Computing (2012) 94:109–123
DOI 10.1007/s00607-011-0170-z

Solvers for the verified solution of parametric linear systems

Michael Zimmer · Walter Krämer · Evgenija D. Popova

Received: 25 November 2010 / Accepted: 5 November 2011 / Published online: 17 November 2011

Abstract We present a newly developed version of our solvers for the verified solution of dense parametric linear systems, i.e. linear systems whose system matrix and right-hand side depend affine-linearly on parameters that vary inside prescribed intervals. The solvers use our C++ class library for reliable computing, C-XSC. The C-XSC library provides many features, especially easy to handle data types for dense and sparse matrices and vectors and the ability to compute dot products and dot product expressions in arbitrary precision. The new solvers can use either sparse or dense matrices as the coefficient matrices for the parameters. The use of sparse coefficient matrices can result in huge improvements in both performance and memory consumption. BLAS and LAPACK routines are used where applicable, and OpenMP is used for the parallelization on multi-core and multi-processor systems. The solvers also provide the ability to compute not only an outer but also a componentwise inner enclosure of the solution set of the system and to choose between two versions of the algorithm, one being very fast and one giving sharp results and extending the range of solvable systems. We give some examples for parametric linear systems (also from real world examples such as worst-case tolerance analysis of linear electric circuits),

The authors have presented the results of this paper during the SCAN 2010 conference in Lyon, September 2010.

M. Zimmer (✉) · W. Krämer
Universität Wuppertal, Gaußstr. 20, 42097 Wuppertal, Germany
e-mail: zimmer@math.uni-wuppertal.de

W. Krämer
e-mail: kraemer@math.uni-wuppertal.de

E. D. Popova
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences,
Acad. G. Bonchev str., block 8, 1113 Sofia, Bulgaria
e-mail: epopova@bio.bas.bg

give performance measurements of our solvers and also demonstrate that they scale very well when using multiple cores or processors.

Keywords Parameter dependent system · Self-verifying solver · Sparse data structure · C-XSC

Mathematics Subject Classification (2000) 65F05 · 65F50 · 65G20 · 65G99

1 Introduction

In this paper we are interested in finding a verified solution to the linear system

$$A(p)x = b(p)\ ,\ \ p \in [p] \tag{1}$$

where the square matrix A and the vector b depend affine-linearly on parameters p_i, which vary inside prescribed intervals $p_i \in [p_i]$. Computing a verified solution means that the computed result is proved to be an interval enclosure of the exact solution set (2).

In the following we present a new version of our solvers for this problem using our C++ class library C-XSC [7], which is much more efficient than previous versions [15,17], both in terms of runtime and (through the optional use of sparse matrices) in terms of memory requirement. The new version also uses shared-memory parallelization via OpenMP to utilize the potential of modern multi-core processors. Currently, we know of no other software packages for the verified solution of parametric linear systems.

1.1 Notation

In this paper, intervals are used in infimum–supremum representation, that is an interval $[x] = [\underline{x}, \overline{x}]$ describes a set $\{x | x \geq \underline{x}, x \leq \overline{x}\}$. Here, $\underline{x}$ is called the infimum and $\overline{x}$ is called the supremum. On a computer the endpoints of an interval are floating point numbers, i.e. $\underline{x}, \overline{x} \in \mathbb{F}$ where $\mathbb{F}$ is the set of all floating point numbers.

Interval quantities are denoted by square brackets, e.g. $[x]$. The set of all real intervals is denoted with $\mathbb{IR}$, the set of all complex intervals with $\mathbb{IC}$. The midpoint of a real interval is defined as

$$mid([x]) = \frac{\underline{x} + \overline{x}}{2}$$

and the radius of a real interval is defined as

$$rad([x]) = \frac{\overline{x} - \underline{x}}{2}.$$

Interval matrices and vectors are matrices and vectors whose elements are intervals. The midpoint and radius are computed element-wise, i.e. $mid([A]) = (mid([a_{ij}]))$

and $rad([A]) = (rad([a_{ij}]))$ for $[A] \in \mathbb{IR}^n$ or $[A] \in \mathbb{IR}^{m \times n}$. For a thorough introduction to interval arithmetic we refer to [1].

A $\diamond$ denotes the interval hull of the following term, e.g. $\diamond(b - Ax)$. On the computer a tight interval enclosure is computed in these cases by using the higher precision arithmetic of C-XSC (see Sect. 2) or by manipulation of the rounding mode.

1.2 Outline

Section 2 gives a short overview of the C-XSC library, with a special focus on the most important parts used for this work.

Section 3 then deals with the solvers themself. First, in Sect. 3.1, the theory behind the verified solution of parametric linear systems is briefly explained. Then, in Sect. 3.2, we give an overview of the actual implementation in our solvers.

Next, Sect. 4 deals with some test cases and also a small real-world example. We give time and speed up measurements and some numerical results. Finally, Sect. 5 contains some final remarks and an outlook on future work.

2 The C-XSC library

The C-XSC (eXtended Scientific Computing) library [7] is a C++ class library for reliable computing. It provides many useful data types and functionalities, especially matrix and vector datatypes and the possibility to compute dot products and dot product expressions in arbitrary (and even maximum) precision.

The basic data types of C-XSC are `real` (a wrapper class for `double`), `interval` (for real intervals), `complex` and `cinterval` (for complex intervals). Based on these several data types for dense and sparse matrices and vectors are defined. The matrix/vector data types of C-XSC use operator overloading for ease of use and provide several useful features, such as the possibility to cut out slices of matrices and vectors (which reference the original data, so they can be used to overwrite portions of a matrix) and to freely reassign the index range of such types (1-based, 0-based, etc.).

Furthermore, the precision for explicitly or implicitly computed dot products using the overloaded operators, for example in matrix–matrix products, can be set by the user with the global variable `opdotprec`. For a value of `1`, standard floating point operations are used (with rounding mode manipulation for intervals). For a value of `0`, dot products in maximum precision (at most one rounding away from the true result) using a long accumulator [12,13] are used, while for values of `2` or larger computations are done in simulated higher precision using the DotK algorithm [14,26].

The data types for sparse matrices have only recently been introduced. They are based on the compressed column storage format, which is used in most software packages for sparse matrices and thus allows easy interfacing between different software packages. They have the same benefits as the dense types and are very efficient (basic operations are about as fast as sparse matrices in Matlab). For more information see [24].

The focus of C-XSC for a long time was on rich functionality and on accuracy while neglecting the performance of operations. In recent versions we have begun to include

many features that can optionally be activated to (sometimes drastically) increase performance. One of these features is the option to use highly optimized BLAS-routines [4] for all dot product operations, especially for matrix–matrix products. These can be activated simply by using the compiler switch `-DCXSC_USE_BLAS` and linking to a CBLAS library (the precision must be set to `1`). In the future we also plan to provide a complete interface to LAPACK [2]. As of today however, LAPACK support is only implemented for the matrix inversion routine needed for the solvers described in his paper. The compiler switch `-DCXSC_USE_LAPACK` has to be activated and one must link to a LAPACK library. Further information about using C-XSC in high-performance computing can be found in [25].

C-XSC provides many more features which are not important for this work. For a more complete overview, see [7].

3 Verified solution of parametric linear systems

In this Section, we explain the theoretical aspects of the verified solution of parametric linear systems and then explain our implementation in more detail.

3.1 Theory

Solving the problem $A(p)x = b(p)$ with $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n$ that depend affine-linearly on parameters $p \in \mathbb{R}^k$, where p varies in $[p] \in \mathbb{IR}^k$ means finding verified bounds for the interval hull $\diamond(\Sigma^p)$ of the solution set

$$\Sigma^p = \Sigma(A(p), b(p), [p]) := \{x \in \mathbb{R}^n \mid \exists p \in [p] : A(p)x = b(p)\} \tag{2}$$

if the solution set is bounded and non-empty. Note that using the corresponding non-parametric matrix $A([p]) := \diamond\{A(p)|p \in [p]\}$ for the solution set of the corresponding non-parametric interval system

$$\Sigma(A([p]), b([p])) := \{x \in \mathbb{R}^n \mid \exists A \in A([p]), \exists b \in b([p]) : Ax = b\} \tag{3}$$

it holds

$$\Sigma^p \subseteq \Sigma \tag{4}$$

and that in general Σ will be much broader than Σ^p, since the corresponding non-parametric interval system treats each occurrence of a parameter p_i as independent of the other occurrences.

Computing the actual solution set of a parametric linear system or the exact interval hull of it is NP-hard. Instead we try to find a tight enclosure of the interval hull using an iteration method by Rump [19] based on the Krawczyk operator [11]. First we rewrite the system matrix as $A(p) = (a_{ij}(p))$ and the right-hand side as $b(p) = (b_i(p))$ with

$$a_{ij}(p) = a_{ij}^{(0)} + \sum_{\nu=1}^{k} p_\nu a_{ij}^{(\nu)}, \tag{5}$$

$$b_i(p) = b_i^{(0)} + \sum_{\nu=1}^{k} p_\nu b_i^{(\nu)}. \tag{6}$$

Now we can rewrite (1) as

$$(A^{(0)} + \sum_{\nu=1}^{k} p_\nu A^{(\nu)})x = b^{(0)} + \sum_{\nu=1}^{k} p_\nu b^{(\nu)},\ p \in [p] \in \mathbb{IR}^k \tag{7}$$

and with $[u] = [v] \in \mathbb{IR}^n$ use the Gauss–Seidel iteration

$$1 \le i \le n : y_i := \{\diamond([z] + [C] \cdot [u])\}_i,\ [u] := \{y_1, \ldots, y_{i-1}, v_i, \ldots, v_n\}^T \tag{8}$$

to finally bound the error of an approximate solution $\tilde{x}$ of the system by $[y] \in \mathbb{IR}^n$ (normally one computes $\tilde{x}$ by solving the midpoint system $A(mid([p]))x = b(mid([p]))$ approximately using floating point operations). The quantities $[z]$ and $[C] = [C(p)]$ in (8) are defined as

$$[z] := R(b^{(0)} - A^{(0)}\tilde{x}) + \sum_{\nu=1}^{k} (R(b^{(\nu)} - A^{(\nu)}\tilde{x}))[p_\nu], \tag{9}$$

$$[C(p)] := I - RA^{(0)} - \sum_{\nu=1}^{k} (RA^{(\nu)})[p_\nu]. \tag{10}$$

In (9) and (10) R is an approximate inverse of $A(mid([p]))$ (which usually is also used to compute $\tilde{x} \approx Rb(mid([p]))$) and I is the identity matrix. Note that using a LU-decomposition instead of an approximate inverse would require solving two triangular systems with an interval right-hand-side, which is not a trivial task. Simple forward and backward substitution in general leads to severe overestimation of the result. For more details see Rump [19].

Theorem 1 *Let $A(p)x = b(p)$, $p \in [p] \in \mathbb{IR}^k$ be as given in (7). Define $[z]$ and $[C]$ by (9), (10) and $[y]$ and $[v]$ by (8). If it holds that*

$$[y] \subseteq [v]^\circ$$

($[y]$ lies in the interior of $[v]$) then every matrix $A(p)$, $p \in [p]$ is regular and for every $p \in [p]$ the unique solution lies in $\tilde{x} + [y]$.

For a proof see [18]. Note that with Theorem 1 one can only verify strong regularity of the system matrix, which is a weaker but sufficient condition for its regularity, or else make no assertion at all (the matrix might be singular, but it could also be badly conditioned or not strongly regular). The verification of the singularity of a matrix is not possible without exact computations [22].

Definition 1 An interval matrix $[A] \in \mathbb{IR}^{n\times n}$ is called regular if every point matrix $A \in [A]$ is regular. It is called strongly regular if it is regular and

$$\rho(|(mid([A]))^{-1}| \cdot rad([A])) < 1$$

(ρ denotes the spectral radius).

For the definition of strong regularity for parametric matrices see Popova [18].

Originally Rump proved Theorem 1 with the iteration matrix $[C] = [C([p])] := I - RA([p])$. This method is usually faster in terms of computing time but requires that the non-parametric matrix $A([p])$ is strongly regular. A parametric matrix might be strongly regular while the corresponding non-parametric one is not. The alternative iteration matrix $[C(p)]$ does require strong regularity of the parametric matrix $A(p)$ and in general leads to better (tighter) results and solvability of a broader class of parametric systems. In our implementation, both methods can be used (see Sect. 3.2).

If the iteration in Theorem 1 reached an inclusion in the interior then with $[D] := [C][y]$ an inner inclusion of the interval hull of the solution set can be computed [15,16] as

$$[\tilde{x} + \underline{z} + \overline{D}, \tilde{x} + \overline{z} + \underline{D}] \subseteq [inf\,\Sigma^p, sup\,\Sigma^p]. \tag{11}$$

In an implementation one must compute a lower bound on $\overline{z}$ and an upper bound on $\underline{z}$. Also, since (11) uses exact operations, correct rounding has to be used on the machine. The inner enclosure can be used as a reference on the quality of the computed outer enclosure. Our implementation also supports the computation of an inner enclosure (see Sect. 3.2).

3.2 Implementation

Based on the theory described in Sect. 3.1, Algorithm 1 by Rump [19] with the modification by Popova [15] (optional use of the sharp iteration matrix $[C(p)]$) is used in the implementation.

The cost of Algorithm 1 is dominated by the computation of $[C]$ and $[z]$ and the computation of the approximate inverse R. The cost for the computation of R is $\mathcal{O}(n^3)$ and does not depend on the number of parameters k or the sparsity of the coefficient matrices.

The cost for the computation of $[z]$ is dominated by $2(k+1)$ matrix–vector-products (with a point matrix and an interval vector, so this corresponds to roughly $6(k+1)$ floating point matrix vector products [20]). The cost for the computation of $[C]$ depends on the used iteration matrix. When using $[C([p])]$, the cost is dominated by the product of a point matrix and an interval matrix, which takes 3 floating point matrix–matrix products [20]. When using the sharp iteration matrix $[C(p)]$, it is dominated by the computation of the enclosure of $k + 1$ point matrix products, which take 2 floating point matrix–matrix products each.

If the coefficient matrices are sparse, the products in the computation of $[C]$ and $[z]$ can take advantage of the sparsity and are much faster. The cost for the defect

Compute approximate inverse R of $mid(A([p]))$
Compute approximate solution $\widetilde{x} := Rb(mid([p]))$
// Defect iteration
repeat
  $\widetilde{x} := \widetilde{x} + R(b(mid([p])) - A(mid([p]))\widetilde{x})$
until $\widetilde{x}$ *accurate enough or max iterations reached*
// Compute enclosures of the residuum and the iteration matrix
$[z] := R \diamond (b(p) - A(p)\widetilde{x})$ as in (9)
if *Compute sharp iteration matrix* **then**
  $[C] := \diamond\{I - RA(p) \mid p \in [p]\}$ as in (10)
else
  $[C] := \diamond(I - RA([p]))$ using $A([p])$
// Verification
$[y] := [z]$
repeat
  // Blow up iterate to "catch" nearby fixed point
  $[y_A] := blow([y], \epsilon)$
  Compute $[y]$ by (8) with $[u] = [v] = [y_A]$
until $[y] \subset int([y_A])$ *or max iterations reached*
// Check result
if $[y] \subset int([y_A])$ **then**
  Unique solution in $x \in \widetilde{x} + [y]$
else
  Algorithm failed, $A(p)$ is not strongly regular

Algorithm 1: Algorithm used for the implementation

iteration and the verification are negligible, and normally two or three iteration steps are sufficient.

Our new implementation is based on the previous work described in [15,17]. The old solver uses an older C-XSC version and performs all computations in maximum precision using the long accumulator, which is very slow. In this new version, we focus on improving the runtime and memory requirements of the solver with the new features of C-XSC described in Sect. 2.

BLAS and LAPACK The first step is to speed up the most computationally intensive tasks, which are the computation of the approximate inverse R and especially the computation of $[C]$, which requires either $k + 1$ matrix–matrix products with real point matrices of dimension $n \times n$ if the sharp iteration matrix $[C(p)]$ is used or one matrix–matrix product of a real point matrix and an interval matrix, each of dimension $n \times n$ if the iteration matrix $[C([p])]$ is used. For both steps, double precision computations are sufficient for most practical cases (if the system matrix is very ill conditioned with a condition number of 10^{16} or higher a different approach as discussed in [10,21] needs to be taken) and thus the highly optimized routines of BLAS and LAPACK can be used. This also has the added benefit that a modern BLAS and LAPACK implementation will already be multi-threaded an can thus directly make use of multi-core and multi-processor systems.

However, for the computation of $[C]$ interval enclosures of the exact result are needed. To achieve this using the approximate computations of the BLAS' `_gemm` routine, the rounding mode of the processor needs to be switched (this can be done

using a function from the standard C library). Then an enclosure for the product of two real point matrices can be computed with Algorithm 2. The respective algorithms for interval and complex matrices are given in [20].

```
// Round downwards
setround(-1);
// Compute infimum
SetInf(C, A*B);
// Round upwards
setround(1);
// Compute supremum
SetSup(C, A*B);
```

Algorithm 2: Computation of a machine interval enclosure $[C]$ of a floating-point matrix product $A \cdot B$ using BLAS

OpenMP Beside the usage of multi-threaded BLAS and LAPACK libraries, OpenMP is used to implement further parallelizations. One of these parallelizations is the computation of $[z]$. This is a straightforward distribution of a for-loop among all threads.

Also, the computation of $[C]$ (for both $[C(p)]$ and $[C[p]]$) is further parallelized, independent of the parallelization already implemented in the used BLAS library. This is done by letting each thread compute one $l \times n$ block of the matrix $[C]$, where $l = \lfloor \frac{n}{nt} \rfloor$, nt being the number of threads used (if $n \mod nt \neq 0$, the remaining part of the matrix is computed by the master thread). Accessing slices of a matrix (both for read and for write access) in C-XSC is easy using the `()`-Operator. This approach of course has advantages when not using BLAS routines, but also with BLAS routines a few percent performance are gained, since the matrix additions are also parallelized.

Sparse coefficient matrices In practice, the coefficient matrices $A^{(i)}, i = 0, \ldots, k$, are often sparse. Storing them in a dense matrix type leads both to increased memory and computing time requirements, especially when using the sharp iteration matrix $[C(p)]$. Therefore, we supply two different versions of the solver, one where the coefficient matrices are dense and one where they are sparse, using the sparse matrix data types of C-XSC (see Sect. 2). Note however that the system matrix $A(p)$ is still supposed to be dense. While the solvers will of course work with sparse systems, the used algorithm is inefficient in these cases because of the need to compute an approximate inverse R, which in general will be dense.

Further features The interval parameters can also be complex intervals. This can for example be useful when dealing with linear electrical circuits (see Sect. 4). The theory from Sect. 3.1 can be applied straightforward for this case, see [17] for more details. Other features of the solver include choosable dot product precision (used for the residual iteration and the computation of $[z]$), the ability to compute inner enclosures (using (11)) and, as said before, the ability to choose between the sharp iteration

matrix $[C(p)]$, giving better results and broadening the range of solvable systems, and the iteration matrix $[C([p])]$ leading to faster computations, but in general also to overestimations of the solution sets.

Data format. The coefficient matrices have to be given to the solvers in a special format. When using dense coefficient matrices for the k parameters, one matrix of dimension $(k+1)n \times n$ is passed to the solver, where the coefficient matrix $A^{(i)}$ is stored in the slice between row $(i-1)n+1$ and row $i \cdot n$.

For sparse coefficient matrices this approach is not used, since cutting slices out of sparse matrices is a lot more costly due to the underlying data structure. Instead, one passes a `vector` from the Standard Template Library (which is basically a managed array) which contains all the coefficient matrices.

In all cases, the right-hand side is stored as a matrix of dimension $n \times (k+1)$, where $b^{(i)}$ can be found in column $i+1$ of the matrix.

4 Examples and tests

In this section we give a few examples for parametric linear systems and show some time and speed up measurements and comparisons to the old solver. First we take a look at a small example system that shows the benefits of using the sharp iteration matrix $[C(p)]$:

$$A(p) = \begin{pmatrix} 3 & p & p \\ p & 3 & p \\ p & p & 3 \end{pmatrix}, \quad b(p) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad [p] = [0, 2].$$

The parametric matrix is strongly regular for all $p \in [p]$. However, the non-parametric matrix $A([p])$ is not strongly regular since

$$\rho(|mid(A([p]))^{-1}|rad(A([p]))) = \frac{6}{5} > 1$$

and thus using the iteration matrix $[C([p])]$ does not yield a result. Using the sharp iteration matrix $[C(p)]$ however we get the outer enclosure

$$x = \begin{pmatrix} [-0.3327234817713, 1.1327234817713] \\ [-0.7961011636355, 0.5961011636355] \\ [-0.7849912184268, 0.5849912184268] \end{pmatrix}.$$

Some more examples comparing the properties of the algorithm when using the iteration matrix $[C(p)]$ or $[C([p])]$, respectively, can be found in [16].

Next we look at another small example, the following 2×2 symmetric system given by Behnke [3] (see also [19]), for which the interval hull of the solution set is known:

$$[A] = \begin{pmatrix} 3 & [1,2] \\ [1,2] & 3 \end{pmatrix}, \quad [b] = \begin{pmatrix} [10, 10.5] \\ [10, 10.5] \end{pmatrix}.$$

When viewing this as a symmetric problem, there is a dependency between the two interval entries $[1, 2]$ in $[A]$, and thus we can model this problem as a parametric system using one parameter for the two interval entries in $[A]$, i.e.

$$A(p) = \begin{pmatrix} 3 & p_1 \\ p_1 & 3 \end{pmatrix}, \quad b(p) = \begin{pmatrix} p_2 \\ p_3 \end{pmatrix}, \quad [p] = ([1, 2], [10, 10.5], [10, 10.5])^T.$$

Computing an inner and outer enclosure with our solvers gives the result

$$\begin{pmatrix} [2.075, 2.480] \\ [2.077, 2.479] \end{pmatrix} \subseteq \diamond\Sigma = \begin{pmatrix} [1.810\ldots, 2.688\ldots] \\ [1.810\ldots, 2.688\ldots] \end{pmatrix} \subseteq \begin{pmatrix} [1.618, 2.938] \\ [1.631, 2.925] \end{pmatrix}.$$

To show how easy the solvers are to use, we give example code for this problem in Listing 1.

Listing 1 Source code for the symmetric example

```
int n=2;
rmatrix Ap(n*4,n); //Coefficient matrices
rmatrix bp(n,4); //Right hand side
ivector p(4); //Parameter vector

//Fill coefficient matrices
Ap[1][1]=3; Ap[2][2]=3;
Ap[3][2]=1; Ap[4][1]=1;
bp[1][3]=1; bp[2][4]=1;

//Define parameter intervals
p[1]=1;
p[2]=interval(1,2);
p[3]=p[4]=interval(10,10.5);

ivector xx, yy; //Vectors for outer and inner enclosure

//Solve system using sharp iteration matrix and
//computing outer and inner enclosures
ParLinSolve(Ap, bp, p, xx, true, yy, Err, 0);
```

Table 1 New solvers, $Q(2, p)$ system, multi-core platform, time in s

Solver	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
Dense, 1 Core	0.242	2.272	9.625	26.958	61.080
Dense, 2 Cores	0.143	1.315	5.374	14.187	32.096
Dense, 4 Cores	0.094	0.814	3.251	8.907	19.721
Dense, 8 Cores	0.118	0.654	3.083	6.620	16.839
Sparse, 1 Core	0.131	1.029	3.460	8.200	16.449
Sparse, 2 Cores	0.072	0.527	1.770	4.142	8.240
Sparse, 4 Cores	0.044	0.284	0.917	2.160	4.307
Sparse, 8 Cores	0.040	0.164	0.540	1.217	2.379

For the performance tests and comparisons to the old solver we use the so-called $Q(2, p)$ system, which is defined as follows:

$$q_{ij}(2, p) := \begin{cases} p_j & i \leq j \\ 0 & i = j + 2, \\ 1 & \text{otherwise} \end{cases}$$

$$b(p) = (p_1, \ldots, p_n)^T,$$

$$p_k \in [k \pm k \cdot 0.05], \quad k = 1, \ldots, n.$$

This system has n parameters, which means that $n + 1$ (sparse) coefficient matrices are needed, leading to huge memory and computing time requirements for higher dimensions when using dense coefficient matrices. We use this system to compare the runtime of the old solver, the new solvers with dense and sparse coefficient matrices, and show the speed up of the new solvers. All tests were done on a machine with two Intel Xeon 2.26 GHz processors with four cores each and 24GB of RAM running openSuse Linux 11.1. We used the Intel compiler 11.1 and the MKL 10.2.

First, Table 1 shows the time measured for the dense and sparse solvers for dimensions $n = 100, \ldots, 500$ and 1,2,4 and 8 core(s), respectively. The timings show that using sparse coefficient matrices for this test system gives huge performance gains. The memory requirements are of course also a lot smaller for the sparse case. The timings also demonstrate the efficiency of the parallelization. Table 2 shows the speed up gained through the parallelization of the solvers. The speed ups are good throughout, especially for the sparse version, which approaches a nearly ideal speed up for large enough problem sizes in relation to the number of cores. The speed ups for the dense version are a little worse, but still good, especially for 2 cores.

In Table 3, we compare the time measured for the new solvers with the old solver for dimension $n = 100, \ldots, 500$. Here, we also compare the quality of the results. As the results show, the new solvers are not only a lot more efficient, both when using dense coefficient matrices (as the old solvers do) and especially when using sparse coefficient matrices, but also achieve results of about the same quality as the old solvers. The small variances are mostly due to a different order of operations.

Table 2 New solvers, $Q(2, p)$ system, speed-up

Solver	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
Dense, 1 Core	1.0	1.0	1.0	1.0	1.0
Dense, 2 Cores	1.695	1.728	1.791	1.900	1.903
Dense, 4 Cores	2.560	2.790	2.960	3.027	3.097
Dense, 8 Cores	2.042	3.475	3.122	4.072	3.627
Sparse, 1 Core	1.0	1.0	1.0	1.0	1.0
Sparse, 2 Cores	1.824	1.952	1.954	1.979	1.996
Sparse, 4 Cores	3.003	3.620	3.774	3.796	3.819
Sparse, 8 Cores	3.286	6.289	6.408	6.740	6.951

Table 3 Comparison between old and new solvers, $Q(2, p)$ system

What?	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
Old solvers, time in s	2.107	30.316	214.850	911.413	2762.350
New dense, time in s	0.242	2.272	9.625	26.958	61.080
New sparse, time in s	0.131	1.029	3.460	8.200	16.449
Old solvers, Rel. Err.	$3.24E+012$	$1.42E+012$	$6.74E+011$	$7.62E+011$	$4.92E+011$
New dense, Rel. Err.	$1.89E+012$	$2.05E+013$	$2.27E+012$	$1.11E+012$	$3.94E+011$
New Sparse, Rel. Err.	$4.87E+012$	$1.33E+012$	$1.31E+012$	$5.27E+011$	$9.09E+011$

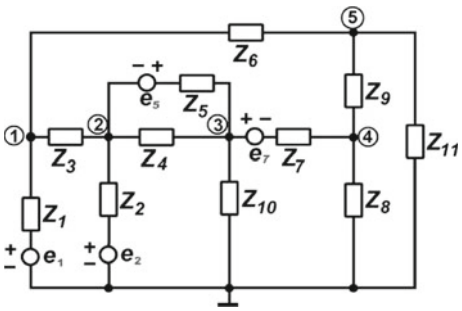

Fig. 1 Electrical circuit with five nodes and complex parameters $p_i = 1/Z_i$

Finally, we show a practical example for a complex parametric linear system, which occurs in a worst-case tolerance analysis of linear electrical circuits. The use of interval methods in this area has been the subject of many papers, cf. [6,8,9]. Here, we take a look at an example from [8], where the electrical circuit shown in Fig. 1 is given. We are interested in finding bounds for the node voltages $V_i, i = 1, \ldots, 5$.

The parameters have the values

$$
\begin{aligned}
&e_1 = e_2 = 100V, \quad e_5 = e_7 = 10V,\\
&Z_j = R_j + \imath X_j \in \mathbb{C},\ R_j = 100\Omega,\ X_j = \omega L_j - \frac{1}{\omega C_j},\ j = 1, \ldots, 11,\\
&\omega = 50, \quad X_{1,2,5,7} = \omega L_{1,2,5,7} = 20, \quad X_3 = \omega L_3 = 30,\\
&X_4 = -\frac{1}{\omega C_4} = -300, \quad X_{10} = -\frac{1}{\omega C_{10}} = -400, \quad X_{6,8,9,11} = 0
\end{aligned}
$$

which lead to the complex linear parametric system

$$
\begin{pmatrix}
\frac{1}{Z_1}+\frac{1}{Z_3}+\frac{1}{Z_6} & -\frac{1}{Z_3} & 0 & 0 & -\frac{1}{Z_6}\\
-\frac{1}{Z_3} & \frac{1}{Z_2}+\frac{1}{Z_3}+\frac{1}{Z_4}+\frac{1}{Z_5} & -\frac{1}{Z_4} & 0 & 0\\
0 & -\frac{1}{Z_4} & \frac{1}{Z_4}+\frac{1}{Z_5}+\frac{1}{Z_7}+\frac{1}{Z_{10}} & -\frac{1}{Z_7} & 0\\
0 & 0 & -\frac{1}{Z_7} & \frac{1}{Z_7}+\frac{1}{Z_8}+\frac{1}{Z_9} & -\frac{1}{Z_9}\\
-\frac{1}{Z_6} & 0 & 0 & -\frac{1}{Z_9} & \frac{1}{Z_6}+\frac{1}{Z_9}+\frac{1}{Z_{11}}
\end{pmatrix}
\times
\begin{pmatrix} V_1\\ V_2\\ V_3\\ V_4\\ V_5 \end{pmatrix}
=
\begin{pmatrix}
\frac{e_1}{Z_1}\\
\frac{e_2}{Z_2} - \frac{e_5}{Z_5}\\
\frac{e_5}{Z_5} + \frac{e_7}{Z_7}\\
-\frac{e_7}{Z_7}\\
0
\end{pmatrix}
$$

where

$$
\begin{aligned}
Z_1 = Z_2 = Z_5 = Z_7 &= 110 + i20\\
Z_3 &= 100 + i30\\
Z_4 &= 100 - i300\\
Z_6 = Z_8 = Z_9 = Z_{11} &= 100\\
Z_{10} &= 100 - i400
\end{aligned}
$$

and as complex interval parameters we use $p_i = \frac{1}{Z_i}$ with a tolerance of 10%. Computing this system with our solvers gives the correct enclosure

$$
\begin{pmatrix}
[4.9021900635077813E+001, 6.3782338144818816E+001]\\
\quad + i[-6.8400205603540068E+000, -1.0277087785239471E+000]\\
[4.0273840294108360E+001, 5.4752678432853387E+001]\\
\quad + i[-7.9124035354960772E+000, -1.5582600826862842E+000]\\
[1.3138301166837424E+001, 2.0989859759659016E+001]\\
\quad + i[-5.8195849568360592E-001, 6.5691430975830159E+000]\\
[5.6469311369255450E+000, 1.4149265965229715E+001]\\
\quad + i[-1.1666539312767415E+000, 2.7739633213572846E+000]\\
[1.6367827470687267E+001, 2.7832317823330030E+001]\\
\quad + i[-2.8213369595936980E+000, 7.3453030999456104E-001]
\end{pmatrix}
$$

for the voltages $V_i, i = 1, \ldots, 5$.

5 Conclusion and future work

We showed a new optimized implementation of our verified solvers for real parametric and complex parametric linear systems. These new versions offer much better performance without sacrificing any significant accuracy. Optionally, sparse coefficient matrices can be used, which increase the performance gain even more and reduce the memory requirements. The solvers are efficiently parallelized using OpenMP and a multi-threaded BLAS/LAPACK library. They are Open Source software and freely available [23] under the GNU General Public License.

In the future, we plan to implement an MPI version of the solvers for distributed memory systems, using the same methods as in our already available parallel verified solver for non-parametric linear systems [10].

References

1. Alefeld G, Herzberger J (1983) Introduction to interval computations. Academic Press, New York
2. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK Users' guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia
3. Behnke H (1989) Die Bestimmung von Eigenwertschranken mit Hilfe von Variationsmethoden und Intervallarithmetik. Dissertation, Inst. für Mathematik, TU Clausthal
4. Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (2002) An updated set of basic linear algebra subprograms (BLAS). ACM Trans Math Softw 28(2):135–151
5. Cuyt A et al (eds) (2009) Numerical validation in current hardware architectures. In: Springer lecture notes in computer science (LNCS), vol 5492. Springer, Berlin
6. Dreyer A (2005) Interval analysis of analog circuits with component tolerances. Shaker Verlag, Aachen, Germany. Doctoral thesis, TU Kaiserslautern
7. Hofschuster W, Krämer W (2004) C-XSC 2.0: a C++ library for extended scientific computing. Numerical software with result verification. Lecture notes in computer science, vol 2991/200. Springer, Heidelberg, pp 15–35
8. Kolev L (1993) Interval methods for circuit analysis. World Scientific, Singapore
9. Kolev L (2002) Worst-case tolerance analysis of linear DC and AC electric circuits. IEEE Trans Circ Syst 49(12):1–9
10. Krämer W, Zimmer M (2009) Fast (parallel) dense linear interval system solvers in C-XSC using error free transformations and BLAS. In: Cuyt A et al (eds) (2009) Numerical validation in current hardware architectures. Springer lecture notes in computer science (LNCS), vol 5492, pp 230–249
11. Krawczyk R (1969) Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. Computing 4:187–201
12. Kulisch U, Miranker W (1986) The arithmetic of the digital computer: a new approach. SIAM Rev 28(1):1–40
13. Kulisch U (1997) Die fuenfte Gleitkommaoperation fuer Top-Performance Computer. Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und numerische Algorithmen mit Ergebnisverifikation
14. Ogita T, Rump SM, Oishi S (2005) Accurate sum and dot product. SIAM J Sci Comput 26:6
15. Popova E (2004) Parametric interval linear solver. Numer Algorithms 37(1–4):345–356
16. Popova E, Krämer W (2007) Inner and outer bounds for the solution set of parametric linear systems. J Comput Appl Math 199(2):310–316
17. Popova E, Kolev L, Krämer W (2010) A solver for complex-valued parametric linear systems. Serdica J Comput 4(1):123–132
18. Popova E (2004) Generalization of a parametric fixed-point iteration. PAMM: Proc Appl Math Mech 4:680–681

19. Rump S (1994) Verification methods for dense and sparse systems of equations. In: Herzberger J (ed) Topics in validated computations. N. Holland, pp 63–135
20. Rump SM (1999) Intlab: interval laboratory. In: Developments in reliable computing, pp 77–104
21. Rump SM (1980) Kleine Fehlerschranken bei Matrixproblemen. PhD thesis, University of Karlsruhe
22. Rump SM (2010) Verification methods: rigorous results using floating-point arithmetic. Acta Numer 19:287–449
23. Solver: http://www2.math.uni-wuppertal.de/org/WRST/xsc/cxsc_software.html. Accessed 1 Nov 2011
24. Zimmer M, Krämer W, Hofschuster W (2009) Sparse matrices and vectors in C-XSC. Preprint 2009/7, Universität Wuppertal
25. Zimmer M, Krämer W, Hofschuster W (2009) Using C-XSC in high performance computing. Preprint 2009/5, Universität Wuppertal, 2009. Revised version submitted
26. Zimmer M, Krämer W, Bohlender G, Hofschuster W (2010) Extension of the C-XSC Library with scalar products with selectable accuracy. Serdica J Comput 4(3):349–370