this is `intval` for interval input and `double` for floating point input. The statement `typeadj(a,type)` adjusts the type of the input `a` to the type `type`. Especially, in case `type` is `double`, the midpoint of `a` is returned. This allows to write one source code for various applications, from pure floating point computation to complex interval gradients and others.

The above implementation requires rigorous standard functions. This has been indeed a major task in previous approaches. Following we present a simple method to implement rigorous standard functions over real and complex intervals.

5 Standard functions

For single precision it is not difficult to test all values of a built-in standard function for their accuracy and to add a suitable error margin. For double precision this is not possible.

Usually the built-in standard functions are very accurate, and it is seldom that a value is not correctly rounded to least significant bit accuracy. In fact, we rarely found cases where the computed result is off by more than one bit - at least for arguments in a reasonable range. However, there is no proof for the accuracy of the results, and in order to achieve truly rigorous results a guess of accuracy is not sufficient.

The idea is to use a table approach together with some correction formulas. Take, for example, the exponential. We suppose that multiple precision functions $\underline{F}, \overline{F}$ are available such that for a given floating point number $x \in \mathbb{F}$ it is

$$\underline{F}(x) \le e^x \le \overline{F}(x)$$

with high accuracy. Define

$$R_{\exp} := \{\pm(0, 1, \ldots, 2^{14} - 1) \cdot 2^{-14}\} \subseteq \mathbb{F}$$

as a reference set for the exponential. For given $x \in \mathbb{F}$ define $y = \exp_\square(x) \in \mathbb{F}$ to be the floating approximation computed by the given (floating point) exponential function. Then the error of such approximations over the reference set is defined as follows.

$$\begin{array}{rcll} \underline{\varepsilon} & := & \max\limits_{x \in R_{\exp}} \{(y - \underline{F}(x))/|y| : & y = \exp_\square(x)\}, \\ \overline{\varepsilon} & := & \max\limits_{x \in R_{\exp}} \{(\overline{F}(x) - y)/|y| : & y = \exp_\square(x)\}. \end{array}$$

A short computation yields

$$y - \underline{\varepsilon} \cdot |y| \le e^x \le y + \overline{\varepsilon} \cdot |y|.$$

Now lower and upper bounds for the left hand side and right hand side, respectively, are computable for every $x \in R_{\exp}$ by

```
ys = exp(x);
setround(-1)
y.inf = ys + (-eps)*abs(ys);
setround(1)
y.sup = ys + eps*abs(ys);
```

with the property

$$\texttt{y.inf} \le e^x \le \texttt{y.sup} \quad \forall x \in R_{\text{exp}}$$

where `eps`$:= \varepsilon_{\text{exp}} = \max(\underline{\varepsilon}, \overline{\varepsilon})$. The advantage is that $\underline{F}(x), \overline{F}(x)$ for all $x \in R_{\text{exp}}$ and ε_{exp} have to be computed *only once*. From then on the constant ε_{exp} is a system constant to be used in the further computations. For general $X \in \mathbb{F}$ inclusions of the exponential are computed as follows. In an initialization procedure we compute floating point numbers $E_\nu, \underline{E}_\nu, \overline{E}_\nu$ with

$$E_\nu + \underline{E}_\nu \le e^\nu \le E_\nu + \overline{E}_\nu \quad \text{for } \nu \in \mathbb{Z}, -744 \le \nu \le 709.$$

For $x \le -745$ or $x \ge 710$, e^x is outside the double precision floating point range. Otherwise for $X \in \mathbb{F}$, split $X := X_{\text{int}} + x$ with $X_{\text{int}} = \text{sign}(X) \cdot \lfloor |X| \rfloor, -744 \le X_{\text{int}} \le 709$, and $-1 < x < 1$. Furthermore, set

$$\begin{aligned} \tilde{x} &= 2^{-14} \cdot \lfloor 2^{14} x \rfloor \\ d &= x - \tilde{x}. \end{aligned}$$

Then $\tilde{x}$ has no more than 14 leading bits in its binary representation and $\tilde{x} \in R_{\text{exp}}$. This implies

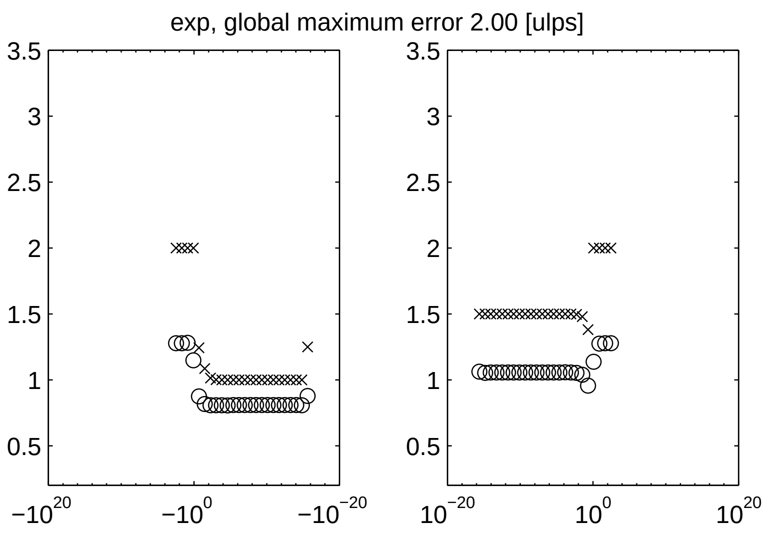
$$e^x = e^{\tilde{x}} \cdot e^d \quad \text{with} \quad \sum_{i=0}^{3} \frac{d^i}{i!} \le e^d \le \sum_{i=0}^{3} \frac{d^i}{i!} + e \cdot \frac{d^4}{4!}.$$

By the choice of the reference set it is $0 \le d < 2^{-14}$ and

$$e \cdot \frac{d^4}{4!} \le 0.68 d \cdot \frac{d^3}{3!} < 1.6 \cdot 10^{-18}.$$

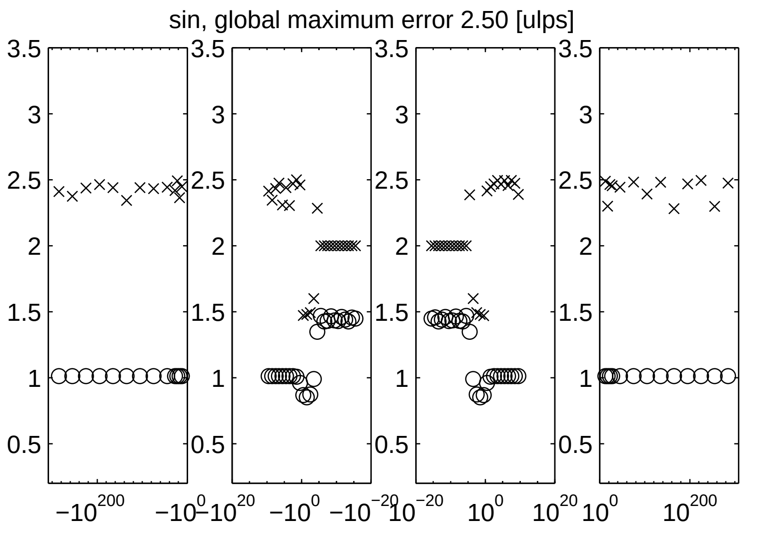
Putting things together yields rigorous and very sharp bounds for the value of the exponential over the entire floating point range.

Corresponding reference sets and formulas for splitting arguments have been developed for all elementary standard functions. A careful implementation of the formulas yields standard functions of very high accuracy, in fact always better than 3 ulp. For example, the relative error for the exponential, tested over some 50 million test cases, looks as follows. Here crosses depict the maximum relative error of the lower and upper bounds against each other over a certain domain, whereas the circles depict the average error.



Graph 1. Accuracy exponential

The maximum error is not more than 2 ulps, whereas the average relative error is about 1 ulp. Even for the trigonometric functions, where problems with argument reduction may cause significant cancellation errors, high accuracy is achieved. For example, the error plot for the sine is as follows.



Graph 2. Accuracy sine

The test set for the sine comprises of some 100 million floating point numbers in the range from -10^{300} to 10^{300}.

Guaranteed accuracy does not come free of cost, especially when the entire implementation is performed in Matlab itself, suffering from interpretation overhead. For the sine of a single floating point number the verified computation takes about 700 times the computing of an approximate computation. For vector input interpretation overhead decreases. For example, vectors of length 100 take about 200 times more computing for the verified sine.

The computing time for trigonometric functions suffers from argument reduction. In a C- or Fortran implementation the factor would be much smaller. Other functions behave better. In the following table we list timings for exp and atan on a 300 MHz Pentium I Laptop for input vector x with n components.

time	n=1	n=100
[msec]	approx./verified/ratio	approx./verified/ratio
exp	0.03 / 3.9 / 152	0.22 / 7.7 / 35
atan	0.02 / 3.5 / 164	0.10 / 6.4 / 58

The table shows that computing time for verified computation only doubles when going from one double number to a vector with 100 components. This is due to interpretation overhead.

For the other standard functions similar considerations apply. The formulas must be developed carefully in order to maintain the anticipated accuracy of 3 ulp. This has been performed for exp, log, log10, the trigonometric functions and their inverse functions, and for the hyperbolic functions and their inverses. All are included in the new version INTLAB.

For standard functions over rectangular intervals, efficient and accurate algorithms have been developed by Braune (1987) and Krämer (1987). For complex interval functions in our circular arithmetic we use the results by Börsken (1978). Denote for given $a \in \mathbb{C}, 0 \le r \in \mathbb{R}$

$$A =< a, r >:= \{z \in \mathbb{C} : |z - a| \le r\}.$$

Then Börsken defines the midpoint of $f(A)$ to be $f(a)$ and shows

$$\begin{array}{lcl} \exp(< a, r >) & \subseteq & < \exp(a), |\exp(a)| \cdot (\exp(r) - 1) > \\ \log(< a, r >) & \subseteq & < \log(a), -\log(1 - r/|a|) > \\ \mathrm{sqrt}(< a, r >) & \subseteq & < \mathrm{sqrt}(a), |\sqrt{|a| - r} - \sqrt{|a|}| > . \end{array}$$

These bounds are sharp for certain arguments, in other cases there may be quite some overestimation due to the choice of the midpoint. However, other formulas defining a different and more optimized midpoint with respect to, for example, the area of the inclusion may become quite involved. To our knowledge nothing is known in this direction, a research opportunity.

With those three standard function being available, all other mentioned elementary functions can be expressed by standard formulas. Doing this, another problem occurs. Complex standard functions are usually defined by some main value. This causes discontinuities. For example, $\sqrt{-4} = +2i$, but $\sqrt{-4 + \varepsilon i}$ for small $\varepsilon < 0$ yields a value near $-2i$. This causes problems when an interval approaches the negative real axis and $f(X)$, for a complex interval X, is defined to enclose the result of the usual power set operation

$$f(\mathbf{X}) := \{f(x) : x \in \mathbf{X}\}.$$

For the moment, we choose to use the above formulas. This implies that for a given function f the following weaker statement is true:

$$\mathbf{Y} = f(x) \Rightarrow \forall x \in \mathbf{X}\ \exists y \in \mathbf{Y} : f^{-1}(y) = x.$$

There is also space for future development and research. We note that for complex rectangles individual algorithms for each elementary standard function have been given in Braune (1987) and Krämer (1987), which cover also the above problem.

6 Long arithmetic

The above approach for the definition of standard functions requires some multiple precision arithmetic with error bounds. There are a number of such packages available, for example this of Aberth and Schaefer (1992), only to mention one. However, we choose to write a package in Matlab in order to maintain best possible portability.

The data type *long* has the structure

```
C.sign       in {-1,1}
C.mantissa   in 0 .. beta-1
C.exponent   representable integer ( -2^52+1 .. 2^52-1 )
C.error      nonnegative double, stored by C.error.mant and
             C.error.exp
```

representing the long number

$$\texttt{C.sign} \cdot \sum_{\nu=1}^{n} \texttt{C.mantissa}_{\nu} \cdot \beta^{\texttt{C.exponent}-\nu},$$

where `C.mantissa` is an array of length n corresponding to the precision in use. The field `C.error` is optional; if specified it represents the number

$$\texttt{C.error.mant} \cdot \beta^{\texttt{C.error.exp}},$$

where both `C.error.mant` and `C.error.exp` are nonnegative double numbers. It is interpreted as the radius of an interval with the above midpoint. Therefore the radius is stored in only two double numbers, whereas the midpoint is stored in an array of double numbers. The basis β is a system constant. It is a power of 2; usually 2^{25} is used.

The definition of multiple precision arithmetic is standard. For the current implementation we have two additional difficulties. First, it is no integer arithmetic but a long floating point arithmetic (to base β). This makes addition and subtraction more involved. Secondly, all long routines are written to support vector input in a vectorized computation. Treating vectors one component after the other causes a significant interpretation overhead. For example, given two long vectors X and Y of length 100 and precision of 500 decimal places, the calculation of the 100 products $X(i) * Y(i)$ by

```
for i=1:100, Z(i) = X(i)*Y(i); end
```

takes 25.9 sec, whereas the vectorized multiplication

```
Z = X.*Y;
```

takes only 0.6 sec on a 300 MHz Pentium I Laptop. Otherwise, the vectorized operations in Matlab can be used. For example, the multiplication of multiple precision numbers is a convolution and already built into Matlab.

7 An example of INTLAB code

Finally we give an example of INTLAB code to demonstrate its ease of use and readability. We print the full code to compute inclusions of multiple eigenvalues and corresponding invariant subspaces for a given (real or complex, not necessarily symmetric or Hermitian) matrix. We also give the full code how to call the algorithm. So the following is executable code in INTLAB under Matlab.

```
function [L,X] = VerifyEig(A,lambda,xs)
%VERIFYEIG  Verification of eigencluster near (lamda,xs)
%
%   [L,X] = VerifyEig(A,lambda,xs)
%
%Input: an eigenvalue cluster near lambda, where xs(:,i), i=1:k
%  is an approximation to the corresponding invariant subspace.
%
%On output, L contains (at least) k eigenvalues of A, and X
%  includes a base for the corresonding invariant subspace.
%By principle, L is a complex interval.
%

% written  07/15/99 S.M. Rump
%

  [n k] = size(xs);

  [dummy, index] = sort(sum(abs(xs),2));   % choose normalization
                                           %  part
  u = index(1:n-k);
  v = index(n-k+1:n);
  midA = mid(A);

  % one floating point iteration
  R = midA - lambda*speye(n);
  R(:,v) = -xs;
  y = R\(midA*xs-lambda*xs);
  xs(u,:) = xs(u,:) - y(u,:);
  lambda = lambda - sum(diag(y(v,:)))/k;
```

```
R = midA - lambda*speye(n);
R(:,v) = -xs;
R = inv( R );
C = A - intval(lambda)*speye(n);
Z = - R * ( C * xs );
C(:,v) = -xs;
C = speye(n) - R * C;
Y = Z;
Eps = 0.1*abs(Y)*hull(-1,1) + midrad(0,realmin);
m = 0;
mmax = 15 * ( sum(sum(abs(Z(v,:))>.1)) + 1 );
ready = 0;
while ( ~ready ) & ( m<mmax ) & ( ~any(isnan(Y(:))) )
  m = m+1;
  X = Y + Eps;                % epsilon inflation
  XX = X;
  XX(v,:) = 0;
  Y = Z + C*X + R*(XX*X(v,:));
  ready = all(all(in0(Y,X)));
end

if ready
  M = abs(Y(v,:));                 % eigenvalue correction
  [Evec,Eval] = eig(M);
  [rho,index] = max(abs(diag(Eval)));
  Perronx = abs(Evec(:,index));
  setround(1);
  rad = max( ( M*Perronx ) ./ Perronx );   % upper bound for
                                           % Perron root
  setround(0)
  L = tocmplx(midrad(lambda,rad));
  Y(v,:) = 0;
  X = xs + Y;
else
  disp('no inclusion achieved')
  X = NaN*ones(size(xs));
  L = NaN;
end
```

Algorithm 2. Rigorous inclusion of multiple eigenvalues

The algorithms follows Rump (2000) to be published in Linear Algebra and its Applications. It is based on the following.

For $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$ denote by $A \in M_n(\mathbb{K})$ an $n \times n$ matrix, by $\tilde{X} \in M_{n,k}(\mathbb{K})$ an approximation to an invariant subspace corresponding to a multiple or a cluster of eigenvalues near $\tilde{\lambda} \in \mathbb{K}$, such that $A\tilde{X} \approx \tilde{\lambda}\tilde{X}$.

The degree of arbitrariness is removed by freezing k rows of the approximation $\tilde{X}$. If the index set of the remaining rows is denoted by u, then we denote by $U \in M_{n,n-k}(\mathbb{R})$ the submatrix of the identity matrix with columns in u. Correspondingly, we set $v := \{1, \ldots, n\} \backslash u$ and define $V \in M_{n,k}(\mathbb{R})$ to comprise of the columns in v out of the identity matrix. That means $UU^T + VV^T = I$, and $V^T\tilde{X}$ is the normalizing part of $\tilde{X}$. Then the following is true (Rump 2000).

Theorem 1 *Let $A \in M_n(\mathbb{K}), \tilde{X} \in M_{n,k}(\mathbb{K}), \tilde{\lambda} \in \mathbb{K}, R \in M_n(\mathbb{K})$ and $\mathbf{X} \in \mathbb{I}M_{n,k}(\mathbb{K})$ be given, and let U, V partition the identity matrix as defined before. Define*

$$f(\mathbf{X}) := -R(A\tilde{X} - \tilde{\lambda}\tilde{X}) + \{I - R\big((A - \tilde{\lambda}I)UU^T - (\tilde{X} + UU^T \cdot \mathbf{X})V^T\big)\} \cdot \mathbf{X}.$$

Suppose

$$f(\mathbf{X}) \subseteq \mathit{int}(\mathbf{X}).$$

Then there exists $\hat{M} \in M_k(\mathbb{K})$ with $\hat{M} \in \tilde{\lambda}I_k + V^T\mathbf{X}$ such that the Jordan canonical form of $\hat{M}$ is identical to a $k \times k$ principal submatrix of the Jordan canonical form of A, and there exists $\hat{Y} \in M_{n,k}(\mathbb{K})$ with $\hat{Y} \in \tilde{X} + UU^T\mathbf{X}$ such that $\hat{Y}$ spans the corresponding invariant subspace of A.

Denote the k eigenvalues of $\hat{M}$ by $\mu_i, 1 \le i \le k$. Then the theorem implies that $\tilde{\lambda} + \mu_i$ are eigenvalues of A, and by Perron-Frobenius theory it is $|\mu_i| \le \rho(\hat{M}) \le \rho(|\hat{M}|)$ for $1 \le i \le k$. This proves that `L` is indeed an inclusion of (at least) k eigenvalues of A.

For successful termination of Algorithm 2, the matrix A must have a cluster of k eigenvalues near the input approximation `lambda`, which is well enough separated from the rest of the spectrum. The necessary degree of separation depends on the condition number of the cluster. In case of a multiple eigenvalue, the larger the maximum size of a corresponding Jordan block is, the larger the separation needs to be.

E.g., for a Jordan block of size m, the sensitivity of the eigenvalue to an ε-perturbation is $\varepsilon^{1/m}$, and practical experience shows that the algorithm terminates successfully if the separation is of the order $10\varepsilon^{1/m}$ (Rump 2000). Henceforth, the choice of the dimension k of the invariant subspace is important, as the separation and the sensitivity of the cluster depends on k.

Following we give two examples how to call the algorithm. The first one generates a random matrix, calculates approximations for the eigenvalues and eigenvectors and calls the algorithm for the first approximate eigenvalue/eigenvector pair. Note that `[V,D] = eig(A)` calculates a matrix `V` of eigenvectors and diagonal matrix `D` of eigenvalues, such that `A*X` is approximately equal to `X*D`. Furthermore, `rand` produces random numbers uniformly distributed in the interval [0,1] such that the entries of `A` are uniformly distributed within [-1,1].

```
n = 100; A = 2*rand(n)-1;
tic, [V,D] = eig(A); toc
tic, [L,X] = verifyeig(A,D(1,1),V(:,1)); toc
format long
L
```

This produces the following output:

```
elapsed_time =
   0.6100
elapsed_time =
   0.9900
intval L =
  -4.6875246581698_ +  3.4404126988075_i
```

The underscore in the output of the inclusion `L` of the eigenvalue indicates that the last digit of the real and the imaginary part is uncertain. More precisely, subtracting and adding one to the last displayed figure (before the underscore) yields a correct inclusion. Note that input/output is also rigorous by means of specific INTLAB routines for interval I/O (Rump 1999b).

The second example produces a triple eigenvalue 1 together with some randomly choosen 97 eigenvalues in $[-1, 1]$.

```
X = 2*rand(100)-1; A = X * diag([1 1 1 2*rand(1,97)-1]) * inv(X);
tic, [V,D] = eig(A); toc
index = find( abs(diag(D)-1)<1e-12 );
k = index(1);
tic, [L,X] = verifyeig(A,D(k,k),V(:,index)); toc
format long
L
```

The result is as follows.

```
elapsed_time =
   0.5000
elapsed_time =
   0.9900
intval L =
   1.000000000000__ -  0.000000000000__i
```

The inclusion fails if one of the 97 randomly chosen eigenvalues is, by chance, too close to 1 or, if X is too ill-conditioned such that `eig` delivers poor approximate eigenvectors and -values in V and D or, the size k of the cluster is incorrect such that the separation is too bad.

There are many more examples including ill-conditioned ones in the paper cited above (Rump 2000). Here our main objective is ease of use and readability. Note especially the index notation in Algorithm 2.

Conclusion

We presented some of the main ideas of the toolbox INTLAB for Matlab. INTLAB is available in its third release for PCs, a number of workstations and mainframes. More details can be found in Rump (1999b). The only machine dependency is the routine `setround` for switching the rounding mode. This assembly language routine is available for a number of machines. In Release 5.3

of Matlab under Windows even this is a built-in routine of Matlab. Then the entire toolbox is plain Matlab code.

All other code, some 362 functions and some 20 kLOC, is written in Matlab and therefore as portable as it can be. INTLAB is freely available for non-profit use from our homepage.

Acknowledgement

The author thanks the anonymous referee for his thorough reading and valuable comments.

References

Aberth, O., Schaefer, M. J. (1992): Precise computation using range arithmetic, via C++. ACM Trans. Math. Softw. 18(4): 481-491

Börsken, N. C. (1978): Komplexe Kreis-Standardfunktionen (Ph.D.), Freiburger Intervall-Ber. 78/2, Inst. f. Ange- wandte Mathematik, Universität Freiburg

Braune, K. D. (1987): Hochgenaue Standardfunktionen für reelle und komplexe Punkte und Intervalle in beliebigen Gleitpunktrastern (Ph.D.). Universität Karlsruhe

Dongarra, J. J., Du Croz, J. J., Duff, I. S., Hammarling, S. J. (1990): A set of level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Softw. 16: 1-17

ANSI/IEEE 754 (1985): Standard for Binary Floating-Point Arithmetic

Knüppel, O. (1994): PROFIL/BIAS - A fast interval library. Computing 53: 277-287

Krämer, W. (1987): Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerab- schätzungen für beliebige Datenformate (Ph.D.). Universität Karlsruhe

MATLAB User's Guide, Version 5 (1997): The Math Works Inc.

Oishi, S. (1998): private communication

Rall, L. B. (1981): Automatic Differentiation: Techniques and Applications. Lecture notes in Computer Science 120. Springer Verlag, Berlin-Heidelberg-New York

Rump, S. M. (1999a): Fast and parallel interval arithmetic. BIT 39(3):539-560

Rump, S. M. (1999b): INTLAB - INTerval LABoratory. In: Csendes, T. (ed.): Developements in Reliable Computing. Kluwer Academic Publishers, 77-104

Rump, S. M. (2000): Computational Error Bounds for Multiple or Nearly Multiple Eigenvalues. LAA, to appear