Fast (Parallel) Dense Linear System Solvers in C-XSC Using Error Free Transformations and BLAS

Walter Krämer[1] and Michael Zimmer[1]

University of Wuppertal, Gaussstr.20, 42097 Wuppertal, Germany

Abstract. Existing selfverifying solvers for dense linear (interval-)systems in C-XSC provide high accuracy, but are rather slow. A new set of solvers is presented, which are a lot faster than the existing solvers, without losing too much accuracy. This is achieved through two main changes. First, an alternative method for the computation of exact dot products based on the DotK-Algorithm is implemented. Then, optimized BLAS and LAPACK routines are used for the most costly parts, in terms of runtime, of the algorithm. Verified results are achieved by manipulating the rounding mode of the processor. Finally, an efficient parallel version of these solvers for distributed memory systems, based on ScaLAPACK, is presented, which allows to solve very large dense systems.
The new solver is compared to other solvers with respect to runtime and to numerical quality of the final result.

Keywords: Selfverifying methods, large linear interval systems, DotK methods, parallelization, block cyclic distribution, C-XSC, interval computations

AMS subject classification: 65H10, 15-04, 65G99, 65G10, 65-04, 68W15

0 Some Notation

The set of floating point numbers with t mantissa digits, base b and exponents between $e_{\min}$ and $e_{\max}$ is denoted by $\mathbb{F} = \mathbb{F}(b, t, e_{\min}, e_{\max})$; operations in this set are denoted by a box around the corresponding operator (e. g. $\boxplus$, $\boxminus$, $\boxdot$). Sometimes floating point operations are alternatively denoted by $\mathrm{fl}(\cdot)$, for example, $\mathrm{fl}(\sum_{i=1}^{n} x_i)$ means $x_1 \boxplus x_2 \ldots \boxplus x_n$. All other mathematical operations are meant to be exact. If we want to emphasize that a quantity is a (machine) interval quantity, we surround it by brackets, for example $[C]$. Computing interval enclosures of expressions is sometimes indicated in the form $\diamond(\ldots)$, e. g. $[C] = \diamond(I - RA)$. The function *mid* computes an approximation to the midpoint of an interval (componentwise for interval matrices and interval vectors). It is guaranteed that $mid([x]) \in [x]$. As condition number of a matrix A we use the definition $cond(A) = ||A||_\infty ||A^{-1}||_\infty$.

Dagstuhl Seminar Proceedings 08021
Numerical Validation in Current Hardware Architectures
http://drops.dagstuhl.de/opus/volltexte/2008/1436

1 Introduction

In this paper we discuss solvers that compute a verified solution x of the problem

$$Ax = b, \qquad (\text{or } [A]x = [b])$$

where A is a dense square matrix, while the right hand side b and the solution x are vectors of corresponding size. The elements of A and b can be real, complex, intervals or complex intervals. The solution x is an interval vector or a complex interval vector whose diameter should be as small as possible.

There are already a number of selfverifying solvers available in C-XSC to attack this problem (see Section 1.1), however, time measurements show that these solvers are quite slow. For example, solving a system with a random floating point matrix $A \in \mathbb{R}^{n\times n}$ with Matlab's floating point solver, the selfverifying solver from Intlab [25], and the selfverifying solver from the C-XSC toolbox [8] on a Pentium 4, 2.8 GHz with 1 GB RAM, one gets the timings shown in Table 1.

Table 1. Time in s for solving a real $n \times n$ system

n	**C-XSC toolbox**	**Intlab**	**Matlab**
100	0.952	0.028	0.002
500	112.72	0.57	0.07
1000	797.15	3.72	0.48

The non-verifying Matlab solver of course is the fastest, but the difference between Intlab and C-XSC is huge, especially when considering that both solvers use basically the same algorithm [26]. The main reason is that the C-XSC solver uses the long accumulator [4] (see Section 2.1) throughout the algorithm to compute every dot product exactly, which of course has, if done without hardware support, a huge impact on the performance.

On the other hand, using exact dot products (the long accumulator) enables the C-XSC solver to compute results of very high accuracy. For example, solving a system with the ill-conditioned Hilbert-Matrix of dimension $n = 10$ yields the results shown in Figure 1.

In this case, the C-XSC solver is able to compute the exact result [1]. Intlab computes an enclosure of the result, but loses about 7 digits, while the pure floating point Matlab solver loses about 12 digits (and of course doesn't compute a verified enclosure).

Therefore, the goal was to develop new solvers in C-XSC that are considerably faster while maintaining a high accuracy in the numerical results. This has been achieved through the use of an alternative algorithm for (approximating) dot products (Section 2) and the use of BLAS and LAPACK libraries (Section 3).

[1] It holds $(Ax - b) = 0$ which can be checked in C-XSC using exact dot products componentwise.

C-XSC toolbox			Intlab			Matlab		
+	1.000000000000000	E+2	+	1.000000000------	E+2	+	9.999989996685883	E+1
-	4.950000000000000	E+3	-	4.95000000-------	E+3	-	4.949990494288593	E+3
+	7.920000000000000	E+4	+	7.92000000-------	E+4	+	7.919978253948466	E+4
-	6.006000000000000	E+5	-	6.00600000-------	E+5	-	6.005979106504940	E+5
+	2.522520000000000	E+6	+	2.52252000-------	E+6	+	2.522509586181721	E+6
-	6.306300000000000	E+6	-	6.30630000-------	E+6	-	6.306270330270943	E+6
+	9.609600000000000	E+6	+	9.60960000-------	E+6	+	9.609549857202141	E+6
-	8.751600000000000	E+6	-	8.75160000-------	E+6	-	8.751550319100756	E+6
+	4.375800000000000	E+6	+	4.37580000-------	E+6	+	4.375773357049561	E+6
-	9.237800000000000	E+5	-	9.23780000-------	E+5	-	9.237740322228931	E+5

Fig. 1. Results of the different solvers for a real system with the Hilbert-Matrix, $n = 10$

Furthermore, to also allow large system matrices, an efficient parallelization of these new solvers is necessary. Such a parallelized solver for dense matrices is presented in Section 4.

1.1 Previous Work

There are a number of previous solvers with different purposes and features in C-XSC, some of which this work is based on:

- **LSS** [8]: A verified solver for real linear systems, which is part of the C-XSC Toolbox. This solver does not support for over- and undertermined systems, and does not use the second stage of the algorithm (see Section 3).
- **ILSS** [18, 10]: A verified solver for real interval linear systems, which is available as additional software for C-XSC. It supports over- and underdetermined systems and also supports the second stage of the algorithm.
- **PILSS** [7]: A parallel version of the solver `ILSS` for distributed memory systems using MPI.
- **BLSS**[11]: A solver, specifically designed for system matrices with banded structure.

1.2 Some Remarks on the C-XSC Library

C-XSC is a C++ class library for verified scientific computing, developed mainly at the University of Karlsruhe and the University of Wuppertal. It provides many data types, especially interval data types, with corresponding operators. The basic data type are `real`, `interval`, `complex` and `cinterval`. For each of these datatypes, corresponding vector and matrix types are available as well. The result of every single operation in C-XSC, using the corresponding C-XSC operator, is computed with maximum precision , i. e. with only one final rounding (using the long accumulator, if necessary), see Section 2.1. To allow the exact result of more complex dot product expressions, this long accumulator is also available as an own datatype, `dotprecision`. C-XSC also contains an extensive toolbox with useful software routines for many mathematical problems.

For further information, please see [13, 9], or visit the XSC-languages website at `http://www.math.uni-wuppertal.de/wrswt/xsc-sprachen.html`.

2 Exact/Accurate Dot Products

The ability to compute (enclosures of) dot products $x \cdot y = \sum_{i=1}^{n} x_i \cdot y_i$ in high(er) precision is very important, especially in verificational numerics. C-XSC relies on a powerful algorithm for this task, the long accumulator [20, 4], which is briefly explained in the next section. In Section 2.2 we present the implementation of an alternative algorithm in C-XSC, which allows to compute dot products in K-fold working precision, and which is at least for small K (in general) faster than the algorithm based on exact scalar product computations (long accumulator).

2.1 Exact Dot Product Computations Using the Long Accumulator

The basic idea of this algorithm is to use a fixed-point accumulator (long accumulator) of sufficient length, in which all the computations of the dot product are performed. This fixed-point accumulator is shown in Figure 2.

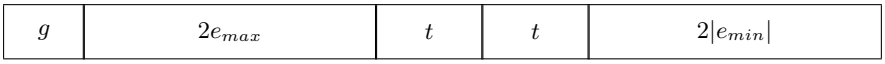

Fig. 2. Long accumulator

In this figure, t is the length of the matnissa, e_{min} and e_{max} are the smallest and largest exponent, respectively and g is a certain number of guard digits to prevent overflow. Inside the long accumulator, the value of a dot product of two floating point vectors can be computed and represented exactly. Only when converting the final result back to working precision, one final rounding has to be done.

It is important to note that, if realised in hardware, this method is even faster than ordinary floating point computation of the dot product by a loop [21]. However, this feature is not supported in current processors, and thus the long accumulator used in C-XSC has to be implemented in software leading to significant execution time penalties.

In C-XSC, the long accumulator is realised in four different classes. These are `dotprecision` for real results, `idotprecision` for interval results, `cdotprecision` for complex results and finally `cidotprecision` for complex interval results. It can be used in a relatively simple and self-explanatory way, as shown in the small example in Listing 1.1.

Listing 1.1. Example for the usage of the long accumulator in C-XSC

```
rvector x, y;                    //floating point vectors
```

```
//...
dotprecision accu(0.0);    //long accumulator
accumulate(accu, x, y);    //compute exact result of x*y
real result = rnd(accu);   //final rounding to nearest
```

The `accumulate` function computes the exact dot product inside the accumulator variable `accu`, while the function `rnd` rounds the current result stored inside the accumulator to the nearest floating point number.

2.2 The DotK Algorithm for Computing Dot Products in K-fold Working Precision

The DotK algorithm [23] is an alternative algorithm for dot product computations. Based on error-free transformations it computes dot products in a simulated K-fold working precision.

Let us first discuss the basic error-free transformations [15, 6, 5]:

Definition 1. *For all $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \cdot\}$ there exists an $y \in \mathbb{F}$ with*

$$a \circ b = x + y,$$

where $x = a \boxdot b$.

This means that every single operation of the dot product can be transformed into a sum $x + y$ of floating point numbers, where x is the floating point result of the operation under consideration and y is the corresponding error. Thus, all information of the correct result is known.

These error-free transformations can be computed in pure floating point. For the transformation of a sum of two floating point numbers, the algorithm `TwoSum` [15, 23] is used.

Data: Two floating point numbers $a, b \in \mathbb{F}$
Result: Two floating point numbers $x, y \in \mathbb{F}$ such that $x = a \boxplus b$ and $a + b = x + y$ hold
$x = a \boxplus b$
$z = x \boxminus a$
$y = (a \boxminus (x \boxminus z)) \boxplus (b \boxminus z)$

Algorithm 1: TwoSum

For the error-free transformation of a product of two floating point numbers, first the algorithm `Split` [6, 23] is needed. Here, $factor$ is a constant with $factor := 2^s + 1$ and $s := \lceil t/2 \rceil$, where t is defined by $eps = 2^{-t}$. So for `double` precision we get $t = 53$ and $s = 27$. This algorithm computes for a given floating point number a two floating point numbers x and y with $a = x + y$, $|y| \leq |x|$ and non overlapping mantissas of x and y.

Data: One floating point number $a \in \mathbb{F}$
Result: Two floating point numbers $x, y \in \mathbb{F}$ with $a = x + y$
$c = factor \boxdot a$
$x = c \boxminus (c \boxminus a)$
$y = a \boxminus x$

Algorithm 2: Split

Data: Two floating point numbers $a, b \in \mathbb{F}$
Result: Two floating point numbers $x, y \in \mathbb{F}$ with $a \cdot b = x + y$
$x = a \boxdot b$
$[a_1, a_2]$ =Split(a)
$[b_1, b_2]$ =Split(b)
$y = a_2 \boxdot b_2 \boxminus (((x \boxminus a_1 \boxdot b_1) \boxminus a_2 \boxdot b_1) \boxminus a_1 \boxdot b_2)$

Algorithm 3: TwoProduct

With this an algorithm `TwoProduct` [6, 23] for the error-free transformation of a product can be formulated (Algorithm 3).

With help of these algorithms it is possible to compute the dot product of two floating point vectors in two-fold working precision using the following algorithm:

Data: Two floating point vectors $x, y \in \mathbb{F}^n$
Result: The result res of the dot poduct $x \cdot y$ as if computed in two-fold working precision
$[p, s]$ =TwoProduct(x_1, y_1)
for *i=2:n* **do**
  $[h, r]$ =TwoProduct(x_i, y_i)
  $[p, q]$ =TwoSum(p, h)
  $s = s \boxplus (q \boxplus r)$
$res = p \boxplus s$

Algorithm 4: Dot2

This algorithm transforms a dot product of two floating point vectors of length n into a sum of floating point values of length $2n$. Computing this sum in pure floating point leads to an approximation of the dot product as if computed in two-fold working precision. The proof of this statement can be found in [23]. Now, to achieve a result in K-fold working precision, the sum of length $2n$, into which the dot product is transformed in Algorithm 4, must be computed in a simulated higher precision. This finally leads to the desired algorithm for dot product computations in $K-$fold precision (Algorithm 5).

It can be shown (see [23]) that this algorithm computes the result of a dot product of two given floating point vectors, as if computed in K-fold working precision. As proofed in [23], it is also possible to compute a reliable error bound

Data: Two vectors $x, y \in \mathbb{F}^n$, desired precision K
Result: The result res of the dot poduct $x \cdot y$, as if computed in K-fold working precision
$[p, r_1] =$TwoProduct(x_1, y_1)
for *i=2:n* **do**
  $[h, r_i] =$TwoProduct(x_i, y_i)
  $[p, r_{n+i-1}] =$TwoSum(p, h)
$r_{2n} = p$
for *K=1:K-2* **do**
  for *i=2:2n* **do**
    $[r_i, r_{i-1}] =$TwoSum(r_i, r_{i-1})
$res =$fl$(\sum_{i=1}^{n-1} p_i + p_n)$

Algorithm 5: DotK

for the achieved result, again only using pure floating point operations, resulting in the following algorithm `Dot2err`:

Data: Two vectors $x, y \in \mathbb{F}^n$
Result: The dot product $x \cdot y$, as if computed in two-fold working precision, as well as a strict error bound err for the result
$[p, s] =$TwoProduct(x_1, y_1)
$e = abs(s)$
for *i=2:n* **do**
  $[h, r] =$TwoProduct(x_i, y_i)
  $[p, q] =$TwoSum(p, h)
  $t = q \boxplus r$
  $s = s \boxplus t$
  $e = e \boxplus abs(t)$
$res = p \boxplus s$
$\delta = (n \boxdot eps) \boxslash (1 \boxminus 2n \boxdot eps)$
$\alpha = eps \boxdot abs(res) \boxplus (\delta e \boxplus 3eta \boxslash eps)$
$err = \alpha \boxslash (1 \boxminus 2eps)$

Algorithm 6: Dot2err

Thus, the only additional information needed to compute a valid error bound is the length n of the dot product as well as the floating point sum of the absolute values of the computed error terms in the final summation.

There also exists a parallel version of this algorithm for shared memory architectures [24].

Remarks on the Implementation. The main goal of the implementation of the DotK algorithm in C-XSC was to provide a handling similar to the

`dotprecision`-classes. Thus, four different DotK-classes have been implemented corresponding to the different four basic C-XSC datatypes:

- Class `RDotK` for dot products with real results
- Class `IDotK` for dot products with interval results
- Class `CDotK` for dot products with complex results
- Class `CIDotK` for dot products with complex interval results

Each of these new classes provides a set of useful constructors, operators and some basic set and get methods. More importantly, each class offers three essential methods: First a method `addDot`, with two floating point vectors as parameters, which computes the dot product of these vectors in the desired precision (denoted by a data member `K` of the class, which is changeable at runtime). Second, the two methods `res` and `res_enclosure`, which return either a result ignoring the error bounds or a reliable enclosure of the correct result using the error bounds.

To store the current intermediate result in this implementation of the DotK-algorithm a `dotprecision` variable of corresponding type is used. This has serveral advantages: It is possible to compute dot products seamlessly with the long accumulator using the DotK classes without having to change the source code. In the current implementation simply selecting precision $K = 0$ will let the DotK class compute dot products inside the `addDot` method using the long accumulator. Furthermore, this allows to add and substract single values from the intermediate result without introducing additional numerical errors. This special feature increases the accuracy of the new solvers presented below significantly.

For the interval, complex, and complex interval cases, the DotK algorithm has to be adapted appropriately. These changes are fairly easy, since these dot products can be reduced to (several) real dot products:

- A complex dot product of length n can be computed as two real dot products of length $2n$
- An interval dot product of length n can be computed as two real dot products of length n
- A complex interval dot product of length n can be computed as four real dot products of length $2n$

Table 2 presents time measurements for our implementation[2]. These tests have been performed on a Pentium 4 with 3.2GHz and 2GB RAM. The speed difference between the long accumulator and the DotK classes can be vastly different, depending on the used processor. The speed of the accumulator largely depends on the integer unit of the CPU, while the speed of the DotK algorithm depends on the floating point performance of the processor. On another Pentium 4 processor the DotK classes were faster than the accumulator even for a precision of $K = 10$, while on the Core 2 Duo architecture the long accumulator can

[2] When using the C-XSC interval operators for interval-types, the rounding mode is switched very often, which is very time-consuming. For complex types, the long accumulator is used internally for multiplication.

be even *faster* than the DotK classes for $K = 2$. The numerical results are not presented here but have been checked successfully to be of the expected K-fold precision.

n	**Computed with...**	**real**	**interval**	**complex**	**cinterval**
1000	C-XSC operators	0.01	0.58	1.20	3.84
	Accumulator	0.19	0.40	0.72	1.64
	DotK, K=2	0.03	0.12	0.20	0.47
	DotK, K=3	0.09	0.21	0.37	0.87
	DotK, K=4	0.12	0.27	0.47	1.07
	DotK, K=5	0.14	0.32	0.57	1.28
10000	C-XSC operators	0.06	5.83	12.07	40.01
	Accumulator	1.85	4.02	7.05	16.22
	DotK, K=2	0.35	1.16	2.03	4.64
	DotK, K=3	0.98	2.27	4.02	9.56
	DotK, K=4	1.24	2.79	5.04	11.62
	DotK, K=5	1.49	3.30	6.07	13.70
100000	C-XSC operators	0.64	58.37	120.16	373.2
	Accumulator	18.65	40.32	70.73	161.82
	DotK, K=2	3.53	11.66	20.41	46.46
	DotK, K=3	9.86	24.57	41.48	97.83
	DotK, K=4	12.44	29.76	51.91	118.75
	DotK, K=5	15.02	34.95	62.34	139.64

Table 2. Timings for dot products, $cond = 10^{30}$, repeated 1000 times

3 The New Serial Solvers

With the help of the DotK classes a set of new solvers for linear (interval) systems has been implemented. These new solvers as well as existing solvers in C-XSC are based on the same well known algorithm described by Rump [26], which itself is based on the Krawczyk-Operator [22]. Algorithm 7 works (in slightly modified forms) for real point and interval systems, as well as for complex point and complex interval systems. The last part of the algorithm, which uses an approximate inverse of double length $(R_1 + R_2)$, is also known as Rump's device and will be called part two or second stage of the algorithm in this paper.

The new solvers were designed to have the following basic features:

Data: A square matrix A und a right hand side b
Result: An interval enclosure of the solution of $Ax = b$
Compute approximate inverse R of A
Compute approximate solution $\widetilde{x} := Rb$
`// Defect iteration`
repeat
  $\widetilde{x} := \widetilde{x} + R(b - A\widetilde{x})$
until $\widetilde{x}$ *accurate enough or max iterations reached*
`// Compute enclosures of the residuum and the iteration matrix`
$Z := R \diamond (b - A\widetilde{x})$
$C := \diamond(I - RA)$
`// Verification`
$Y := Z$
repeat
  $Y_A := blow(Y, \varepsilon)$ //ε-inflation
  $Y := Z + C \cdot Y_A$
until $Y \subset interior(Y_A)$ *or max iterations reached*
`// Check results`
if $Y \subset interior(Y_A)$ **then**
  Unique solution in $x \in \widetilde{x} + Y$
else
  if *Approximate inverse of double length not yet used* **then**
    //Second stage
    $R_1 := R$
    $S := R_1 \cdot A$
    Compute approximate inverse S_1 of S
    $S := S_1 \cdot R_1$
    $R_2 := S_1 \cdot R_1 - S$
    $R_1 := S$
    Restart algorithm with new approximate inverse $R = R_1 + R_2$
  else
    Algorithm failed, A is singular or very ill-conditioned

Algorithm 7: Basic algorithm for selfverifying linear system solver

- Support for all basic C-XSC datatypes (`real`, `interval`, `complex`, `cinterval`)
- Support for over- and underdetermined systems
- Part two of the algorithm, using the inverse of double length, is available

Furthermore, in a first basic step, the required approximate inverse is computed using the well known Gauss-Jordan algorithm [27]. With these basic solvers one gets the results shown in Table 3. Intlab will be used as a reference point in the following measurements. The timings for Intlab contain some overhead due to interpretation and Matlab itself. However, this is not a serious drawback, because Matlab and thus Intlab use BLAS-routines for most computations, which are realized by native code.

What?	Solver	real	interval	complex	cinterval
Time	Intlab	3.86	5.16	16.00	17.14
	C-XSC	435.15	901.84	5971.27	6335.61
Exact Digits	Intlab	6.09	0.93	(6.67, 5.90)	(0.81, 0.05)
	C-XSC	15.79	1.93	(15.82, 15.78)	(1.71, 1.56)

Table 3. Timings for basic solvers

As a first step to improve the performance of these basic solvers, the DotK algorithm can be used for all dot products. This is a quite simple modification, as can be seen in the following example:

Listing 1.2. Computation of `[C]` $= \diamond$ `(I-R*A)` using the long accumulator

```
dotprecision Accu;

for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++) {
     Accu = (i == j) ? (real)1.0 : (real)0.0;
     accumulate(Accu,-R[i],A[Col(j)]);
     rnd(Accu,C[i][j]);
  }
}
```

Listing 1.3. Computation of `[C] = (I-R*A)` using the new DotK classes

```
RDotK dot(K);

for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++) {
     dot = (i == j) ? (real)1.0 : (real)0.0;
     dot.addDot(-R[i],A[Col(j)]);
     C[i][j] = dot.res_enclosure();
   }
}
```

Evidently, the required modifications are straight forward and rather small. Since the DotK-classes incorporate the accumulator, the programm can be "switched back" to ordinary C-XSC computations using the long accumulator without going back to the original source code, simply by setting the precision K equal to 0. Then, the DotK classes will simply use the `accumulate` function of the long accumulator to compute the dot product exactly, thus leading to the same results as in the original C-XSC version.

The modifications described above result in a significant performance enhancement as shown in Table 4.

What?	Solver	real	interval	complex	cinterval
Time	Intlab	3.86	5.16	16.00	17.14
	C-XSC, K=2	80.82	255.02	2914.91	3538.78
	C-XSC, K=3	91.09	324.55	2996.16	3923.15
Exact digits	Intlab	6.09	0.93	(6.67, 5.90)	(0.81, 0.05)
	C-XSC, K=2	13.99	1.93	(7.93, 7.93)	(1.70, 1.55)
	C-XSC, K=3	15.79	1.93	(15.82, 15.78)	(1.71, 1.56)

Table 4. Results after introducing DotK classes

As can be seen, the precision of the results is only slightly worse than before. The solvers are now between three and four times faster using the DotK classes with $K = 2$ than with the long accumulator.

However, a lot of additional potential for optimization still lies in the matrix-matrix product when computing the interval matrix $[C]$ and in the computation of the approximate inverse R. To speed up these parts of the algorithm, optimized routines from the BLAS and LAPACK [2] are used. For the computation of the approximate inverse the corresponding LAPACK-routines `xgetrf` and `xgetri` can be used in a straightforward way. For the computation of $[C]$ however, an enclosure of the product RA or $R[A]$, respectively, is needed. To achieve this with the ordinary BLAS-routine `dgemm`, manipulation of the rounding mode is used, in a similar way as in Intlab [25].

For this, a function `setround` is introduced, which sets the rounding mode of the processor to down (`setround(-1)`), up (`setround(1)`) or to nearest (`setround(0)`). With the help of this function, the enclosure of a real matrix-matrix product can be computed using Algorithm 8 (C-XSC like pseudocode).

Data: Two real matrices A and B
Result: An interval enclosure of the matrix $C = AB$
setround(-1);
SetInf(C, A*B);
setround(1);
SetSup(C, A*B);
setround(0);

Algorithm 8: Computation of real matrix product using BLAS

The two matrix-matrix-products in this algorithm are computed using the corresponding BLAS routine `dgemm`. The algorithms for the interval and complex case follow the same basic idea and are not shown here. These are basically the algorithms presented in [25] with some minor adaptations.

Using the BLAS routines in this way, as well as the LAPACK routine for the computation of the approximate inverse and the DotK-algorithm and utilizing

some other minor optimizations (see [28]) that are not detailed here, one finally gets the results shown in Table 3.

What?	Solver	real	interval	complex	cinterval
Time	Intlab	3.86	5.16	16.00	17.14
	C-XSC, K=2	3.96	5.34	15.82	18.88
	C-XSC, K=3	4.38	5.65	16.80	19.02
Exact digits	Intlab	6.09	0.93	(6.67, 5.90)	(0.81, 0.05)
	C-XSC, K=2	14.22	1.93	(13.63, 12.87)	(1.86, 1.11)
	C-XSC, K=3	15.79	1.93	(15.82, 15.78)	(1.86, 1.11)

Table 5. Final results of the new solvers

The new serial solvers are now a lot faster than the original C-XSC toolbox solver. For precision $K = 2$ they work roughly on the same speed as the Intlab solver `verifylss`, but they still maintain high accuracy, which in many practical cases are even the same as for the original C-XSC solvers. Furthermore, by increasing the precision of the DotK classes (increasing the value of `K`), the results shown in Table 3 often can be improved even more.

3.1 Some Numerical Results Concerning Ill-conditioned Matrices

In this paragraph we use the so called Boothroyd/Decker matrices A_n of dimension n to compare the numerical quality of our new solver with the numerical quality of the Intlab solver `verifylss`. The ill-conditioned matrices A_n are defined as follows

$$A_n = (a_{ij}) \text{ with } a_{ij} := \binom{n+i-1}{i-1}\binom{n-1}{n-j}\frac{n}{i+j+1}, i, j = 1, \ldots, n$$

For $n \leq 20$ all matrix elements are exactly representable as IEEE double numbers. As right hand side we use $b = (1, 1, \ldots, 1)^T$. Then, the exact solution vector is $x = (1, -1, 1, -1, \ldots)^T$.

Our new solver (using $K = 3$ for the DotK objects) returns for matrices up to order $n = 20$ either the exact solution vector or a tight enclosure of it whereas the solver `verifylss` breaks down for $n = 13$. The computed enclosure of the result vector in case of matrix order $n = 12$ is:

```
[    0.99999999982079,    1.00000000016779]
[   -1.00000000149304,   -0.99999999836399]
[    0.99999998842304,    1.00000001074763]
[   -1.00000004335473,   -0.99999995272265]
[    0.99999981774945,    1.00000016863901]
[   -1.00000048155897,   -0.99999947345126]
[    0.99999843469062,    1.00000144278687]
[   -1.00000335758376,   -0.99999630705000]
[    0.99999107401347,    1.00000823290467]
[   -1.00001753552019,   -0.99998078814825]
[    0.99994100039713,    1.00005551901631]
[   -1.00008847223671,   -0.99990482653736]
```

This Intlab result is only accurate to about 4 decimals. Our new solvers compute the exact result in this case. If $n > 12$, Intlab produces NaNs for all the components of the solution vector.

The comparison shows clearly that the new C-XSC solver outperforms the solver coming with Intlab. We should mention that the original C-XSC toolbox solver produces the same numerical results as our new solver.

4 Parallelization

In the following we explain an efficient parallelization of these new solvers. The basic ideas of the serial solver, the usage of the DotK classes and of BLAS/LAPACK routines, are carried over to this new parallel solver. Instead of LAPACK, the parallel version ScaLAPACK [3] is used.

For a more efficient approach than in the exisiting parallel solver `PLSS`, especially concerning memory, the matrices used in the algorithm are distributed equally among all processes. Every process saves certain parts of the matrices A, R and $[C]$, which it stores throughout the algorithm, while the occurring vectors are stored completely in every process. All processes are involved in every step of the algorithm, so there is no root node (no master/slave or worker/manager approach) or a similar approach.

The matrices are distributed using a two dimensional block cyclic distribution scheme, the same distribution model used by ScaLAPACK. For this distribution model, the available processes are arranged in a two dimensional grid. The matrix is then distributed according to this grid and a predefined block size (see Figures 3 and 4). The optimal values for the number of rows and columns in the process grid, as well as the block size used, are hardware dependent and are not trivial to compute, because a balance between the costly communications and an equal work load has to be found. In our solvers, the number of rows and columns of the process grid is chosen to be equal or as close as possible (the number of processes should idealy be $P = p^2, p \in \mathbb{N}$), while the block size can be determined by the user when starting the solver (a value of about 256 delivered the best results in our tests).

P_0	P_1	P_2
P_3	P_4	P_5
P_6	P_7	P_8
P_9	P_{10}	P_{11}

Fig. 3. Process grid for the two dimensional block cyclic distribution, 12 processes

P_0	P_1	P_0	P_1
P_2	P_3	P_2	P_3
P_0	P_1	P_0	P_1
P_2	P_3	P_2	P_3

Fig. 4. Distribution of a matrix in two dimensional block cyclic distribution using a 2×2 process grid based on 4 processes P_i

The inversion and the computation of the matrix-matrix product can now easily by computed using the corresponding ScaLAPACK routines in a similar way as in the serial case. However, the matrix vector products occurring in the algorithm also have to be parallelized. For this, so called MPI communicators are introduced for every row and every column in the process grid. With these communicators it is possible to perform a broadcast which is restricted to a specified row or column in the grid.

Every process then computes the parts of the occurring dot products for which it stores the needed matrix entries. All processes in one row in the process grid then have computed different parts of the same dot product. Thus, to compute a final result of this dot product, the data has to be broadcasted inside their row in the process grid (using the MPI communicators). With the help of the long accumulator, the final result can then be computed in every single process.

Now, all processes in one row know the parts of the final result corresponding to their matrix rows. Since every process needs to know the complete result vector, these parts have to be exchanged with the processes corresponding to the other rows of the result. Thus, another broadcast, this time in the columns of the process grid, has to be performed to exchange these parts. After this step, all processes know the complete final result vector.

Thus, the parallelization of these matrix-vector products is quite complicate and communication intensive. But since it is only used in the $O(n^2)$ parts of the algorithm and the used data distribution reduces the memory needed in each

process significantly, it is well worth the effort, as may be seen in the final results (Tables 6 and 7).

Computed with...	P	**real**	**interval**	**complex**	**cinterval**
DotK, K=2	1	265.30	363.88	1357.63	1723.44
	2	159.77	206.41	625.37	737.33
	4	99.98	128.77	365.58	411.14
	8	61.62	78.18	211.46	239.83
DotK, K=3	1	279.64	383.29	1387.32	1801.44
	2	172.94	217.06	665.44	771.45
	4	104.13	134.21	379.01	427.99
	8	63.92	81.02	218.35	246.95

Table 6. Time measurements parallel solvers, $cond = 10^{10}$, $n = 5000$

Computed with...	P	**real**	**interval**	**complex**	**cinterval**
DotK, K=2	1	1.0	1.0	1.0	1.0
	2	1.66	1.76	2.17	2.34
	4	2.65	2.83	3.71	4.19
	8	4.31	4.65	6.42	7.18
DotK, K=3	1	1.0	1.0	1.0	1.0
	2	1.62	1.77	2.08	2.34
	4	2.69	2.86	3.66	4.21
	8	4.37	4.73	6.35	7.29

Table 7. Speedup parallel solvers, $cond = 10^{10}$, $n = 5000$

Part two of the alogrithm (approximate inverse of double length), if used for very ill-conditioned systems (which is indeed its main purpose), will in general not compute a verified solution when BLAS routines are used, since the results of the matrix-matrix-products will then have too wide diameters. Therefore, these products have to be computed using a higher precision dot product (using the DotK classes). This means that these products have to be manually parallelized, while maintaining the distribution of the matrix entries as described above. To achieve this, a special case of the two dimensional block cyclic distribution scheme is used, where the process grid has P rows and one column, and the block size is $nb = \left\lceil \frac{n}{P} \right\rceil$. The matrix A is then distributed as shown in Figure 5. If

part one of the algorithm was executed first, the data now must be redistributed to fit this scheme (the cost of this redistribution is neglectable).

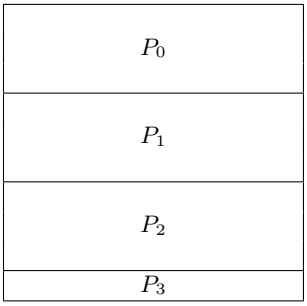

Fig. 5. Distribution of matrix entries for part two (nb rows and n columns per block)

The matrix-matrix-product is now computed as follows: Every process stores a horizontal block of matrix entries, both for the first and the second matrix. Now the dot products of the rows of the first matrix with the columns of second matrix have to be computed. Since every process already stores complete rows of both matrices, it only needs the missing data for the columns of the second matrix. To keep memory demands low, the computation is divided into several steps, where in each step the data for a $n \times nb$ block of the second matrix is broadcasted between the processes. Once this data has been sent, every process can compute a part of the result matrix for itself (the part of the result matrix computed is a part of the processes horizontal block of the result matrix and thus has not to be communicated in any way). As our timings show, this parallelization is quite efficient (Tables 8 and 9).

Computed with...	P	**real**	**interval**	**complex**	**cinterval**
DotK, K=3	1	432.53	706.16	1598.94	2699.34
	2	213.63	351.24	811.59	1335.43
	4	109.95	178.41	410.44	679.90
	8	61.36	100.33	214.68	349.62

Table 8. Time measurements parallel solvers, $cond = 10^{17}$, $n = 1000$

Computed with...	P	real	interval	complex	cinterval
DotK, K=3	1	1.0	1.0	1.0	1.0
	2	2.02	2.01	1.97	2.02
	4	3.93	3.96	3.90	3.97
	8	7.02	7.04	7.45	7.72

Table 9. Speedup parallel solvers, $cond = 10^{17}$, $n = 1000$

5 Conclusion and Future Work

With the new solvers described in this paper it is possible to compute a verified solution of a dense linear (interval-)system in C-XSC a lot faster than before without losing too much accuracy in the result (and in many practical cases without losing any accuracy at all compared to the original toolbox solvers). The speed of the new serial solver now compares very well to the speed of the corresponding Inlab solver but gives much better numerical results. In some cases it was not possible to compute a verified result using Intlab, whereas our new solver worked very satisfactory.

To attack very large linear systems some parallelization has to be done. To the authors knowledge there are no parallel selfverifying solvers available as open source software world wide. Intlab also does not support such an approach. The parallel version of our new solvers allows to compute the solution of very large dense systems (see also [17]) and it has been shown that it is very efficient. All software described in this paper will be made available as open source.

Additionally, our DotK classes are designed and integrated in C-XSC such that they and our matrix-matrix-product routines using BLAS can also be used in other C-XSC programs. The long accumulator (dotprecision variable) of C-XSC may be replaced by a DotK object to gain speed in dot product computations. But please keep in mind that the best way still would be to have hardware support for exact dot products [14]. This gave highest accuracy and optimal speed simultaneously!

The solvers presented in this paper are intended to solve dense systems and do not make use of any special structure of the matrix A (banded, sparse, symmetric, positive definite, Toeplitz, Hankel, circulant, ...). While such systems of course can also be solved using our solvers, this approach would be quite inefficient and memory consuming. Thus, as further work, special solvers for these cases and especially for sparse systems will be investigated in general and implemented in C-XSC.

Acknowledgement: We would like to thank Gerd Bohlender, Werner Hofschuster, Rudi Klatte, and Mariana Kolberg for many fruitful discussions on the topic of this paper.

References

1. Downloads: C-XSC library: http://www.math.uni-wuppertal.de/ xsc/xsc/cxsc.html Solvers: http://www.math.uni-wuppertal.de/ xsc/xsc/cxsc_software.html
2. L.S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley: An Updated Set of Basic Linear Algebra Subprograms (BLAS). ACM Trans. Math. Soft., 28-2 (2002), pp. 135–151.
3. Blackford, L. S. and Choi, J. and Cleary, A. and D'Azevedo, E. and Demmel, J. and Dhillon, I. and Dongarra, J. and Hammarling, S. and Henry, G. and Petitet, A. and Stanley, K. and Walker, D. and Whaley, R. C.: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, 1997, Philadelphia, PA, ISBN = 0-89871-397-8 (paperback)
4. Bohlender, G.: What do we need beyond IEEE Arithmetic? In Computer Arithmetic and Self-validating Numerical Methods, pp 1-32, Academic Press, San Diego, 1990.
5. Bohlender, G.; Walter, W.; Kornerup, P.; Matula, D.W.; Kornerup, P.; Matula, D.W.: Semantics for Exact Floating Point Operations. Proceedings, 10th IEEE Symposium on Computer Arithmetic, 26-28 June 1991, IEEE, 1991.
6. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math., 18:224, 1971.
7. Grimmer, M.: Selbstverifizierende Mathematische Softwarewerkzeuge im High-Performance Computing. Konzeption, Entwicklung und Analyse am Beispiel der parallelen verifizierten Loesung linearer Fredholmscher Integralgleichungen zweiter Art. Logos Verlag, 2007.
8. Hammer, R.; Hochks, M.; Kulisch, U.; Ratz, D.: Numerical Toolbox for Verified Computing I: Basic Numerical Problems. Springer Verlag, 1993.
9. Hofschuster, W.; Krämer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing. Numerical Software with Result Verification, Lecture Notes in Computer Science, Volume 2991/2004, Springer-Verlag, Heidelberg, pp. 15 - 35, 2004.
10. Hölbig, C.; Krämer, W.: Selfverifying solvers for dense systems of linear equations realized in C-XSC. Technical Report BUW-WRSWT 2003/1, 2003.
11. Hölbig, C.; Krämer, W., Diverio, T.A.: An Accurate and Efficient Selfverifying Solver for Systems with Banded Coefficient Matrix In: Parallel Computing: Software Technology, Algorithms, Architectures and Applications. Elsevier Science B.V., Amsterdam, v.13, pp. 283-290, 2004.
12. Kersten, Tim: Verifizierende rechnerinvariante Numerikmodule. Dissertation, University of Karlsruhe, 1998
13. Klatte, Kulisch, Wiethoff, Lawo, Rauch: C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Heidelberg, 1993. Due to the C++ standardization (1998) and dramatic changes in C++ compilers over the last years this documentation describes no longer the actual C-XSC environment. Please refer to more accurate documentation available from the web site of our research group.
14. Kirchner, R., Kulisch, U.: Hardware Support for Interval Arithmetic. Reliable Computing, Volume 12, Number 3, June 2006 , pp. 225-237(13).
15. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms. Addison Wesley, 1969, vol. 2.
16. Kolberg, M., Fernandes, L. F., Claudio, D.: Dense Linear System: A Parallel Self-verified Solver. International Journal of Parallel Programming, 0885-7458 (Print) 1573-7640 (Online), Springer Netherlands, 2007.

17. Kolberg, M., Krämer, W., Zimmer, M. A Noten on Solving Problem 7 of the SIAM 100-Digit Challenge Using C-XSC BUW-WRSWT 2008/2, 2008.
18. Krämer, W., Kulisch, U., Lohner, R.: Numerical Toolbox for Verified Computing II, Advanced Numerical Problems. Draft, about 400 pages. Available on the web: `http://www.uni-karlsruhe.de/ Rudolf.Lohner/papers/tb2.ps.gz`
19. Kulisch, U.: Computer Arithmetic and Validity - Theory, Implementation. De Gruyter, to appear 2008.
20. Kulisch, U.; Miranker, W.: The arithmetic of the digital computer: A new approach. SIAM Rev., 28(1):1-40, 1986.
21. Kulisch, U.: Die fünfte Gleitkommaoperation für Top-Performance Computer. Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und numerische Algorithmen mit Ergebnisverifikation, 1997.
22. Krawczyk, R.: Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. Computing, 4:187-201, 1969.
23. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM Journal on Scientific Computing, 26:6, 2005.
24. Oishi, S., Tanabe, K., Ogita, T., Rump, S.M., Yamanaka, N.: A Parallel Algorithm of Accurate Dot Product. Submitted for publication, 2007.
25. Rump, S.M.: Intlab - Interval Laboratory. Developments in Reliable Computing, pp. 77-104, 1999.
26. Rump, S.M.: Kleine Fehlerschranken bei Matrixproblemen. Dissertation, University of Karlsruhe, 1980.
27. Stoer, J.; Bulirsch, R.: Introduction to Numerical Analysis. Springer-Verlag, New York, 1980.
28. Zimmer, Michael: Laufzeiteffiziente, parallele Loeser fuer lineare Intervallgleichungssysteme in C-XSC. Master thesis, University of Wuppertal, 2007.