To appear in SIMAX, 2026

VERIFIED ERROR BOUNDS FOR SPARSE SYSTEMS PART II: INERTIA-BASED BOUNDS, LEAST SQUARES AND NONLINEAR SYSTEMS*

SIEGFRIED M. RUMP†

Abstract. Verification methods provide mathematically correct error bounds for the solution of a numerical problem. That includes the proof of solvability of the problem and often uniqueness of the solution within the computed bounds. There are many verification methods for standard problems in numerical analysis, including linear and nonlinear systems of equations, matrix decompositions, eigenproblems, local and global optimization, ordinary and partial differential equations. Many of those verification methods are included in INTLAB, the Matlab/Octave toolbox for reliable computing. Despite many efforts, the solution of general sparse linear systems was an open problem.

In Part I of this note we presented an algorithm computing verified bounds for the solution of general real or complex sparse linear systems with condition number up to the limit 10^{16} in double precision. That algorithm splits into three subalgorithms for symmetric positive definite, symmetric indefinite and general input matrix A. It is based on a mathematically correct lower bound on the smallest singular value $\sigma_{\min}(A)$.

In this Part II we use a method published by the author in 1995 based on the inertia of a symmetric matrix. A key point is, as in Part I, a factorization L_1L_2 such that L_1 and L_2 have identical sets of singular values with the smallest one close to $\sigma_{\min}(A)^{1/2}$. Numerical evidence suggests that the method is often a little slower than that in Part I, however, more robust. That means, for some of the few cases where the method in Part I could not compute verified bounds successfully, the method in this Part II succeeded.

Furthermore. we show how to compute inclusions with almost maximal accuracy for all entries, i.e., all bounds differ by few bits. That is based on a fast method to compute accurate approximations and bounds for extremely ill-conditioned dot products using a very efficient Matlab implementation.

We show how to treat a system with nearly symmetric input matrix in the sense that $\|A^T - A\|$ is small compared to $\|A\|$. That is useful if symmetry is lost due to rounding and may improve the performance significantly. We show that Algorithm "verifySparseNearSym1" in Part I of the note is correct for unsymmetric input matrix without change.

Moreover, algorithms are given to compute verified error bounds for a least squares problem and an underdetermined system of linear equations with sparse input matrix. Furthermore, we show how to compute verified error bounds for the solution of a system of nonlinear equations with sparse Jacobian. In all cases the algorithms for sparse square linear systems of Part I and this Part II can be used.

Key words. sparse linear systems, nonlinear systems, verification methods, least squares, underdetermined linear systems, inertia, mathematically correct error bounds, accurate dot products, INTLAB

MSC codes. 65G20, 65F99

1. Introduction.

This paper in two parts presents verification methods for the solution of a linear system with sparse input matrix, i.e., the computation of rigorous error bounds for the solution. The bounds are computed in pure floating-point arithmetic and they are true with mathematical certainty. That includes the proof of solvability of the problem and uniqueness of the solution within the computed bounds. For overviews on verification methods cf. [28, 43, 31] and the literature cited over there. Many verification algorithms are included in INTLAB [41], the Matlab/Octave toolbox for reliable computing.

*Submitted to the editors DATE.

†Institute for Reliable Computing, Hamburg University of Technology, Am Schwarzenberg-Campus 3, Hamburg 21073, Germany, and Faculty of Science and Engineering, Waseda University, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan (rump@tuhh.de).

As mentioned in Part I, a verification method for sparse linear systems is the challenge formulated in [43, Challenge 10.15]. That is solved with the algorithms in Parts I and II of this note.

In Part I we presented the splitting of the input matrix A in two factors $A \approx L_1 L_2$ based on some LDL^T-decomposition. One key to a successful verification method is to avoid residuals with products of three matrices, here $\|A - L_1 L_2\|_2$ instead of $\|A - LDL^T\|_2$. The advantage is that each entry of the residual $A - L_1 L_2$ is one dot product for which fast and accurate algorithms are available.

The methods in Parts I and II explore the ideas in [39, 40, 42] published in the 1990's. For the time being the algorithms for LDL^T-decomposition were not reliable enough to allow for robust verification methods. Nowadays good scaling and equilibration routines are available [8, 9] making those methods attractive. That was observed by Terao and Ozaki [48] and triggered both parts of this note.

One key of our methods is the following theorem [40, Theorem 1.1]:

Theorem 1.1. *Let symmetric $A \in \mathbb{R}^{n\times n}$, $0 < \tilde{\lambda} \in \mathbb{R}$ and $\tilde{L}_1, \tilde{D}_1, \tilde{L}_2, \tilde{D}_2 \in \mathbb{R}^{n\times n}$ be given. If the inertia of $\tilde{D}_1$ and $\tilde{D}_2$ are equal, then for any matrix norm*

$$\sigma_{\min}(A) > \tilde{\lambda} - \max\{\|A - \tilde{\lambda}I - \tilde{L}_1\tilde{D}_1\tilde{L}_1^T\|, \|A + \tilde{\lambda}I - \tilde{L}_2\tilde{D}_2\tilde{L}_2^T\|\}. \tag{1.1}$$

If all eigenvalues of $\tilde{D}_1$ are positive, then

$$\sigma_{\min}(A) > \tilde{\lambda} - \|A - \tilde{\lambda}I - \tilde{L}_1\tilde{D}_1\tilde{L}_1^T\|. \tag{1.2}$$

The proof is clear from the fact that the inertia of $\tilde{L}_k\tilde{D}_k\tilde{L}_k^T$ and $\tilde{D}_k$ coincide for $k \in \{1,2\}$. We use "tilde" to indicate that approximate factorizations are used.

An application to symmetric (positive definite) A sets $\tilde{G} := \tilde{L}_1^T$ and $\tilde{D}_1 = I$, such that (1.2) implies

$$\sigma_{\min}(A) > \tilde{s} - \|A - \tilde{s}I - \tilde{G}^T\tilde{G}\| =: \varrho \tag{1.3}$$

for an approximate Cholesky decomposition $A - \tilde{s}I \approx \tilde{G}^T\tilde{G}$. This certifies a lower bound ϱ of the smallest singular value $\sigma_{\min}(A)$ based on some approximation $\tilde{s}$. If ϱ is positive it proves positive definiteness of A.

That approach for symmetric (positive definite) A was further explored in [45]. It is appealing that a priori bounds for $\|A - \tilde{s}I - \tilde{G}^T\tilde{G}\|_2$ are available at practically no cost solely based on the diagonal of A. This is based on [6], see also [11, Theorem 10.5]. In Lemma 2.10 and Corollary 2.12 in Part I of this note we improve the bound ϱ by using linear estimates on the rounding error of dot products [17, 18, 19] and a tailored application of Perron-Frobenius Theory. The new bound often improves on existing ones by a factor 100 and more, see Figure 2 in Part I.

Another application [42, 48] of Theorem 1.1 gives a lower bound on $\sigma_{\min}(A)$ of a general matrix A by using the augmented matrix $B := \begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix}$. The eigenvalues of B are $\pm\sigma_k(A)$ so that the inertia of B is known to be $(n, n, 0)$ for nonsingular A, i.e., n positive, n negative and no zero eigenvalue. Hence

$$\sigma_{\min}(A) = \sigma_{\min}(B) > \tilde{s} - \|B - \tilde{s}I - \tilde{L}\tilde{D}\tilde{L}^T\| =: \varrho \tag{1.4}$$

for an anticipated lower bound $\tilde{s}$ of $\sigma_{\min}(A) = \sigma_{\min}(B)$ is true if $\tilde{D}$ has n positive and n negative eigenvalues for an approximate LDL^T-decomposition $B - \tilde{s}I \approx \tilde{L}\tilde{D}\tilde{L}^T$. Note that $\varrho > 0$ implies that B has full rank and therefore A is nonsingular.

If $\sigma_{\min}(A) \geqslant \varrho > 0$, then for an approximate solution $\tilde{x}$ of a linear system $Ax = b$ it follows

$$\|A^{-1}b - \tilde{x}\|_\infty \leqslant \|A^{-1}(b - A\tilde{x})\|_2 \leqslant \varrho^{-1}\|b - A\tilde{x}\|_2$$

as noted in Part I. However, for ill-conditioned A that bound may be quite some overestimation. Therefore it is improved by a residual iteration as described in Section 4 of Part I. If accurate dot products are available, often close to maximally accurate entrywise bounds for the solution are computed, i.e., the left and right bounds differ by few bits. In our examples that is sometimes not the case, and to that end we present a further improvement of the accuracy of the bounds at the end of Section 2.

In Part I of this note we treat three cases separately, namely symmetric (positive definite), symmetric indefinite and general matrices. As has been explained "positive definite" is not an assumption but a property proved by the method a posteriori. In this Part II we will improve on the second and third case, both based on a factorization F_1F_2 with $\sigma_{\min}(F_1) = \sigma_{\min}(F_1) \approx \sqrt{\sigma_{\min}(A)}$. More precisely, $F_2 = SF_1^T$ for a signature matrix S, i.e., real diagonal S with entries ± 1 on the diagonal. Hence, the factors have identical sets of singular values and the inertia of F_1F_2 is equal to that of S. The methods are based on that together with estimates on the error of the factorization F_1F_2 and Theorem 1.1.

That sounds simpler than the methods presented in Part I. However, there is no clear picture. Often the methods in Part I are faster, sometimes much faster, but those in this Part II seem are a little more often successful. We elaborate on that in several numerical examples in Section 10.

As a second improvement in this Part II over Part I, the positive definite and symmetric subalgorithms will allow "nearly" symmetric input matrices. That may be useful if symmetry is broken due to rounding errors and may speed up the algorithms significantly. We show that Algorithm "verifySparseNearSym1" in Part I works correctly for unsymmetric input matrix without change.

As in Part I, our primary target is that our algorithm ends successfully, i.e., verifies non-singularity of the input matrix and computes error bounds for the solution of the linear system. Our algorithms are tuned to that goal accepting some penalty in computing time. In addition to the mathematically rigorous verification, the second focus is to compute accurate bounds for the solution.

Our notation is as in Part I. In particular we assume a set of floating-point numbers $\mathbb{F}$ with an arithmetic according to the IEEE 754 floating-point standard [14]. We use double precision (binary64) in a nearest rounding[1] with relative rounding error unit $\mathbf{u} = 2^{-53} \approx 10^{-16}$, and we use directed rounding downwards (towards $-\infty$) and upwards (towards $+\infty$). In INTLAB [41] the command `setround(-1)` switches the rounding to downwards. That means that henceforth all floating-point operations are executed in rounding downwards until the next call of `setround`. That includes in particular vector and matrix operations. Similarly, `setround(1)` switches the rounding to upwards, and `setround(0)` to nearest. For simplicity, we assume throughout this note that neither overflow nor underflow occurs because the adaptation of the following to intermediate results outside the floating-point range is rather straightforward.

We use float$(\cdot)$ to indicate the result of an expression with all operations executed in floating-point. If the order of execution is not unique, results are true for any order.

We borrow some results of Part I of this note as follows.

[1] Our results in rounding to nearest are true for any rounding of ties.

Part II	Part I	description
(1.5)	Table 1	Algorithm `NormBnd` to compute upper bound of $\|\cdot\|_2$
(1.6)	(1.11)	A, E symm. $\Rightarrow \lambda_{\min}(E) \leqslant \lambda_k(A+E) - \lambda_k(A) \leqslant \lambda_{\max}(E)$
(1.7)	(2.10)	norm of residual using a priori bounds
(1.8)	(2.13)	norm of residual using extended precision
(1.9)	(3.1)	equilibration of spd matrix
(1.10)	(3.2)	equilibration of a symmetric matrix
(1.11)	(3.3)	equilibration of a general matrix
(1.12)	(3.5)	$[\mathtt{L},\mathtt{D},\mathtt{p}] = \mathtt{ldl}(\mathtt{A},\mathtt{thresh},'\mathtt{vector}');$
(1.13)	(3.7)	remedy for LDL^T-decomposition
(1.14)	(3.9)	approximation of smallest singular value
(1.15)	(6.2)	a priori bound for Cholesky decomposition
(1.16)	(7.1)	decomposition of D

The left-most column is the reference used in this Part II of our note.

We begin with an alternative method to compute accurate approximations and inclusions of residuals. That is paramount to our methods. Using this we show how to improve even more the accuracy of our inclusions. This leads to inclusions which are almost always maximally accurate for all entries.

First we show how to adapt Algorithm "verifySparseSPD1" in Part I to work correctly for nearly symmetric matrices. Next we discuss how to compute the inertia of the block matrix D of an LDL^T-decomposition. Then we explain our alternative methods for symmetric and for general input matrix, the former working for nearly symmetric matrices as well. Based on that we show how to compute inclusions of the solution of a least squares problem and of an underdetermined system of equations. We present our second Algorithm `verifySparselss2` to compute rigorous error bounds for a linear system with square or rectangular, real or complex sparse matrix and multiple right hand sides. Based on the solution of sparse square linear systems, we present Algorithm `verifySparseNlss` for computing error bounds for the solution of a system of nonlinear equations.

Numerical examples for square, over- and underdetermined systems, for randomly generated matrices and nonlinear systems are shown. We close this note with concluding remarks and an open problem.

2. Approximation and estimation of matrix residuals. A key point to our methods are upper bounds on the spectral norm of some residual $AB - C$ for compatible matrices A, B, C. Those are based on accurate dot products, with or without error bound. To that end any of the many accurate dot product algorithms is suitable. There are Matlab implementations, however, they suffer from interpretation overhead, in particular for sparse data. We used Advanpix [12] in Part I of this note, a multiple-precision Matlab package emulating a large number of Matlab's algorithms. The number `d` of decimal digits of precision can be freely specified by `mp.Digits(d)`.

However, according to [12] the precision in use is `d` decimal digits plus some guard digits, but there is no specific information about the accuracy of a result. Moreover, for a general specification `mp.Digits(d)` the package does not respect the rounding mode.

To that end there is one exception, namely `mp.Digits(34)`. That is a particularly fast implementation of extended precision arithmetic with relative rounding error unit 2^{-113} and satisfying the IEEE 754 standard [14]. That implementation respects the specified rounding mode, for the arithmetic operations as well as for the type cast `double(·)` from `mp` to double precision. The conversion from `double` to `mp` is error-free. Thus the code

```
setround(-1); E = abs(double(mp(A) * B - C));
setround(+1); E = max(E, abs(double(mp(A) * B - C)));
```

computes a floating-point matrix E such that $|AB - C| \leqslant \mathtt{E}$ is true for the real matrix $AB-C$ using entrywise absolute value and comparison, see Lemma 2.5 in Part I of this note. We first convert to double precision to minimize the number of mp-operations.

The main reason to use the toolbox Advanpix [12] in Part I was to show a fair comparison with [48] because it was also used in there. However, in this Part II we want to use higher precision to achieve even more accurate bounds. That is not possible when using [12] because except extended precision as by `mp.Digits(34)` no clear definition of the arithmetic is available.

An alternative to Advanpix [12] is Matlab's multiple precision package `vpa`. However, that is very slow, see the timing in Table 1.

Recently we work [20] on a new algorithm improving on [34]. The mathematical basis for the accurate computation of a dot product $a^T b$ of $a, b \in \mathbb{F}^n$ is as follows. In

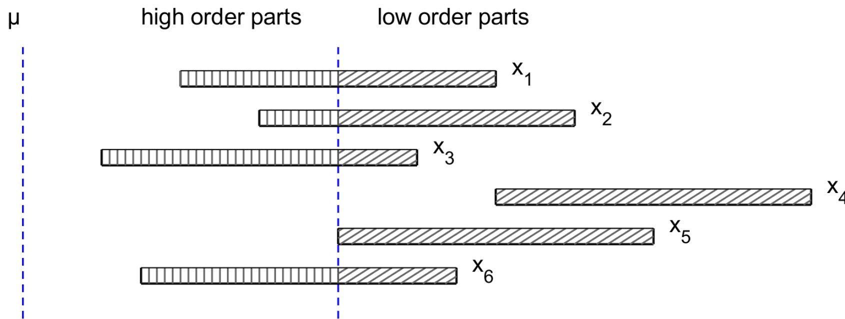

Fig. 1. *The Zielke/Drygalla scheme to extract high and low order parts*

[49] an *absolute splitting* of vectors was introduced, following the scheme in Figure 1. The vectors a, b are split into high and low order parts $a = p + q,\ b = r + s$ in such a way that the dot product $p^T r$ of the high order parts is computed without error in floating-point. The constant μ determines the splitting and is chosen such that all products $p_i r_i$ and their sum reside in the range of digits of one floating-point number in the given format. That method was analysed in [46] and is also used for reproducible results [2].

One specific advantage of the absolute splitting is the applicability to matrix products. The recursive application leads to the following *Ozaki scheme* for the matrix product AB of two floating-point matrices. It was originally published in [32, 36, 37]

with improvements in [33, 34]. In the first step A is split into $k+1$ parts

$$A = A^{(1)} + A^{(2)} + \ldots + A^{(k)} + \underline{A}^{(k)} \tag{2.1}$$

where each part $A^{(i)}$ holds a limited range of mantissa digits and $\underline{A}^{(k)}$ is the least significant part containing the remainder. A similar splitting is applied to B. Such a splitting is well known and often used, for example, in multiple precision arithmetic. The point here is that the ranges for the mantissa digits in $A^{(i)}$ and $B^{(j)}$ are chosen in such a way that all the individual matrix products $A^{(i)}B^{(j)}$ are computed error-free independent of the order of evaluation.[2] The number k is usually small, typically $k = 1$ or $k = 2$. Ozaki et al. [36, 35, 34] exploited this by computing AB as the unevaluated sum of $\frac{(k+1)(k+2)}{2}$ individual matrix products

$$AB = \sum_{i+j \leqslant k+1} A^{(i)}B^{(j)} + \underbrace{\sum_{i=1}^{k} A^{(i)}\underline{B}^{(k+1-i)} + \underline{A}^{(k)}B}_{\text{remainder terms}}. \tag{2.2}$$

where the sum of these is realized via an accurate summation algorithm, for instance [3, 27, 7, 30, 46]. Then the overall error is determined by the rounding errors in the computation of the $k+1$ remainder terms which are least significant. By using the particular splitting approach proposed in [36], one can expect the error to be roughly of the size $(2n\mathbf{u})^{k/2+1}|A||B|$, where $\mathbf{u}$ denotes the relative rounding error unit. Hence, with increasing k there is a significant increase in the precision.

A major advantage of Ozaki's scheme over other approaches for computing accurate matrix-matrix products is the efficient use of highly optimized level-3 BLAS routines. For algorithms based on vector transformations, such as `Dot2` [30], reaching peak performance is more difficult and requires to perform optimizations by hand. A second benefit of Ozaki's scheme is the relatively low computational complexity for small k. The biggest drawback is that the computational complexity and the required memory increase quadratically with k.

In [20] we discuss various improvements to the original Ozaki scheme resulting in Algorithms `prodK` for full and `spProdK` for sparse matrices. An important advantage is the determination of nearly optimal splitting points as in Figure 1. When compared to the original splitting by Ozaki's methods, this yields roughly an additional precision of k digits. Moreover, instead of the infinity norm of the respective column or row vectors, we use the Euclidean norm to determine suitable splitting parameters. This often gives another factor two in precision.

The Algorithms `prodK` and `spProdK` are pure Matlab code in an ingenious implementation by Marko Lange [20]. Despite the interpretation overhead they are very fast. Timing ratios of `vpa`, `mp` and `prodK` for full matrices are shown in Table 1, where the column $t_{\texttt{prodK}}$ is timing in seconds. We used the default precisions 32 digits for `vpa`, extended precision `mp.Digits(34)` and $k = 2$ for `prodK` and `spProdK` corresponding to about $(k/2+1)$-fold, i.e., double precision.

For all data the accuracy of the results is similar. For full matrices `vpa` is much slower than `mp`, and for little larger dimension `mp` is much slower than `prodK`.

For sparse matrices much effort is necessary to ensure an efficient memory management. To that end Marko Lange provided a special implementation `spProdK` which

[2]This is true for standard matrix multiplication but requires further modifications to work with asymptotically faster approaches such as the Strassen or the Coppersmith–Winograd algorithm.

Table 1
Timing ratio for full matrix multiplication $A, B \in \mathbb{F}^{n\times n}$

	real data			complex data		
n	$t_{\texttt{vpa}}/t_{\texttt{mp}}$	$t_{\texttt{mp}}/t_{\texttt{prodK}}$	$t_{\texttt{prodK}}$	$t_{\texttt{vpa}}/t_{\texttt{mp}}$	$t_{\texttt{mp}}/t_{\texttt{prodK}}$	$t_{\texttt{prodK}}$
100	464	0.9	0.02	236	3.1	0.03
300	326	18.0	0.03	220	10.8	0.05
1000	306	30.1	0.21	206	62.6	0.48

is, as `prodK`, included in INTLAB [41]. Timing of `vpa`, `mp` and `spProdK` for sparse matrices is shown in Table 2 for matrices with some 100 nonzero entries per row.

Table 2
Timing ratio for sparse matrix multiplication $A, B \in \mathbb{F}^{n\times n}$

	real data			complex data		
n	$t_{\texttt{vpa}}/t_{\texttt{mp}}$	$t_{\texttt{mp}}/t_{\texttt{spProdK}}$	$t_{\texttt{spProdK}}$	$t_{\texttt{vpa}}/t_{\texttt{mp}}$	$t_{\texttt{mp}}/t_{\texttt{spProdK}}$	$t_{\texttt{spProdK}}$
1000	1325	0.3	0.06	1010	0.3	0.10
3000	2437	1.0	0.08	3466	0.6	0.09
10000	-	0.6	0.25	-	0.8	0.22
30000	-	1.0	0.53	-	1.0	0.49

The used precisions are as before, and again all results are of similar accuracy. For dimension 10,000 and larger `vpa` stopped with memory problems. However, `vpa` would only be an option to compute accurate approximations, but it is not suitable for verified inclusions because it does not allow the computation of error bounds. The same is true for Advanpix except for extended precision using `mp.Digits(34)`.

One may ask why the highly optimized C-code of the Advanpix mp-toolbox does not outperform `prodK` and `spProdK` which are both written in pure Matlab code, thus severely suffering from interpretation overhead. We hastily want to stress that the comparison is not fair as `prodK` is tuned to matrix residuals while the mp-toolbox provides a general arbitrary-precision arithmetic. Besides that, according to the author Pavel Holoborodko [13], the main reason that Advanpix is slower than `prodK` is the significant overhead that MATLAB itself imposes for custom data types. This involves extra data copying and a time-consuming process of creating new mp-objects by calling the MATLAB interpreter from within native code.

In Part I of this note we exclusively used the mp-toolbox Advanpix [12] for accurate residual approximations and inclusions to ensure a fair comparison with [48], throughout this Part II we use exclusively `prodK` and `spProdK`.

For `prodK` and similarly for `spProdK` typical calls are

$$\begin{array}{ll} \texttt{res} = \texttt{prodK}(\texttt{L},\texttt{U},-1,\texttt{A},\texttt{k}); & LU - A \approx res \\ [\texttt{res},\texttt{err}] = \texttt{prodK}(\texttt{L},\texttt{U},-1,\texttt{A},\texttt{k}); & LU - A \in res \pm err \\ [\texttt{res},\texttt{err}] = \texttt{prodK}(\texttt{A},\texttt{x},\texttt{A},\texttt{y},-1,\texttt{b},\texttt{k}); & Ax + Ay - b \in res \pm err \\ \texttt{res} = \texttt{prodK}(\texttt{A},\texttt{x},-1,\texttt{b},\texttt{k},'\texttt{OutputTerms}',2); & Ax - b \approx res_{\{1\}} + res_{\{2\}} \end{array} \tag{2.3}$$

For the pairs of input parameters $p_1, q_1, p_2, q_2, \ldots$ the value $\sum p_i q_i$ will be computed, where each of the first parameters may be a scalar. For one output parameter *res*

the result will be approximated in about $(k/2+1)$-fold precision. For two output parameters, $res \pm err$ is a correct inclusion, also computed in $(k/2+1)$-fold precision. Finally, `'OutputTerms',m` specifies that the result is stored in a cell array with m members. That corresponds to an unevaluated sum of m addends.

In (1.6) in Part I we introduced a notation for the approximation and inclusion of a residual $Ax-b$ with sample Matlab/INTLAB code in (1.7). Here we extend the notation allowing evaluation in higher precision. The subindices $_{k,1}$ indicate that the expression is evaluated in k-fold precision and rounded into working precision. The last parameter k in the calls of `prodK` and `spProdK` implies a result "as if" evaluated in $k/2+1$-fold precision. Therefore using `spProdK` sample Matlab/INTLAB code is

(2.4)

$[\![expr]\!]_{k,1}$
```
res = spProdK(A,x,-1,b,2*(k-1));
```
$\langle\!\langle expr\rangle\!\rangle_{k,1}$
```
[res,err] = spProdK(A,x,-1,b,2*(k-1));
res = midrad(res,err);
```

For compatible matrices A, B, C we borrow the function `NormBnd` in Table 1 and the code in (2.11) of Lemma 2.5 in Part I to bound $\|AB-C\|_2$:

(2.5)
```
setround(-1); E = abs(A*B-C);
setround(+1); E = max(E,abs(A*B-C));
beta = NormBnd(E,symm);
```

The second parameter `symm` in the function `NormBnd` is chosen to be `true` if $AB-C$ is symmetric/Hermitian. A bound $\|AB-C\|_2 \leqslant \gamma$ computed in higher precision as in (2.12) and (2.13) of Lemma 2.5 in Part I becomes much simpler and is now replaced by

(2.6)
```
[res,err] = spProdK(A,B,-1,C,k);
setround(1)
gamma = NormBnd(abs(res)+err,symm);
```

Then $\|AB-C\|_2 \leqslant \gamma$ because the sum `abs(res)+err` in the last statement is computed in rounding upwards and $\|M\|_2$ is monotone for nonnegative M.

As explained above we work with a factorization $A \approx L_1L_2$ so that the entries of the residual $L_1L_2 - A$ consist of dot products. For ill-conditioned input matrix it might be necessary to compute an upper bound α of the spectral norm of a residual $LDL^T - A$. Here extra care is necessary because now the product of three matrices is involved. The following code in Table 3 computes an upper bound α of $\|LDL^T - A\|_2$.

The proof of correctness is as follows. The first line yields matrices C_1, C_2, E_1 with

$$|C_1 + C_2 - DL^T| \leqslant E_1$$

with entrywise absolute value and comparison. The matrix pair (C_1, C_2) approximates DL^T as an unevaluated sum which corresponds to quadruple precision. The matrices C, E_2 in the next line satisfy

$$|LC_1 + LC_2 - A - C| \leqslant E_2\ .$$

The next line uses Algorithm `NormBnd` from Table 1 in Part I of this note to compute an upper bound of the 2-norm of the absolute of a matrix. Thus rounding upwards

```
function p = residualBoundLDLT(A,L,D)
   [C1,C2,E1] = spProdK(D,L',2);
   [C,E2] = spProdK(L,C1,L,C2,-1,A,2);
   setround(1)
   alpha1 = NormBnd(abs(C)+E2,false);
   alpha = NormBnd(L,false)*NormBnd(E1,false) + alpha1;
end  % function residualBoundLDLT
```

Table 3
Computation of an upper bound α of $\|LDL^T - A\|_2$

implies $\||C| + E_2\|_2 \leqslant \alpha_1$ and finally

$$\begin{aligned} \|LDL^T - A\|_2 &= \|L(DL^T - C_1 - C_2) + C + L(C_1 + C_2) - A - C\|_2 \\ &\leqslant \|L\|_2 \|E_1\|_2 + \|\ |C| + |L(C_1 + C_2) - A - C|\ \|_2 \\ &\leqslant \|L\|_2 \|E_1\|_2 + \|\ |C| + E_2\ \|_2 \\ &\leqslant \|L\|_2 \|E_1\|_2 + \alpha_1 \end{aligned}$$

is true because first summand in the final line of Algorithm `residualBoundLDLT` ensures $\|L\|_2 \|E_1\|_2 \leqslant$ `NormBnd(L,false)` $*$ `NormBnd(E1,false)` and because the sum in the last line is computed in rounding upwards. The extra parameter "false" in `NormBnd` indicates that the input matrix is not necessarily symmetric. We choose not to calculate $\|LE_1\|_2$ but to bound it by $\|L\|_2 \|E_1\|_2$ to save a matrix multiplication. Since E_1 is very small this does no harm. Note that due to rounding errors E_2 need not be symmetric.

Accurate bounds for matrix residuals are mandatory to compute accurate error bounds for the solution of a linear system. In Section 4 in Part I of this note we introduced in Table 2 the function `ErrorBound`. It stores an approximate solution of $A^{-1}b$ in two parts $\tilde{x}, \tilde{y}$ such that the unevaluated sum $\tilde{x} + \tilde{y}$ produces a small residual $\varrho = \|A\tilde{x} + A\tilde{y} - b\|_2$. The computation of ϱ is very ill-conditioned and requires at least double the working precision. To that end `mp.Digits(34)` is sufficient to improve an approximation and the inclusion.

In order to obtain almost always error bounds close to maximal accuracy for all entries of the solution, we follow [38] and store an approximation as an unevaluated sum of three parts $\tilde{x}, \tilde{y}, \tilde{z}$. Then the residual $\varrho = \|A\tilde{x} + A\tilde{y} + A\tilde{z} - b\|_2$ is even more ill-conditioned and using twice the working precision is not sufficient. More precisely, when using `mp.Digits(34)` there would be no improvement whether using two or three parts for the approximation.

A higher precision can be specified in `mp`, however, there is not enough information about the arithmetic in use to compute valid error bounds. In contrast, higher precision can be specified in `prodK` and `spProdK` to compute an accurate approximation and with the possibility to obtain verified error bounds. For example, an inclusion of $\|A\tilde{x} + A\tilde{y} + A\tilde{z} - b\|_2$ is computed by

$$[\mathtt{c},\mathtt{e}] = \mathtt{spProdK}(\mathtt{A},\mathtt{xs},\mathtt{A},\mathtt{ys},\mathtt{A},\mathtt{zs},-1,\mathtt{b},\mathtt{k})$$

implying that

$$|A\tilde{x} + A\tilde{y} + A\tilde{z} - b - c| \leqslant e$$

is satisfied for all entries. The parameter k specifies that about $(k/2+1)$-fold precision is used. For an approximation in three parts $k = 4$ corresponding to 3-fold precision

is suitable. This leads to an improved and very accurate version `ErrorBound3` of Algorithm `ErrorBound` in Table 2 in Part I of this note. Algorithm `ErrorBound3` is given in Table 4. If necessary, Step 6 may be repeated two or three times. The implementation of $[\![\cdot]\!]_{k,1}$ follows (2.4).

1	$[\tilde{x},\delta] = \text{ErrorBound3}(A,b,s,\text{“}solve\text{“})$	
2	$\tilde{x} = \text{solve}(A,b)$	% $A^{-1}b \approx \tilde{x}$
3	$\tilde{y} = \text{solve}(A,[\![b - A\tilde{x}]\!]_{2,1})$	% $A^{-1}b \approx \tilde{x}+\tilde{y}$
4	$[\tilde{x},\tilde{y}] = \text{TwoSum}(\tilde{x},\tilde{y})$	
5	$\tilde{z} = \text{solve}(A,[\![b - A\tilde{x} - A\tilde{y}]\!]_{2,1})$	% $A^{-1}b \approx \tilde{x}+\tilde{y}+\tilde{z}$
6	$\tilde{z} = \text{solve}(A,[\![b - A\tilde{x} - A\tilde{y} - A\tilde{z}]\!]_{3,1})$	% $A^{-1}b \approx \tilde{x}+\tilde{y}+\tilde{z}$
7	$\text{setround}(-1);\ c_1 = \tilde{y}+\tilde{z};\ \varrho = \text{abs}\,([\![A\tilde{x}+A\tilde{y}+A\tilde{z}-b]\!]_{3,1})$	
8	$\text{setround}(+1);\ c_2 = \tilde{y}+\tilde{z};\ \varrho = \max\,(\,\varrho\,,\,\text{abs}\,([\![A\tilde{x}+A\tilde{y}+A\tilde{z}-b]\!]_{3,1})\,)$	
9	$\delta = \max(\lvert c_1\rvert,\lvert c_2\rvert) + \texttt{vecnorm}(\varrho)/s$	

Table 4
Improved residual iteration and inclusion of the solution $A^{-1}b$

The proof of correctness is as for `ErrorBound` in Part I of this note because only the approximation was changed from $\tilde{x}+\tilde{y}$ to three parts $\tilde{x}+\tilde{y}+\tilde{z}$. Of course it is possible to split the approximation into an unevaluated sum of even more parts, where increasing the parameter k in `prodK` or `spProdK` would compute the residuals with sufficient accuracy. However, we refrained from doing this because we rarely encountered entries with not maximally accurate inclusion.

3. Nearly symmetric/Hermitian matrices. The following remains true, *mutatis mutandis*, when replacing “symmetric” by “Hermitian”.

Suppose we want to solve a linear system with a nearly symmetric positive definite but unsymmetric input matrix A, i.e., there is an s.p.d. matrix $\tilde{A}$ with $\|\tilde{A}-A\|$ small compared to $\|A\|$. As a consequence, Algorithm “verifySparseSPD1” introduced in Part I of this note is not applicable.

A simple way to fix that is to solve a linear system with interval matrix $\mathbf{A}$ with symmetric midpoint and $A \in \mathbf{A}$. Then our symmetric solver “verifySparseSPD1” can be applied. However, the solution of all linear systems $\widehat{A}x = b$ with $\widehat{A} \in \mathbf{A}$ are included, among them the given matrix A.

Following we show a simple way to treat nearly symmetric matrices, i.e., when $\|A^T - A\|$ is small. We define A^{sym} by mirroring the lower half to the upper, i.e.,

$$A^{\text{sym}}_{ij} := \begin{cases} A_{ij} & \text{if } i \geqslant j \\ A_{ji} & \text{otherwise} \end{cases}, \tag{3.1}$$

in Matlab notation `tril(A)+tril(A,-1)'`. Then A^{sym} is symmetric, and $A^T = A$ if, and only if, $A^{\text{sym}} = A$. Note that $A^{\text{sym}} - A = \text{triu}(A' - A, 1)$ may be considered as a distance to symmetry. Next we introduce in Table 5 an update of Algorithm “verifySparseSPD1” in Part I of this note computing verified bounds for the solution of a linear system with unsymmetric A. As has been announced earlier, we show later that Algorithm “verifySparseNearSym1” introduced in Part I and intended for symmetric input matrix A works correctly for unsymmetric A as well.

```
1   function [x, δ] = verifySparseSPD2(A, b)
2       If any A_kk A_ℓℓ ⩽ 0, [x, δ] = verifySparseNearSym2(A,b), return
3       If A_11 < 0, A := −A; b := −b;    % A possibly negative definite
4       Equilibrate A by (1.9),  sym := isequal(A', A)
5       If sym, A^sym = A  else  A^sym = tril(A) + tril(A, −1)^T;
6       Compute Cholesky factorization R̃^T R̃ ≈ A^sym
7       If failed, [x, δ] = verifySparseNearSym2(A,b), return
8       Compute s̃(R̃) by (1.14) and set s := 0.9s̃
9       If sym, β := 0, else compute β with ‖triu(A^T − A, 1)‖_2 ⩽ β  by (1.5)
10      If s − β < 10^−16 ‖A‖_∞,  [x, δ] = verifySparseNearSym2(A,b), return
11      Rounding downwards,  As = A^sym − sI
12      setround(0); [Rs, FLAG, p] = chol(As);
13      If succeeded, go to Step 17,  else s := s̃/5,  if s ⩽ β, return
14      Rounding downwards,  As = A^sym − sI
15      setround(0); [Rs, FLAG, p] = chol(As);
16      If failed, [x, δ] = verifySparseNearSym2(A,b), return
17      if sym, ŝ := s, else ŝ := s − β in rounding downwards
18      Compute upper bound α := r.h.s.(1.15) with ‖Rs^T Rs − As‖_2 ⩽ α
19      If α ⩾ ŝ, compute α with ‖Rs^T Rs − As‖_2 ⩽ α using (2.5)
20      If α ⩾ ŝ, compute α with ‖Rs^T Rs − As‖_2 ⩽ α using (1.8)
21      If α ⩾ ŝ, verification failed, return
22      [x, δ] = ErrorBound(A, b, ŝ − α, "solve") using R̃ for solve
```

Table 5

Verified error bounds for $A^{-1}b$ for nearly s.p.d. sparse input matrix A

Basically the difference between Algorithms "verifySparseSPD1" and "verifySparseSPD2" is that Steps $4, 5, 9, 10$ and 17 are added/changed, s is replaced by $\widehat{s}$ in Steps 19 to 22, and A is replaced by A^{sym} in Steps $6, 11$ and 14.

First, suppose "verifySparseSPD2" is called with symmetric input matrix A. Then $A^{\text{sym}} = A$ and $\beta = 0$, $\widehat{s} = s$ from Step 17. Thus the results of Algorithms "verifySparseSPD1" and "verifySparseSPD2" are the same, implying $A^{-1}b \in \tilde{x} \pm \delta$.

Next suppose the input matrix A is not symmetric. Then Steps $6, 11, 14$ are executed with A^{sym} replacing the unsymmetric input matrix A. Matlab's implementation of the Cholesky decomposition in Steps $6, 12$ and 15 uses the lower triangle of A to compute the factorization. Since $\text{tril}(A^{\text{sym}}) = \text{tril}(A)$, the value of s is the same as if "verifySparseSPD1" were applied to A^{sym}. Hence $\sigma_{\min}(A^{\text{sym}}) \geqslant s - \|Rs^T Rs - As\|_2$ and using $A^{\text{sym}} - A = \text{triu}(A^T - A, 1)$ gives

$$\sigma_{\min}(A) \geqslant \sigma_{\min}(A^{\text{sym}}) - \|A^{\text{sym}} - A\|_2 \geqslant s - \|Rs^T Rs - As\|_2 - \beta.$$

Step 18 computes an upper bound α of $\|Rs^T Rs - As\|_2$. If the bound is not small enough, it is improved in Step 19 and eventually again in Step 20. In any case,

$\|Rs^TRs - As\|_2 \leqslant \alpha$ in Step 21. Hence

$$\sigma_{\min}(A) \geqslant s - \alpha - \beta \geqslant \widehat{s} - \alpha$$

since $\widehat{s}$ is computed in rounding downwards in Step 17. As a result, if running to completion, Algorithm "verifySparseSPD2" verifies nonsingularity of A and proves $A^{-1}b \in \tilde{x} \pm \delta$ for symmetric as well as for general input matrix A.

The algorithm is intended to be used for matrices with $A^T \approx A$, i.e., for small β. Mathematically, there is no limit on β: If "verifySparseSPD2" runs to completion, results are verified to be correct. However, the larger β, the larger must be the smallest singular value of A^{sym}.

Another reason why β should not be too large is that the final error bound in Step 22 is computed based on the factorization of A^{sym} in Step 6 and not of A. The rate of convergence has been analysed in [45] showing how it becomes weaker with increasing β, i.e., deviation from symmetry.

As an example of an implementation detail combining Steps 9 and 17 of "verifySparseSPD2" we show that the lower bound $\widehat{s}$ of $s - \|A^{\text{sym}} - A\|_2$ is correctly computed by

```
setround(1);  shat = -(NormBnd(abs(triu(A' - A, 1)) * (1 + eps), false) - s);
```

Parsing the expression on the right leads to the following statements in the left column:

$$\begin{aligned}
\mathtt{P} = \mathtt{A'} - \mathtt{A} \quad &\Rightarrow \quad |(A'-A) - \mathtt{P}| = |(A'-A) - \text{float}(A'-A)| \leqslant 2\mathbf{u}|\mathtt{P}| \\
\mathtt{Q} := \mathtt{triu}(\mathtt{P},1) \quad &\Rightarrow \quad |\text{triu}(A'-A,1) - \mathtt{Q}| \leqslant 2\mathbf{u}|\mathtt{Q}| \\
\mathtt{R} = \mathtt{abs}(\mathtt{Q}) \quad &\Rightarrow \quad |\,|\text{triu}(A'-A,1)\,| - \mathtt{R}\,| \leqslant 2\mathbf{u}|\mathtt{R}| \\
\mathtt{S} = \mathtt{R} * (1 + \mathtt{eps}) \quad &\Rightarrow \quad |\text{triu}(A'-A,1)| \leqslant (1+2\mathbf{u})|\mathtt{R}| \leqslant \mathtt{S} \\
\mathtt{t} = \mathtt{NormBnd}(\mathtt{S},\mathtt{false}) \quad &\Rightarrow \quad \|\text{triu}(A'-A,1)\|_2 \leqslant \|\ |\text{triu}(A'-A,1)|\ \|_2 \leqslant \|\mathtt{S}\|_2 \leqslant \mathtt{t} \\
\mathtt{u} = \mathtt{t} - \mathtt{s} \quad &\Rightarrow \quad \|\text{triu}(A'-A,1)\|_2 - \mathtt{s} \leqslant \mathtt{t} - \mathtt{s} \leqslant \mathtt{u} \\
\mathtt{shat} = -\mathtt{u} \quad &\Rightarrow \quad \mathtt{shat} = -\mathtt{u} \leqslant \mathtt{s} - \|\text{triu}(A'-A,1)\|_2
\end{aligned}$$

We add arguments why the conclusions in each line are valid. Note that Matlab defines `eps` to be $2\mathbf{u}$. We use step by step the error bound for subtraction in rounding upwards, $A^{\text{sym}} - A = \text{triu}(A'-A,1)$, `abs` is error-free, the previous step and the floating-point error bound in rounding upwards, that `NormBnd` computes an upper bound of the 2-norm, rounding upwards and finally negation is error-free.

Next we show that Algorithm "verifySparseNearSym1" in Part I of this note works for unsymmetric matrices as well, i.e., if the algorithm runs to completion, then A is proved to be nonsingular and $A^{-1}b \in \tilde{x} \pm \delta$ regardless whether A is symmetric or not.

Consider the call `verifySparseNearSym1(A,b)` for unsymmetric A. Except for the last step calculating the final inclusion, the input matrix A occurs only[3] in Steps 3 and 10. As Matlab's implementation of the LDL^T-decomposition in Step 3 uses the lower triangle of A to compute the factorization, the computed quantities L, D are the same whether `ldl` is factoring A^{sym} or A. As a consequence, the set of singular values of L_1 and L_2 in Step 6 still coincide. Therefore

$$\sigma_{\min}(A^{\text{sym}}) \approx \sigma_{\min}(L_1L_2) \geqslant \sigma_{\min}(L_1)\sigma_{\min}(L_2) = \sigma_{\min}(L_1)^2 = \sigma_{\min}(L_1L_1^T) \tag{3.2}$$

[3]In Step 9 it does no harm whether using A or A^{sym}.

similar to (8.1) in Part I of this note. All following quantities except α are the same whether Algorithm "verifySparseNearSym1" is applied to A^{sym} or A. Therefore

$$\begin{aligned} \sigma_{\min}(A) &\geqslant \sigma_{\min}(L_1L_2) - \|A - L_1L_2\|_2 \geqslant \sigma_{\min}(L_1L_1^T) - \|A - L_1L_2\|_2 \\ &\geqslant \sigma_{\min}(M) - \|L_1L_1^T - M\|_2 - \|A - L_1L_2\|_2 \geqslant \sigma_{\min}(M) - \beta - \alpha \ , \end{aligned} \tag{3.3}$$

The first inequality in the first line is always true regardless of L_1 or L_2, and the second follows by (3.2). The first inequality in the second line is always true as well, and the definition of α and β verify the second one.

Suppose Algorithm "verifySparseNearSym1" ran to completion. Then $\|\widehat{M} - \tilde{R}^T\tilde{R}\| \leqslant \gamma$. As noted before[4], the matrices M and $\widehat{M}$ are symmetric. That allows to apply Theorem 6.1 in Part I by replacing A and B by M and $\widehat{M}$, respectively. Then $\Delta B = \tilde{R}^T\tilde{R} - \widehat{M}$ and (6.1) in Part I yields

$$\sigma_{\min}(M) \geqslant s - \|\Delta B\|_2 \geqslant s - \gamma \ .$$

Inserting this into (3.3) proves

$$\sigma_{\min}(A) \geqslant s - \alpha - \beta - \gamma$$

for the original, not necessarily symmetric matrix A.

4. Inertia of a 2×2 matrix. For a decomposition $A = LDL^T$ of real A we need the inertia of the block diagonal matrix D. Thus we need the inertia of $M := \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ for $a, b, c \in \mathbb{F}$. For $\lambda_1, \lambda_2 \in \mathbb{R}$ denoting the eigenvalues of M, we have $\lambda_1 + \lambda_2 = \text{trace}(M) = a + c$ and $\lambda_1\lambda_2 = \det(M) = ac - b^2$. The following is true for singular M, however, if successful then nonsingularity of D will be proved a posteriori by our verification algorithm.

If $\det(M) < 0$, then the inertia $i(M) = (i_+, i_-, i_0)$, the number of positive, negative, and zero eigenvalues, is $i(M) = (1, 1, 0)$. If $\det(M) > 0$, then $i(M) = (2, 0, 0)$ if $\text{trace}(M) > 0$ and $i(M) = (0, 2, 0)$ otherwise.

We suppose a floating-point computation in some nearest rounding barring over- and underflow. A nearest rounding is defined by a rounding function $\text{fl} : \mathbb{R} \to \mathbb{F}$ with the property that for $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$ the floating-point result $\text{fl}(a \circ b)$ satisfies

$$|\text{fl}(a \circ b) - a \circ b| = \min\{|f - a \circ b| : f \in \mathbb{F}\} \ .$$

Different nearest roundings are discriminated by the rounding of the tie: If the real result $a \circ b$ is not the midpoint between two adjacent floating-point numbers, then the nearest result is uniquely determined, otherwise it is one of the two neighbours.

Any nearest rounding respects ordering, i.e.,

$$x, y \in \mathbb{R} : \quad \text{fl}(x) < \text{fl}(y) \Rightarrow x < y \qquad \text{and} \qquad x < y \Rightarrow \text{fl}(x) \leqslant \text{fl}(y) \ . \tag{4.1}$$

Since zero is a floating-point number, it follows

$$a, c \in \mathbb{F} : \quad \text{fl}(a + c) < 0 \ \Leftrightarrow \ a + c < 0 \ . \tag{4.2}$$

Here $\Rightarrow$ is clear because $\text{fl}(0) = 0$, and for $\Leftarrow$ note that $\text{fl}(a + c) = 0$ is only possible if $a + c$ is below the smallest denormalized floating-point number. However, in that case $\text{fl}(a + c) = a + c$, cf. [26, Theorem 4.2].

[4] See the remark before (8.2) in Part I about the symmetry of $\text{float}(\tilde{R}^T\tilde{R})$.

```
function p = NumPosEV(a,b,c)
  setround(0)
  d = a*c - b*b;
  if d==0         % determine sign of determinant
    [p,e] = TwoProduct(a,c);   % p+e = ac
    [q,f] = TwoProduct(b,b);   % q+f = b^2
    d = e - f;                 % using p=q
  end
  if d<0          % one positive, one negative eigenvalue
    p = 1;
  elseif d > 0    % eigenvalues have same sign
    if a > -c     % two positive eigenvalues
      p = 2;
    else          % two negative eigenvalues
      p = 0;
    end
  else            % matrix singular
    p = ( a+c>0 );
  end
end  % function NumPosEV
```

Table 6
Computing the number p *of positive eigenvalues of* $M := [\mathtt{a\ b}; \mathtt{b\ c}]$

It remains the problem to compute the sign of $\det(M) = ac - b^2$ in floating-point. Let $p := \mathrm{fl}(ac)$ and $q := \mathrm{fl}(b^2)$. Then (4.1) implies

$$p - q < 0 \;\Rightarrow\; ac < b^2 \;\Leftrightarrow\; \det(M) < 0 \tag{4.3}$$

and similarly for $p - q > 0$. It remains the case $p = q$. Since p, q are computed in floating-point, still $\det(M) \neq 0$ but $\mathtt{d} = 0$ is possible, and the sign has to be decided. In that rare case we use the error-free transformation `TwoProduct` [15, 46, 26]. For $a, b \in \mathbb{F}$ the call `[x,y] = TwoProduct(a,b)` produces $x, y \in \mathbb{F}$ with $x = \mathrm{fl}(ab)$ and $x + y = ab$. Let

$$[\mathtt{p}, \mathtt{e}] = \mathtt{TwoProduct}(\mathtt{a}, \mathtt{c}) \quad \text{and} \quad [\mathtt{q}, \mathtt{f}] = \mathtt{TwoProduct}(\mathtt{b}, \mathtt{b}) \; .$$

Then

$$p = q \quad \Rightarrow \quad \det(M) = ac - b^2 = e - f$$

and the sign of the determinant can be determined as for the trace.

The Algorithm `NumPosEV` in Table 6 is executable Matlab/INTLAB code and computes the number of positive eigenvalues of a symmetric matrix $M := [\mathtt{a\ b}; \mathtt{b\ c}]$. The first line sets the rounding mode to nearest [41]. From what we derived before the correctness is clear for $\det(M) \neq 0$. If $\det(M) = 0$ the eigenvalues are $\lambda_1 = 0$ and λ_2. Thus $\mathrm{trace}(M) = a + c = \lambda_2$ and proves correctness of the algorithm.

5. Symmetric matrices. We show in Table 7 a general outline of our modified subalgorithm "verifySparseNearSym2" to compute verified bounds for the solution of a sparse linear system with symmetric or nearly symmetric matrix.

This method explores on Theorem 1.1 published in [40]; the difference to the method in Part I of this note will be explained at the end of this section. The original

1 function $[x, \delta]$ = verifySparseNearSym2(A,b)
2 Equilibrate A by (1.10), sym $:=$ `isequal(A',A)`
3 If sym, $A^{\text{sym}} = A$ else $A^{\text{sym}} = \text{tril}(A) + \text{tril}(A,-1)^T$;
4 Compute $LDL^T(A^{\text{sym}})$ by (1.12)
5 If D is singular, $[x, \delta]$ = verifySparseGen2(A,b), return
6 Compute $\tilde{s}(L, D)$ by (1.14) and set $s := 0.9\tilde{s}$, $\Phi = true$
7 If sym, $\beta := 0$, else compute β with $\|\text{triu}(A^T - A, 1)\|_2 \leqslant \beta$ by (1.5)
8 If $s - \beta < 10^{-16}\|A\|_\infty$, $[x, \delta]$ = verifySparseGen2(A,b), return
9 Rounding downwards, $A_s = A^{\text{sym}} - sI$, compute $L_s D_s L_s^T(A_s)$ by (1.12)
10 Compute approximate splitting $D_s \approx \widehat{D_s} S \widehat{D_s}^T$ according to (1.16)
11 Compute $L_1 \approx LD_s$ and $L_2 = SL_1^T$
12 Use (1.7) to compute α_- with $\|A_s - L_1 L_2\|_2 \leqslant \alpha_-$
13 If $\alpha_- \geqslant s - \beta$ in rounding downwards, improve α_- by (2.5)
14 If $\alpha_- < s - \beta$ in rounding downwards, $\nu_- =$ `sum(diag(S) > 0)`, go to Step 17
15 Compute α_- with $\|A_s - L_s D_s L_s^T\|_2 \leqslant \alpha_-$ as in Table 3, $\nu_- = \pi(D_s)$
16 If $\alpha_- \geqslant s - \beta$ rounded downwards, first verification failed, go to Step 25
17 Rounding upwards, $A_s := A^{\text{sym}} + sI$ and compute $L_s D_s L_s^T(A_s)$ by (1.12)
18 Compute approximate splitting $D_s \approx \widehat{D_s} S \widehat{D_s}^T$ according to (1.16)
19 Compute $L_1 \approx LD_s$ and $L_2 = SL_1^T$
20 Use (1.7) to compute α_+ with $\|A_s - L_1 L_2\|_2 \leqslant \alpha_+$
21 If $\alpha_+ \geqslant s - \beta$ in rounding downwards, improve α_+ by (2.5)
22 If $\alpha_+ < s - \beta$ in rounding downwards, $\nu_+ =$ `sum(diag(S) > 0)`, go to Step 24
23 Compute α_+ with $\|A_s - L_s D_s L_s^T\|_2 \leqslant \alpha_+$ as in Table 3, $\nu_+ = \pi(D_s)$
24 Set $\alpha = \max(\alpha_-, \alpha_+)$, if $\alpha < s - \beta$ rounded downwards, go to Step 26
25 If Φ, $\Phi = false$, $s = s/5$, go to Step 9, else $\nu_- = 0$
26 If $\nu_- \neq \nu_+$, verification failed, $[x, \delta]$ = verifySparseGen2(A,b), return
27 $[x, \delta] = \text{ErrorBound}(A, b, s - \alpha - \beta, \text{“}solve\text{“})$ using LDL^T for solve

TABLE 7
Verified error bounds for $A^{-1}b$ for nearly symmetric sparse input matrix A

method in [40, Theorem 1.1] relied on approximate LDL^T-decompositions of $A + sI$ and $A - sI$ for a shift s being an anticipated lower bound of $\sigma_{\min}(A)$. In the original paper we used LDL^T, here we use the decompositions $L_1 L_2$ presented in Part I of this note, were $L_2 = SL_1^T$ for a signature matrix S. There are two advantages. First, the inertia of S is trivial to compute. Second and more important, the entries of the residual $A_s - L_1 L_2$ for $A_s = A \pm sI$ compute as one dot product, whereas $A_s - LDL^T$ requires the computation of the product of three matrices. Hence, in the former case we can expect better bounds for the spectral norm of the residuals. Only if the residual $A_s - L_1 L_2$ is not small enough for a verification we turn to $A_s - LDL^T$ as in the original paper. In that case we use Algorithm `residualBoundLDLT` as in Table 3.

The rounding mode is specified in “verifySparseNearSym2” when necessary as in

Steps 9, 13, 14, 16, 17, 21, 22 and 24. Otherwise, rounding to nearest is advisable, but not mandatory. For example, in Step 8, the quick check whether it makes sense to continue, the rounding mode is not important.

We formulate the following Algorithm "verifySparseNearSym2" such that it is working for unsymmetric A as well. We start the analysis with the symmetric case $A^T = A$, i.e., sym = *true* in Step 2, $A^{\text{sym}} = A$ in Steps 9 and 17, and $\beta = 0$ in Step 7. After that we turn to the unsymmetric case.

Lines 4 – 6 are as in subalgorithm `verifySparseNearSym1` in Part I of this note. In Line 6 the factors L, D of A are used to compute s, an anticipated lower bound for the smallest singular value of A. As in `verifySparseNearSym1` we call in Step 8 immediately "verifySparseGen2" if successful finishing is unlikely because s is too small.

In order to distinguish the factors, we denote A_s in Lines 9 and 17 by $\texttt{As}_-$ and $\texttt{As}_+$, respectively. The matrix $\texttt{As}_-$ in Line 9 is computed in rounding downwards and therefore a lower bound on $A - sI$, i.e., $\texttt{As}_- = A - sI - \Delta_-$ for a diagonal and nonnegative matrix Δ_-, and similarly for $\texttt{As}_+ = A + sI + \Delta_+$ for diagonal $\Delta_+ \geqslant 0$.

Suppose matrices P_-, Q_-, P_+, Q_+ are given such that

$$\|\texttt{As}_- - P_- Q_- P_-^T\|_2 \leqslant \alpha_- \qquad \text{and} \qquad \|\texttt{As}_+ - P_+ Q_+ P_+^T\|_2 \leqslant \alpha_+ . \tag{5.1}$$

Denote the eigenvalues of symmetric $M \in \mathbb{F}^{n\times n}$ by $\lambda_1(M) \geqslant \ldots \geqslant \lambda_n(M)$ and let k be the index of the smallest positive eigenvalue of Q_-. Then (1.6) implies

$$\lambda_k(A) = \lambda_k(A - sI) + s \geqslant \lambda_k(\texttt{As}_-) + s \geqslant \lambda_k(P_- Q_- P_-^T) + s - \alpha_- > s - \alpha_- \ .$$

Denote by ℓ the index of the smallest positive eigenvalue of Q_+ such that $\lambda_{\ell+1}(Q_+) \leqslant 0$. Then we conclude similarly

$$\lambda_{\ell+1}(A) = \lambda_{\ell+1}(A + sI) - s \leqslant \lambda_{\ell+1}(\texttt{As}_+) - s \leqslant \lambda_{\ell+1}(P_+ Q_+ P_+^T) - s + \alpha_+ \leqslant -s + \alpha_+ \ .$$

The smallest singular value of A is equal to the smallest absolute value of an eigenvalue $\lambda_\nu(A)$. If the inertia of Q_- and Q_+ coincide, then $k = \ell$ and the ordering of the $\lambda_\nu(A)$ implies

$$\sigma_{\min}(A) = \min\left(-\lambda_{k+1}(A), \lambda_k(A)\right) \geqslant s - \max(\alpha_-, \alpha_+) \ . \tag{5.2}$$

Now in Steps 9 – 11 an approximate decomposition $\texttt{As}_- \approx L_1 S L_1^T$ is computed. Note that the computation of $L_2 = S L_1^T$ does not cause rounding errors because S is a signature matrix, i.e., diagonal with entries ±1 on the diagonal. Hence $L_1 L_2 = L_1 S L_1^T$. Then α_- is computed and possibly improved in Step 13 such that $\|\texttt{As}_- - L_1 S L_1^T\| \leqslant \alpha_-$. If $\alpha_- < s$ in Step 14, we set $P_- := L_1$ and $Q_- := S$. Then the number k of positive eigenvalues of Q_- is equal to ν_- and $\lambda_k(A) > s - \alpha_-$. If $\alpha_- \geqslant s$ in Step 14, we set $P_- := L_s$ and $Q_- := D_s$ and compute in Step 15 the upper bound α_- for $\|A_s - L_s D_s L_s^T\|_2$ using Algorithm `residualBoundLDLT` in Table 3. The number k of positive eigenvalues of Q_- is equal to ν_- which is computed by $\pi(D)$ based on Algorithm `NumPosEV` in Table 6. Hence $\lambda_k(A) > s - \alpha_-$ as well.

If $\alpha_- \geqslant s$, the verification is not yet successful for the choice of s. In that case we go to Step 25 to try once more with decreased s.

The computations in Lines 17 – 23 are similar to those in Lines 9 – 15 replacing the subindex "-" by "+". It follows that the number ℓ of positive eigenvalues of Q_+ is equal to ν_+ and that $\lambda_{\ell+1}(A) \leqslant -s + \alpha_+$. If $\alpha := \max(\alpha_-, \alpha_+) < s$ in Step 24 and $\nu_- = \nu_+$ in Step 26, then $k = \ell$ and (5.2) implies $\sigma_{\min}(A) \geqslant s - \alpha > 0$.

If $\alpha := \max(\alpha_-, \alpha_+) \geqslant s$ in Step 24, then as before a reason may be that s is too large. In that case we reduce s and try the verification from Lines 9 – 24 again. If still $\alpha \geqslant s$ or $\nu_- \neq \nu_+$ in Step 26, then the verification failed and we turn to subalgorithm "verifySparseGen2".

Now suppose the input matrix A is not symmetric, i.e., sym = $false$ in Step 2, A^{sym} is used in Steps 9 and 17, and β is computed in Step 7. As a consequence and if the algorithm runs to completion, $s - \alpha$ is a lower bound on the smallest singular value of A^{sym}. Hence

$$\sigma_{\min}(A) \geqslant \sigma_{\min}(A^{\text{sym}}) - \|A^{\text{sym}} - A\|_2 \geqslant s - \alpha - \beta \ .$$

If the verification was successful, the positive lower bound $s - \alpha$ or $s - \alpha - \beta$ on $\sigma_{\min}(A)$ verifies that the matrix A is nonsingular, and entrywise bounds for the solution of the linear system are computed by Algorithm `ErrorBound` in Table 1 of Part I of this note. To compute almost always maximally accurate inclusions we may use Algorithm `ErrorBound3` as in Table 4.

The difference to Algorithm "verifySparseNearSym1" in Part I of this note is as follows. Here the input matrix is shifted by s to the left and right. If the inertia of the corresponding LDL^T-decompositions are the same, then $s - \alpha$ or $s - \alpha - \beta$ is a lower bound for $\sigma_{\min}(A)$ subject to the maximum α of the residual bounds. The drawback is some additional fill-in of the factors L of the shifted matrices.

In "verifySparseNearSym1" in Part I of this note we decompose $A \approx L_1 L_2$ and estimate the smallest singular value of L_1 using a Cholesky decomposition of $L_1 L_1^T$ subject to a norm bound of the residual $A - L_1 L_2$. That turns out to be faster, but in rare cases it is less robust in the sense that it fails where "verifySparseNearSym2" succeeds, see the numerical results in Section 10. Another reason is that eventually in Steps 15 and/or 23 the residual of the three-fold product $A_s - L_s D_s L_s^T$ is computed by `residualBoundLDLT` in Table 3 improving the estimates.

6. General matrices. As in [40, 42] our method for linear systems with general matrix uses the augmented matrix

$$B := \begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix} \tag{6.1}$$

the singular values of which are ± the eigenvalues of A. This matrix is used in [48] as well. Next we present and discuss Algorithm "verifySparseGen2" in Table 8.

As in the symmetric case we explore on Theorem 1.1 published in [40, Theorem 1.1]. The original method relied for general A on approximate LDL^T-decompositions of $A \pm sI$ for a shift s being an anticipated lower bound of $\sigma_{\min}(A)$. In contrast to the symmetric case, one shift suffices for the augmented matrix B because the inertia of B for nonsingular A is known beforehand. We do not assume nonsingularity of A but prove it a posteriori so that all deductions are true.

Rather than LDL^T as in [40, Theorem 1.1] we use, as in the symmetric case, a decomposition $L_1 L_2$ of $B - sI$ as presented in Part I of this note, where $L_2 = S L_1^T$ for a signature matrix S. That implies the same advantages as in the symmetric case.

In contrast to [40, 42, 48] we proceed for general matrices as follows. After equilibrating the original matrix A we compute an LDL^T-decomposition of the augmented matrix B by (1.12). As has been observed in Part I in some cases the computed D is singular, even for moderately conditioned input matrix. That should not happen,

```
1   function [x, δ] = verifySparseGen2(A,b)
2       Equilibrate A by (1.11)
3       Let B the augmented matrix (6.1)
4       Compute LDL^T(B) by (1.12)
5       If nnz(D) < 2n, compute LDL^T(B) by (1.13)
6       If nnz(D) < 2n, verification failed, return
7       Compute s̃(L, D) ≲ σ_min(B) by (1.14) and set s := 0.9s̃, Φ = true
8       Rounding downwards, B_s := B − sI and compute L_s D_s L_s^T(B_s) by (1.12)
9       Compute approximate splitting D_s ≈ D̂_s S D̂_s^T according to (1.16)
10      Compute L_1 ≈ L D_s and L_2 = S L_1^T
11      Use (1.7) to compute α with ‖B_s − L_1 L_2‖_2 ⩽ α
12      If α < s,  improve α by (2.5)
13      If α < s, ν = sum(Ds) > 0, else improve α by (2.6), ν = π(D_s)
14      If  α < s, go to Step 16
15      If Φ, Φ = false, s = s/5, go to Step 8, else ν = 0
16      If ν ≠ n, verification failed, return
17      [x, δ] = ErrorBound(B, [0; b], s − α, "solve") using LDL^T for solve
```

TABLE 8
Verified error bounds for $A^{-1}b$ for general sparse input matrix A

and we cure it as in (1.13). Fortunately that does not happen any more from Matlab Version 2026a on.

Based on the factors L, D we compute in Step 7 an anticipated lower bound s for the smallest singular value of B which is equal to that of A. Although B has double the size of A, the iteration (1.14) to compute s as a lower bound of $\sigma_{\min}(B)$ rather than of $\sigma_{\min}(A)$ is more robust due to the symmetry of B.

A splitting (1.16) of D is computed in Step 9, and in Step 10 the factors L_1, L_2 such that $L_1 L_2 \approx A$. The factor L_2 is L_1 multiplied by some signature matrix. That computation is error-free, so that as in subalgorithm "verifySparseNearSym1" in Part I of this note the factors L_1, L_2 have identical sets of singular values.

The remaining of the subalgorithm `verifySparseGen2` is, *mutatis mutandis*, identical to subalgorithm `verifySparseGen1` in Table 6 of Part I of this note. Hence, if successful, $s - \alpha$ is a lower bound for $\sigma_{\min}(B) = \sigma_{\min}(A)$.

Error bounds for the solution of the original linear system $Ax = b$ use that

$$\begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \tag{6.2}$$

implies $x = A^{-1}b$ and we proceed as in Part I of this note.

As for "verifySparseNearSym2" the difference is that "verifySparseGen2" shifts the augmented matrix and computes a lower bound for $\sigma_{\min}(B)$ using Sylvester's law of inertia. In contrast, "verifySparseGen1" relies on the factorization $L_1 L_2$ of the original augmented matrix B without shift and computes a lower bound for $\sigma_{\min}(B)$

based on a Cholesky factorization of $L_1 L_1^T$. In rare cases that does not allow a verification where "verifySparseGen2" does. In general, however, "verifySparseGen2" seems slower because the decomposition of the shifted matrix causes additional fill-in, see the computational results in Section 10.

7. Least squares problems and underdetermined linear systems. The methods in Part I and Part II of this note can be used to compute verified error bounds for the solution of least squares problems and underdetermined systems of linear equations with sparse matrix.

For $A \in \mathbb{C}^{m \times n}$ with $m > n$ and $b \in \mathbb{C}^m$ define (cf. [11, Chapter 20]).

$$\begin{pmatrix} 0 & A^H \\ A & -I_m \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \quad \Rightarrow \quad A^H y = 0 \text{ and } Ax - y = b \ , \tag{7.1}$$

where I_m denotes the $m \times m$ identity matrix. Multiplying the second equation by A^H yields $A^H A x = A^H b$. For full-rank A and A^+ denoting the classical Moore-Penrose inverse [11] it follows that $x = (A^H A)^{-1} A^H b = A^+ b$ is the unique least squares solution minimizing $\|Ax - b\|_2$.

The system matrix in (7.1) is symmetric indefinite, so our subalgorithms "verifySparseNearSym1" and "verifySparseNearSym2" are applicable. In [44] we published algorithms to compute verified error bounds for least squares problems and underdetermined linear systems with full matrix. In that paper we used

$$\begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} .$$

Although the system matrix is not Hermitian, we showed numerical evidence in [44] that the computed inclusions are sometimes more accurate than using (7.1). However, for our present approach we stick to the Hermitian input matrix.

For an underdetermined system of linear equations $Ax = b$ with $A \in \mathbb{C}^{m \times n}, b \in \mathbb{C}^m$ and $m < n$ define

$$\begin{pmatrix} -I_n & A^H \\ A & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \quad \Rightarrow \quad Ax = b \text{ and } A^H y = x \ , \tag{7.2}$$

so that multiplying the second equation by A yields $AA^H y = Ax = b$. If A has full rank, then $x = A^H y = A^H (AA^H)^{-1} b = A^+ b$ is the unique solution of $Ax = b$ with minimal $\|x\|_2$.

When computing error bounds for the square linear systems (7.1) or (7.2) by our algorithms, the nonsingularity of the augmented matrix is verified. In turn, that implies that A has full rank and our conclusions are valid.

Before solving (7.1) and (7.2) we get rid of zero rows and columns of the input matrix. Let b and non-square A be given. For a least squares problem, zero rows of A and corresponding entries of b can be deleted without changing the solution. For an underdetermined linear system, zero columns of A can be deleted without changing b and padding the solution of this problem with corresponding zeros. In both cases the problem may change from least squares to square or underdetermined, or vice versa.

The linear systems in (7.1) and (7.2) can be solved by "verifySparseNearSym1" or "verifySparseNearSym2". However, in Algorithm `verifySparselss2` in Table 11 we call recursively "verifySparselss2". Since the augmented matrix is square, that leads directly to the case distinctions for real or complex matrices.

In subalgorithms "verifySparseNearSym1" or "verifySparseNearSym2" the symmetric equilibration (1.11) (which is (3.3) in Part I of this note) is applied, i.e., the Sinkhorn-Knopp algorithm. That means the rows of A^T and rows of A are equilibrated from the left, and similarly from the right. Thus, although B is symmetric, the matrix A is equilibrated independently from the left and right. That increases the robustness of the approach.

There are other possibilities to define the solution of an underdetermined linear system. For example, Matlab computes a solution of $Ax = b$ with at most m nonzero entries. This can be done as follows. First, an LU-decomposition of A^H is computed with partial pivoting. The only purpose is to obtain the pivoting information. Say that is stored in a vector p. Then the x is the solution of $\tilde{A}x = b$ where $\tilde{A}$ consists of the columns $p_1, \ldots, p_m$ of A.

The minimum norm solution A^+b is computed by `lsqminnorm` in Matlab, and we compare to this instead of the "backslash" operator.

8. Systems of nonlinear equations. In this section we need some more details on interval operations, in particular the use of INTLAB [41]. If an operation involves one operand of type `intval`, then the operation is executed using interval arithmetic, i.e., the result is an inclusion of the true real (or complex) result. That is true for all kinds of operations including vectors, matrices, standard functions and so forth. For example, in `a*(b+c)` interval addition and multiplication is used if `b` and/or `c` is of type `intval`. There are toolboxes for gradients, Hessian, Taylor series and Taylor models in INTLAB. Here we use the gradient toolbox to compute an approximation of the function value and the Jacobian of a function. If the argument is of type `intval`, then mathematically rigorous inclusions are computed. For details, see [41, 43].

Let a nonlinear system $f(x) = 0$ with continuously differentiable function $f : \mathbf{D} \to \mathbb{R}^n$ with compact and convex $\mathbf{D} \in \mathbb{IR}^n$ be given. We assume that a Matlab program `f` is given such that `f(x)` evaluates $f(x)$.

Let $\tilde{x} \in \mathbf{D}$ be given. Denote the Jacobian of f at x by $J_f(x)$. Then by the n-dimensional Mean Value Theorem, for $x \in \mathbf{D}$ there exist $\xi_1, \ldots, \xi_n$ in the convex union $x \underline{\cup} \tilde{x}$ of x and $\tilde{x}$ with

$$f(x) = f(\tilde{x}) + \begin{pmatrix} \nabla f_1(\xi_1) \\ \cdots \\ \nabla f_n(\xi_n) \end{pmatrix} \big(x - \tilde{x}\big) \tag{8.1}$$

using the component functions $f_i : \mathbf{D}_i \to \mathbb{R}$ where $\mathbf{D}_i := \{x_i : x \in \mathbf{D}\} \in \mathbb{IR}$. As is well-known, the ξ_i cannot, in general, be replaced by a single ξ, so that the matrix in (8.1) is only rowwise equal to some Jacobian J_f of f.

Using INTLAB's gradient toolbox, the call `J = f(gradientinit(x))` computes for $x \in \mathbb{F}^n \cap \mathbf{D}$ an approximation of the function value $f(x)$ and of the Jacobian $J_f(x)$. More important, let $\mathtt{X} \in \mathbb{IF}^n$ be an interval vector with $\mathtt{X} \subseteq \mathbf{D}$. Then the call

$$\mathtt{Y} = \mathtt{f(gradientinit(X))} \tag{8.2}$$

computes `Y` such that $\mathtt{Y.x} \in \mathbb{IF}^n$ is an interval vector with $\{f(x) : x \in \mathbf{X}\} \subseteq \mathtt{Y.x}$, and `Y.dx` is an interval matrix $\mathtt{Y.dx} \in \mathbb{IF}^{n\times n}$ with $\{\nabla f_k(\xi) : \xi \in \mathtt{X}\} \subseteq \mathtt{Y}_k$ for all $k \in \{1, \ldots, n\}$. For a subset X of $\mathbb{R}^n$ define $\mathrm{hull}(X) \in \mathbb{IR}^n$ by

$$\mathrm{hull}(X) := \bigcap \{\mathbf{Z} \in \mathbb{IR}^n : X \subseteq \mathbf{Z}\} . \tag{8.3}$$

For $x, \tilde{x} \in \mathbf{D}$ also $\mathbf{X} := \mathrm{hull}(x \underline{\cup} \tilde{x}) \subseteq \mathbf{D}$, and (8.2) implies

$$\begin{pmatrix} \nabla f_1(\xi_1) \\ \cdots \\ \nabla f_n(\xi_n) \end{pmatrix} \in \texttt{Y.dx} \tag{8.4}$$

for all $\xi_1, \ldots, \xi_n \in \mathbf{X}$. Therefore [43, Theroem 13.1], using interval operations the Mean Value Theorem can be written in the following elegant way.

THEOREM 8.1. *Let continuously differentiable* $f : \mathbf{D} \to \mathbb{R}^n$ *with* $\mathbf{D} \in \mathbb{IR}^n$ *and* $\texttt{x}, \texttt{xs} \in \mathbf{D} \cap \mathbb{F}^n$ *be given. Define* `Y = f(gradientinit(hull(x,xs)))`*. Then*

$$f(x) \in f(\tilde{x}) + \texttt{Y.dx}\big(x - \tilde{x}\big) \ . \tag{8.5}$$

That is true because, by the principle automatic differentiation with interval arithmetic, `Y.dx` is an inclusion of the left hand side of (8.4) and not only of $\{\frac{\partial f}{\partial x}(x) : x \in \mathbf{X}\}$.

Using this we can formulate [43, Theorem 13.3] the following theorem to compute error bounds for a solution of a system of nonlinear equations $f : \mathbb{R}^n \to \mathbb{R}^n$ based on some approximate solution $\tilde{x} \in \mathbb{R}^n$.

THEOREM 8.2. *Let continuously differentiable* $f : D \to \mathbb{R}^n$ *and* $\tilde{x} \in \mathbb{R}^n$*,* $\mathbf{X} \in \mathbb{IR}^n$*,* $R \in \mathbb{R}^{n \times n}$ *with* $0 \in \mathbf{X}$ *and* $\tilde{x} + \mathbf{X} \subseteq D$ *be given. Suppose*

$$S(\mathbf{X}, \tilde{x}) := -Rf(\tilde{x}) + \big(I - RJ_f(\tilde{x} + \mathbf{X})\big)\mathbf{X} \ \subseteq \ \mathit{int}(\mathbf{X}) \tag{8.6}$$

with int denoting the topological interior. Then R *and all matrices* $M \in J_f(\tilde{x} + \mathbf{X})$ *are nonsingular, and there is a unique root* $\widehat{x}$ *of* f *in* $\tilde{x} + S(\mathbf{X}, \tilde{x})$*.*

The bound $\tilde{x} + S(\mathbf{X}, \tilde{x})$ is computable and is mathematically rigorous including the proof of uniqueness of the root $\widehat{x}$ of f in $\tilde{x} + S(\mathbf{X}, \tilde{x})$.

A practical application as implemented in Algorithm `verifynlss` in INTLAB uses an approximate inverse R of $J_f(\tilde{x})$ which is, in general, a full matrix. Therefore, an inclusion based on Theorem 8.2 is hardly applicable to large systems of nonlinear equations even if the Jacobian is sparse.

In practice, however, often individual variables x_k have few dependencies on other variables. As a consequence, the Jacobian becomes sparse, often a banded matrix. Next we show how the assumption (8.6) of Theorem 8.2 can be verified by solving a linear system with point matrix and interval right hand side. Then our methods for the solution of sparse linear systems are applicable.

We follow [43, Section 13] and compute an inclusion $\mathbf{J} \in \mathbb{IF}^{n \times n}$ of $J_f(\tilde{x} + \mathbf{X})$ as in (8.2). Hence (8.4) implies that for all $\xi \in \tilde{x} + \mathbf{X}$ and for all $k \in \{1, \ldots, n\}$ the gradient $\nabla f_k(\xi)$ is included in the k-th row of $\mathbf{J}$, and Theorem 8.1 is applicable. Denote $M = \mathrm{mid}(\mathbf{J})$ and $\Delta := \mathrm{rad}(\mathbf{J})$, where M and Δ are computed such that $\mathbf{J} \subseteq M \pm \Delta$. Assume that M is nonsingular and suppose

$$\{y \ : \ My = -f(\tilde{x}) - \varrho x, \ -\Delta \leqslant \varrho \leqslant \Delta, \ x \in \mathbf{X}\} \subseteq \mathbf{Y} \ . \tag{8.7}$$

for $\mathbf{Y} \in \mathbb{IF}^n$. Then $\mathbf{Y} \subset \mathrm{int}(\mathbf{X})$ implies (8.6). To see this set $R := M^{-1}$ and observe

$$-M^{-1}f(\tilde{x}) + \big(I - M^{-1}\big(M + \varrho\big)\big)x = M^{-1}\big(- f(\tilde{x}) - \varrho x\big)$$

for $x \in \mathbb{R}^n$ and $\varrho \in \mathbb{R}^{n \times n}$. Applying this to $x \in \mathbf{X}$ and using $|\varrho| \leqslant \Delta$ proves (8.6) for $R := M^{-1}$. Hence there is a unique solution $\widehat{x}$ of $f(x) = 0$ with $\widehat{x} \in \tilde{x} + \mathbf{Y}$. That transforms

the problem of computing verified bounds for the solution of a nonlinear system to the solution of a linear system with interval right hand side, and the successful termination of the sparse linear system solver proves the nonsingularity of M.

Now $\mathbf{X}$ is an anticipated inclusion of the difference of the true solution $\hat{x}$ of the nonlinear system $f(x) = 0$ to the approximate solution $\tilde{x}$. And if successful, i.e. $\mathbf{Y} \subset \mathrm{int}(\mathbf{X})$, then $\hat{x} - \tilde{x} \in \mathbf{Y}$. If $\tilde{x}$ is a good approximation, then $\mathbf{X}$ is small in magnitude and essentially symmetric to the origin. As a consequence we further simplify (8.7) by using the magnitudes[5] $\overline{X}$ and $\overline{Y}$ of $\mathbf{X}$ and $\mathbf{Y}$, and set $\mathbf{X} := [-\overline{X}, \overline{X}]$ and $\mathbf{Y} := [-\overline{Y}, \overline{Y}]$. Then $\overline{Y} < \overline{X}$ with entrywise comparison is equivalent to $\mathbf{Y} \subset \mathrm{int}(\mathbf{X})$.

Let a matrix $A \in \mathbb{F}^{n \times n}$ and interval right hand side $\mathbf{b} \in \mathbb{IF}^n$ be given. We are interested in computing an inclusion of the "outer inclusion set", see (5.1) in Part I of this note:

$$\Sigma(A, \mathbf{b}) := \{x \in \mathbb{R}^n : \exists b \in \mathbf{b} \text{ with } Ax = b\} \ . \qquad (8.8)$$

To that end we use Algorithm "verifySparselss1" as in Table 7 in Part I of this note with small modifications. First, we remove the check for least squares and underdetermined problems. Furthermore, the only modification is replacing the calls of "ErrorBound" in last line in subalgorithms "verifySparseSPD2", "verifySparseNearSym2" and "verifySparseGen2" by the call of "ErrorBoundI" as shown in Table 9.

```
1   function [xs,delta] = ErrorBoundI(A,b,s,@solve)
2     mu = b.mid; r = b.rad;
3     xs = solve(A,mu);
4     xs = xs - solve(A,spProdK(A,xs,-1,mu,2));
5     [rho,err] = spProdK(A,xs,-1,mu,2);
6     setround(1)
7     delta = norm(abs(rho) + err + r , 2)/s;
8   end  % function ErrorBoundI
```

TABLE 9

Executable Matlab/INTLAB code to compute verified error bounds for the solution of a real or complex system of linear equations with interval right hand side

The input parameter s is a lower bound on $\sigma_{\min}(A)$ and `@solve` is some routine delivering an approximate solution of a linear system. As in the original algorithm "ErrorBound", `@solve` is based on the already computed decomposition in each of the subalgorithms.

The proof of correctness of Algorithm "ErrorBoundI" is as follows. Let $A \in \mathbb{F}^{n \times n}$ and $\mathbf{b} \in \mathbb{IF}^n$ be an interval vector. Then $\mu, r \in \mathbb{F}^n$ in Line 2 are computed such that $\mu - r \leqslant b \leqslant \mu + r$ for all $b \in \mathbf{b}$. In Line 3 an approximate solution $\tilde{x}$ of the midpoint equation $Ax = \mu$ is computed and is improved in Line 4 by one residual iteration. According to [47] that implies backward stability of $\tilde{x}$. Line 5 computes an inclusion `rho` ± `err` of the residual $A\tilde{x} - \mu$, such that $|A\tilde{x} - \mu| \leqslant$ |`rho`|+`err`. Now `delta` in Line 7

[5] Recall that for an interval quantity $\mathbf{Z}$ the magnitude $0 \leqslant \mathrm{mag}(\mathbf{X}) \in \mathbb{R}^n$ is the entrywise maximum absolute value, i.e., $|z| \leqslant \mathrm{mag}(\mathbf{Z})$ for all $z \in \mathbf{Z}$. That includes interval vectors and matrices with entrywise absolute value and comparison.

is computed in rounding upwards, and with the lower bound s on $\sigma_{\min}(A)$ it follows

$$\begin{aligned} |A^{-1}b - \tilde{x}| &\leqslant |A^{-1}|\,|b - A\tilde{x}| \\ &\leqslant |A^{-1}|\,(|\mu - A\tilde{x}| + r) \\ &\leqslant \|A^{-1}\,(|\mu - A\tilde{x}| + r)\,\|_\infty \mathbf{e} \\ &\leqslant \|A^{-1}\,(|\mu - A\tilde{x}| + r)\,\|_2 \mathbf{e} \\ &\leqslant \|A^{-1}\|_2 \||\mathtt{rho}| + \mathtt{err} + r\|_2 \mathbf{e} \\ &\leqslant \delta \end{aligned} \tag{8.9}$$

for all $b \in \mathbf{b}$. Let $\mathtt{J} \in \mathbb{IF}^{n\times n}$ be an inclusion of $J_f(\tilde{x} + \mathbf{X})$ computed as in (8.2) and consider

(8.10)
```
y = -f(intval(xs));
setround(1)
b = midrad( y.mid , y.rad + J.rad * mag(X) );
```

The first line computes an inclusion $\mathtt{y} \in \mathbb{IF}^n$ of $-f(\tilde{x})$ with $-f(\tilde{x}) \in$ `y.mid±y.rad`. The second statement switches the rounding to upwards, and finally $\mathbf{b}$ is an inclusion of `y.mid±`ϱ for all $|\varrho| \leqslant$ `y.rad + J.rad*mag(X)`. Thus $-f(\tilde{x}) - \varrho x \in \mathbf{b}$ for all $x \in \mathbf{X}$ and $|\varrho| \leqslant \Delta$. It follows that an inclusion $\mathbf{Y}$ of the linear system with matrix M and right hand side $\mathbf{b}$ satisfies (8.7). As a consequence, $\mathbf{Y} \subseteq \mathrm{int}(\mathbf{X})$ implies $\widehat{x} \in \tilde{x} + \mathbf{Y}$.

The algorithm to solve a system of nonlinear equations works as follows. First we apply some Newton iterations to produce a good approximation $\tilde{x}$ of $f(x) = 0$. Then $f(\tilde{x})$ should be small and the magnitude of $\mathbf{b}$ dominated by the radius Δ of the inclusion of $J_f(\tilde{x} + \mathbf{X})$. The residual of the linear system cannot become smaller than the magnitude and radius of $\mathbf{b}$, which in turn increases with the sensitivity of the problem. Therefore, there is no need to improve an approximate solution of $My = \mathbf{b}$ by a residual iteration and we may apply algorithms "verifySparselss1" or "verifySparselss2" with using Algorithm "ErrorBoundI" as in Table 9 rather than "ErrorBound".

Executable Matlab/INTLAB code of Algorithm `verifySparseNlss` to compute rigorous error bounds for the solution of a nonlinear system $f(x) = 0$ based on an approximate solution $\tilde{x}$ is given in Table 10. The rationale is as follows. In Line 2 the rounding is set to nearest, and in Lines 5 – 13 some Newton iterations are applied to improve the approximation $\tilde{x}$. The statement `y = f(gradientinit(xs))` in Line 7 computes `y` such that `y.x` $\approx f(\tilde{x})$ and `y.dx` is an approximation of the Jacobi matrix of f at $\tilde{x}$ using the gradient toolbox, which in turn is based on forward automatic differentiation [4, 10] and implemented in INTLAB [41]. Therefore Line 8 is one (approximate) Newton step.

The quantity `ys` in Line 14 is an inclusion of $-f(\tilde{x})$ and `Y` its magnitude. Lines 17 – 31 are an interval iteration adapted to the description in [43]. Recall that `Y` is a positive real vector, and the anticipated inclusion of the error with respect to $\tilde{x}$ is the interval vector $[-Y, +Y]$. Line 19 is one step of the so-called epsilon inflation introduced in [38]. The target is $Y < X$, or equivalently $[-Y, +Y] \subseteq \mathrm{int}[-X, +X]$. The inclusion may fail if $[-X, +X]$ is too narrow, so $[-X, +X]$ is intentionally widened a little bit. The success of the epsilon-inflation can be analysed theoretically, see [43]. On the other hand $[-X, +X]$ should not be too wide because that widens the Jacobian and may prevent $Y < X$.

The purpose of the epsilon-inflation is to identify a good candidate for inclusion. The right hand side $\mathbf{b}$ should be a narrow interval around $f(\tilde{x})$. More precisely,

```
1    function [X,kxs,kY] = verifySparseNlss(f,xs)
2      setround(0)
3      n = size(xs,1); phi = 1e-14*sqrt(n);
4      dxs = abs(xs); kxs = 0;
5      while ( kxs < 15 )           % at most 15 Newton iterations
6        kxs = kxs + 1; xsold = xs;
7        y = f(gradientinit(xs));   % function value and gradient
8        xs = xs - y.dx\y.x;        % approximate Newton iteration
9        d = abs(xs-xsold);
10       if all(d<.5*abs(xs)) && ( norm(d,inf)<=phi*norm(xs,inf) )
11         break
12       end
13     end
14     ys = -f(intval(xs));         % inclusion of f(xs)
15     Y = mag(ys);                 % magnitude of ys
16     kY = 0; setround(1)
17     while ( kY < 10 )
18       kY = kY + 1;
19       X = 1.01*Y + realmin;      % epsilon-inflation
20       JJ = f(gradientinit(midrad(xs,X)));
21       J = JJ.dx;                 % inclusion of Jacobian
22       b = midrad( ys.mid , ys.rad + J.rad*X );
23       [Ys,delta] = verifySparse(J.mid,b);
24       Y = abs(Ys) + delta;       % r.h.s. of (7.7)
25       if all( Y < X )
26         X = midrad(xs,Y);        % inclusion successful
27         return
28       elseif ( kY > 2 ) & all( Y > 2*X )
29         break                    % no hope for inclusion
30       end
31     end
32     X = intval(NaN(size(xs)));   % inclusion failed
33   end  % function verifySparseNlss
```

TABLE 10

Executable Matlab/INTLAB code to compute verified error bounds for the solution of a real or complex system of nonlinear equations

according to (8.9) around $-f(\tilde{x})$, where the minus-sign doesn't matter because our inclusion is symmetric to the origin. Therefore, basically $\pm 1.01|f(\tilde{x})|$ is our first choice. We need an inclusion $\mathbf{J} \in \mathbb{IF}^{n\times n}$ of $J_f(\mathbf{Z})$ with $\mathbf{Z} := \tilde{x} + \mathbf{X}$. The quantity[6] `JJ` in Line 20 satisfies $f(z) \in$ `JJ.x` and the Jacobian of f at z is in `JJ.dx` for all $z \in \mathbf{Z}$. Hence `J` in Line 21 is what we need. The next Line 22 computes $\mathbf{b}$ as in (8.10), and the next Line 23 uses some sparse linear system solver such as Sparse1 or Sparse2. That computes an inclusion $Ys \pm \delta$ of the linear system with matrix $M = \mathrm{mid}(\mathbf{J})$ and right hand side $\mathbf{b}$. The magnitude of the inclusion is `Y` as in Line 24, and if $Y < X$ is true for all entries then `midrad(xs,Y)` is an inclusion of the solution of the nonlinear system.

If $Y_k \geqslant X_k$ for some k, then the inclusion is tried again with X replaced by a

[6]In a practical implementation, of course, the same variable `J` is used in Lines 20 and 21.

little widened Y. In some way the interval iteration is also a Newton step. In each step a new Jacobian `J` at $\mathbf{Z}$ is computed, and the widened Y reflects the width of the previous `J`. If not successful after some 10 trials, the verification failed.

Unlike for linear systems we cannot expect, in general, maximally accurate inclusions because the lack of an accurate residual iteration and, more important, because of nonlinearities of f widening the Jacobi matrix. An essential aspect is that in the main error estimate in Section 5 of Part I for the error of the approximation $\tilde{x}$ we have to switch from the ∞-norm of the residual to the 2-norm. As a consequence and in combination with the lack of an accurate residual iteration, an inclusion of the solution of the linear system in Step 23 becomes wider. In turn, that inclusion is used as a next candidate allowing more nonlinearities to come into play, and so forth.

Another essential reason for potential failure is the magnitude of the inclusion `ys` of $-f(\tilde{x})$. After 4 interval iterations, X should be settled to become a promising candidate for an inclusion. In a typical situation where inclusion fails, $\|\texttt{ys.mid}\| = 5.3 \cdot 10^{-12}$ and $\|\texttt{ys.rad} + \texttt{J.rad} * \texttt{X}\| = 4.4 \cdot 10^{-19}$ for the right hand side `b = midrad( ys.mid , ys.rad + J.rad*X )`. Here `J.rad*X` is the product of two small quantities and negligible; the interval vector `b` is widened proportional to the condition number of the Jacobian allowing nonlinearities to spoil the inclusion.

The magnitude of the inclusion `ys` of $-f(\tilde{x})$ may be decreased by using higher precision, for example, the mp-toolbox [12]. To that end only the approximate solution $\tilde{x}$ and the inclusion $-f(\tilde{x}) \in \texttt{ys}$ is calculated in extended precision, the verification part is still executed in double precision. More precisely, let $\tilde{x}_{\mathrm{mp}}$ be an extended precision approximate solution. Then an inclusion of $\texttt{ys}_{\mathrm{mp}}$ of $f(\tilde{x}_{\mathrm{mp}})$ is computed in extended precision, and `ys` is the narrowest double precision interval enclosing $\texttt{ys}_{\mathrm{mp}}$ in Line 14 of Algorithm "verifySparseNlss". Furthermore, let `xs` be an inclusion of $\tilde{x}_{\mathrm{mp}}$ in double precision. Then `midrad(xs,X)` in Line 20 is replaced by `xs + midrad(0,X)`, and accordingly `midrad(xs,Y)` in Line 26 by `xs + midrad(0,Y)`. That ensures that $\tilde{x}_{\mathrm{mp}}$+`X` and $\tilde{x}_{\mathrm{mp}}$+`Y` are included, respectively. Numerical evidence of the effectiveness is presented in the final section.

9. Complex sparse linear systems, data with tolerances and the final sparse lss algorithms. As noted in Part I, the LDL^T-decomposition for sparse matrices is restricted to real data. Therefore we proceed for complex linear systems as in Section 10 in Part I of this note. Data with tolerances may be treated as in Section 5 of Part I of this note.

To distinguish our algorithms, we use `verifySparselss1` for our algorithm presented in Part I (also called Sparse1 in there) and use `verifySparselss2` for the algorithm presented in this Part II (henceforth called Sparse2). Executable code of Algorithm `verifySparselss2` including least squares problems and underdetermined linear systems is presented in Table 11.

The rationale of Sparse2 is as follows. As has been explained Algorithms "verifySparseSPD2" and "verifySparseNearSym2" are correct for unsymmetric matrices. To that end an extra parameter `nearsym` is introduced. If not specified, the default is false. If false, `verifySparselss2` checks for near symmetry in Line 22. If true or A is symmetric, then "verifySparseSPD2" is called which calls "verifySparseNearSym2" if not successful. If `nearsym` is an input parameter and true, then in any case "verifySparseSPD2" is called. That should only be done if the departure $\|A^T - A\|$ from symmetry is small compared to the condition number. If successful, the result is correct independent of the magnitude of $\|A^T - A\|$.

Otherwise, the algorithm first checks for the type of problem, namely $m > n$ for

```
function [xs,delta] = verifySparselss2(A,b,nearsym)
% Approximate solution xs of Ax=b with rigorous error bound delta
  if nargin<3, nearsym = false; end
  [m,n] = size(A);
  if m>n                    % least squares problem
    B = [ sparse(n,n) A' ; A -speye(m) ];
    [xs,delta] = verifySparselss2(B,[zeros(n,size(b,2));b],nearsym);
    xs = xs(1:n,:);
    delta = delta(1:n,:);
    return
  elseif m<n                % underdetermined linear system
    B = [ -speye(n) A' ; A sparse(m,m) ];
    [xs,delta] = verifySparselss2(B,[zeros(n,size(b,2));b],nearsym);
    xs = xs(1:n,:);
    delta = delta(1:n,:);
    return
  end
  if isreal(A)              % linear system with square matrix
    if isreal(b)            % A and b real
      xs = NaN;
      symm = isequal(A',A);
      if ( ~symm ) && ( ~nearsym )   % determine nearsym
        nearsym = all(abs(nonzeros(A'-A)))<1e-15*norm(A,inf);
      end
      if symm || nearsym    % A symmetric or near symmetric
        [xs,delta] = verifySparseSPD2(A,b);
      end
      if isnan(xs(1))       % SPD2 and Sym2 failed
        [xs,delta] = verifySparseGen2(A,b);
      end
    else                    % A real, b complex
      [xs,delta] = verifySparselss2(A,[real(b) imag(b)],nearsym);
      n = size(A,1);
      m = size(b,2);
      xs = complex(xs(:,1:m),xs(:,m+1:end));
      delta = reshape(vecnorm(reshape(delta,[],2),2,2),n,[]);
    end
  else                      % A complex, square matrix
    n = size(A,1);
    A = [real(A) -imag(A);imag(A) real(A)];
    b = [real(b);imag(b)];
    [xs,delta] = verifySparselss2(A,b);
    xs = complex(xs(1:n,:),xs(n+1:end,:));
    delta = reshape(delta,n,[])';  % take care of multiple r.h.s.
    delta = reshape(vecnorm(reshape(delta,2,[]),2),size(b,2),[])';
  end
end  % function verifySparselss2
```

Table 11

Final algorithm to compute verified error bounds for the solution of a real or complex sparse square linear system, for a least squares problem and an underdetermined linear system, all for multiple right hand sides

a least squares problem and $m < n$ for an underdetermined system of equations. We omitted the treatment of eliminating zero rows and columns. In either case Algorithm `verifySparselss2` is called recursively using (7.1) or (7.2), respectively using the parameter nearsym. If $m = n$, verified error bounds for a linear system with square matrix are computed with code identical to Algorithm `verifySparselss1` in Table 7 in Part I of this note except for the handling of nearly symmetric matrix.

We discussed how to compute inclusions with improved accuracy as described in Section 2 by storing an approximation by an unevaluated sum of three instead of two parts as by Algorithm `ErrorBound3`. The computational penalty is small because it affects only the final residual iteration. In the median over all examples the computing time increased by some 9%, over examples with at least 5 seconds computing time less than 2%. Therefore we switch in the practical implementation to more accurate residuals if the maximum relative error of the inclusion is beyond some threshold. We used the threshold 10^{-15} for the maximal error of all entries of the inclusion.

Computational results comparing our two algorithms to each other and to Matlab's backslash operator are presented in the next section.

10. Test results. As in Part I of this note, our computing environment is a Panasonic laptop CF-SV with Intel(R) Core(TM) i7-10810U CPU with 1.10/1.61 GHz and 16 GB RAM. We use Matlab version 2026a [23] under Windows 10. Henceforth we call Algorithm `verifySparselss1` "Sparse1" as in Part I, and Algorithm `verifySparselss2` of this Part II "Sparse2". Strictly speaking, in order to present a fair comparison to [48], Sparse1 in Part I used the mp-toolbox Advanpix [12] for accurate residual computations, whereas here in Part II we use INTLAB's `spProdK` for both Sparse1 and Sparse2. Therefore the results of Sparse1 in Part I and of Sparse1 of Part II may differ.

We use the same set of test matrices from the Suite Sparse Matrix Collection [5] with the interface [16] as in Part I including complex square matrices with

$$10^3 \leqslant n \leqslant 10^5 \quad \text{and} \quad 10^7 \leqslant \kappa_2(A) \leqslant 10^{16} \quad \text{and} \quad \texttt{nnz(A)} \leqslant 10^6 \tag{10.1}$$

plus all test matrices in [48]. As in Part I of this note one goal of the tests is to check whether our methods Sparse1 and Sparse2 work up to the limit condition number $\kappa(A) \lesssim \mathbf{u}^{-1} \approx 10^{16}$. Since both methods rely on a lower bound of the smallest singular value of A, the 2-norm condition number $\kappa_2(A) = \|A^{-1}\|_2\|A\|_2$ is appropriate rather than `condest(A)` which estimates the 1-norm condition number.

After removing diagonal matrices in (10.1) that resulted in totally 400 real, as in Part I, plus 3 complex tests, the latter comprising of 1 spd and 2 general matrices. As in Part I we consider in addition 100 random test cases with matrices of dimension $n = 10^4$, density 0.001 and condition number about 10^{15}, i.e., generated by

$$\texttt{A} = \texttt{sprand(n,n,dens,1/cnd)} \quad \text{with } \texttt{n} = 10^4,\ \texttt{dens} = 0.001 \text{ and } \texttt{cnd} = \texttt{1e15}. \tag{10.2}$$

For least squares and underdetermined systems we tested all matrices from the Suite Sparse Matrix Collection [5] with

$$10^3 \leqslant \max(m,n) \leqslant 10^5 \quad \text{and} \quad 10^7 \leqslant \kappa_2(A) \leqslant 10^{16} \quad \text{and} \quad \texttt{nnz(A)} \leqslant 10^6 \ .$$

This lead to 29 test cases, two of them least squares problems. In order to increase the number of tests, we used the original samples and tested in addition A^T leading to two underdetermined and 27 least squares problems.

```
tic     % compute inclusion by Sparse1 or Sparse2 with equilibration
equil = true; X = Inclusion(A,b,equil);
T = toc;
if isnan(X) % verification with equilibration failed
  tic      % compute inclusion by Sparse1 or Sparse2 without equilibration
  equil = false; X = Inclusion(A,b,equil);
  if ~isnan(X)
    T = T + toc;
  end
end
```

Table 12
Timing T and inclusion X for algorithms Sparse1 and Sparse2

Concerning equilibration and computing time we use the same method for Algorithms Sparse1 and Sparse2 as in Part I of this note, here repeated in Table 12. Using equilibration is the method of choice because in most cases the condition number decreases, often substantially. Since there are rare cases where the condition number increases for the equilibrated matrix and our main target is a verified inclusion of the solution, we try the inclusion algorithms without equilibration if it failed with, and use for both algorithms the method as in Table 12 for timing and inclusion.

A summary of results is displayed in Table 13. The first column indicates the structure, namely spd, symmetric indefinite, general real, all test matrices out of [48], complex Hermitian positive definite and general complex, the random tests according to (10.2) and the rectangular tests with original and transposed matrix.

The random test cases in Part I and Part II of this note are the same. Accidentally, one random test failed when using Algorithm `verifySparselss1` in Part I with the mp-toolbox but Algorithm `verifySparselss1`, i.e., Sparse1 in Part II using `spProdK` succeeded. That may happen due to the random starting vector for computing an approximation of the smallest singular value and slightly differing accuracy of the mp-toolbox and `spProdK`.

As of Table 13, Sparse1 computed verified bounds in 398 out of the 400 real test cases, in 2 out of 3 complex tests, in 99 out of 100 random tests and 28 out of 29 rectangular tests each. Algorithm Sparse2 never failed in all tests.

Table 13
Results for all real and complex samples satisfying (10.1) *or* (10.2)

	performance backslash				performance Sparse1				performance Sparse2			
structure	poor	fail			poor	fail			poor	fail		
real spd	2	0	of	35	0	0	of	35	0	0	of	35
sym	2	0	of	96	0	0	of	96	0	0	of	96
gen	1	0	of	249	0	2	of	249	0	0	of	249
tests [48]	0	2	of	20	0	0	of	20	0	0	of	20
complex spd	1	0	of	1	0	1	of	1	0	0	of	1
gen	0	0	of	2	0	0	of	2	0	0	of	2
random	35	0	of	100	0	1	of	100	0	0	of	100
rect. orig	1	0	of	29	0	1	of	29	0	0	of	29
rect. transp	1	0	of	29	0	1	of	29	0	0	of	29

We begin with some numerical results on equilibration. In (3.1), (3.2) and (3.3) of Part I of this note we introduced our equilibration schemes for s.p.d., general symmetric and general matrices, respectively. We use the same in Part II. We mentioned in Part I that, in rare cases, equilibration may be counterproductive, i.e., may increase the condition number. We tested the effect of equilibration for all 403 test cases and display the ratio of the condition number with and without equilibration in Figure 2.

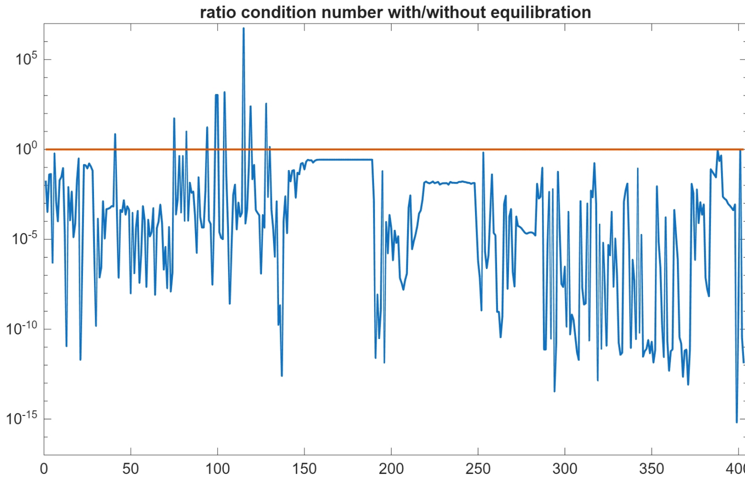

Fig. 2. *Ratios of condition numbers with and without equilibration*

A ratio less than 1 indicates an improvement by equilibration. As can be seen mostly equilibration is beneficial, but in 11 out of the 403 tests the condition number was worse with equilibration, in 8 cases by more than a factor 10. In the worst case, namely number 2536 in [5], the condition number increased from $5.9 \cdot 10^{14}$ to $3.2 \cdot 10^{21}$. That matrix is symmetric and the approximation s of the smallest singular value was so small that Algorithms `verifySparseNearSym2` called `verifySparseGen2` in Step 6. Hence an unsymmetric equilibration is applied, and this drops the condition number from $5.9 \cdot 10^{14}$ to $2.0 \cdot 10^{11}$.

Conversely, equilibration improves the condition number often significantly, in the median over all examples by a factor $3.4 \cdot 10^{3}$. An extreme case is number 1417 in [5] where equilibration dropped the condition number from $5.1 \cdot 10^{22}$ to $3.3 \cdot 10^{7}$. For that matrix Matlab's backslash failed with "out of memory".

Table 14
Nearly symmetric general matrices treated as symmetric

	time [sec]			median rel. error		maximum rel. error	
#	gen	nearsym	ratio	gen	nearsym	gen	nearsym
1404	0.19	0.11	1.71	1.5e-16	1.5e-16	2.2e-16	2.2e-16
930	21.12	0.78	27.06	1.5e-16	1.5e-16	2.2e-16	2.2e-16
1417	18.29	23.27	0.79	1.5e-16	1.5e-16	7.7e-15	5.2e-14

There were 3 examples of nearly symmetric matrices solved by "verifySparseSPD2" or "verifySparseNearSym2" because `nearsym` in Line 22 of `verifySparse-`

`lss2` in Table 11 was `true`. Without the techniques of treating unsymmetric matrices presented in Section 3, Algorithm "verifySparseGen2" would have been called. Timing, median and maximum relative errors of the "nearly symmetric approach" are listed in Table 14, where "gen" corresponds to "verifySparseGen2" and "nearsym" to "verifylssSparse2" which switches automatically to the symmetric case. In the first example number 1404 in [5] there are diagonal entries with different sign, so "verifySparseNearSym2" was called and ended successfully in a little more than half the computing time. In the second example number 930 "verifySparseSPD2" was successful and dropped the computing time by more than a magnitude without changing the quality of the inclusion.

For the third example number 1417 we just mentioned that equilibration dropped the condition number from $5.1 \cdot 10^{22}$ to $3.3 \cdot 10^{7}$. However, that was for the general case with unsymmetric equilibration. The example is also nearly symmetric and, because there are diagonal entries with opposite sign, "verifySparseNearSym2" is called but fails because the condition number increases from $5.1 \cdot 10^{22}$ to $3.4 \cdot 10^{28}$. As a consequence, "verifySparseGen2" is called and treating the matrix as a nearly symmetric matrix is counterproductive in terms of computing time.

Next we continue with the test results for our Algorithms Sparse1 and Sparse2. The dimension, number of nonzero elements and condition number of all 403 test cases is shown in Figure 3. The dimensions vary between $1,019$ and $682,862$ and the number of nonzero elements between $3,699$ and $5,778,545$. For given matrix of dimension n we generate a right hand side `A*(2*rand(n,1)-1))` as in Part I of this note. Hence the solution has, up to rounding errors, uniformly distributed entries between -1 and 1.

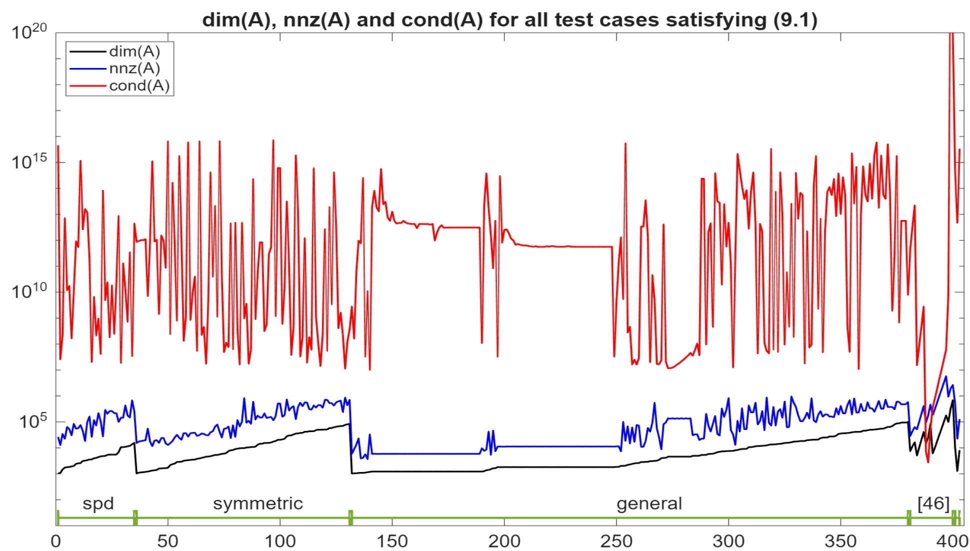

FIG. 3. *Dimension, number of nonzero elements and condition number of all test matrices*

In Figure 4 we show for all tests the ratio of computing times of Matlab's backslash divided by Sparse1 on the left, and Algorithm Sparse2 divided by that of Algorithm Sparse1 on the right. Ratios are displayed if backslash and Sparse1 (and therefore also Sparse2) are successful. That explains the gap at case 30 and $399 \ldots 401$.

A number less than 1 means that backslash is faster than Sparse1 and Sparse2

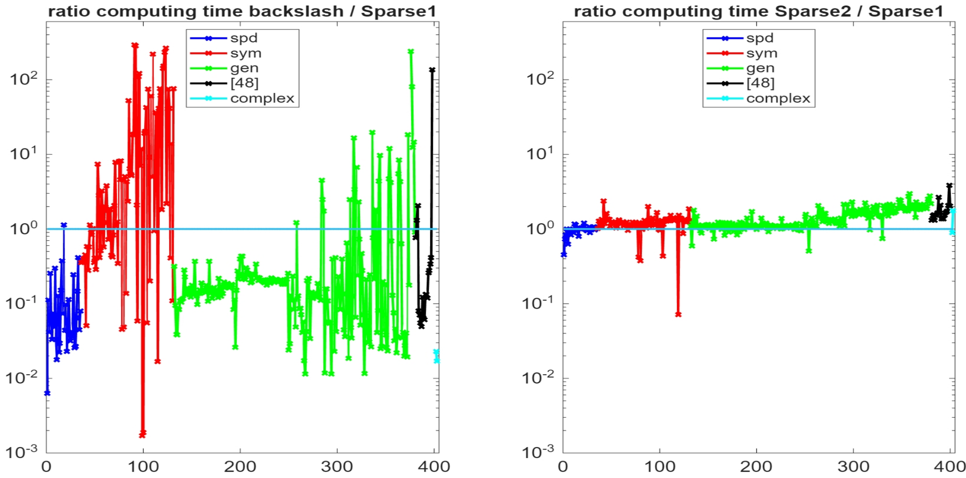

FIG. 4. *Ratios of computing times Matlab's backslash,* `verifySparselss1` *and* `verifySparselss2`

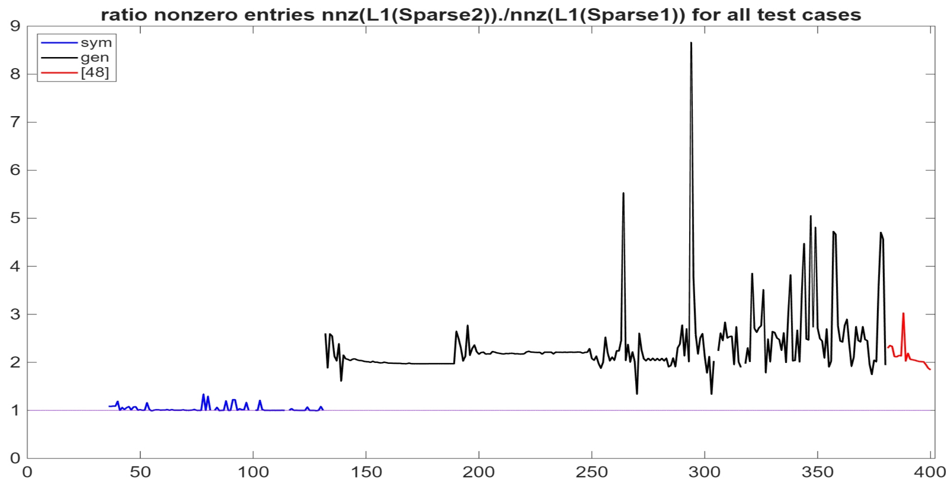

FIG. 5. *Ratio of number of nonzero entries of L_1 in Sparse2 divided by that of Sparse1*

is faster than Sparse1, respectively. In the median backslash is 5.2 times faster than Sparse1, and Sparse1 1.2 times faster than Sparse2. Conversely, Sparse1 is up to 584 times faster than backslash, and Sparse2 up to 14.1 times faster than Sparse1. For spd tests the times for Sparse1 and Sparse2 should basically coincide. The small deviation from the ratio 1 is due to uncertainties in the timing of Matlab, in particular for matrices with small dimension.

We may ask why Sparse2 is often a little slower than Sparse1. One reason is that the factorization of a shifted matrix as in Sparse2 causes significantly more fill-in than without shift as in Sparse1. That was also one reason why `verifylssSparse1` in Part I of this note was mostly faster than the algorithm in [48]. In Figure 5 we display the ratio of the number of nonzero entries of the factor L_1 in Sparse2 divided by that in Sparse1. A value greater than 1 means that Sparse2 produces more fill-in than Sparse1. The median ratio of fill-in over all examples is 2.0, and maximally L_1 in Sparse2 has 8.7 times more elements than L_1 in Sparse1.

Next we show in Figure 6 a rough image of the median relative error of Matlab's backslash and Algorithms Sparse1 and Sparse2. As can be seen, backslash produces

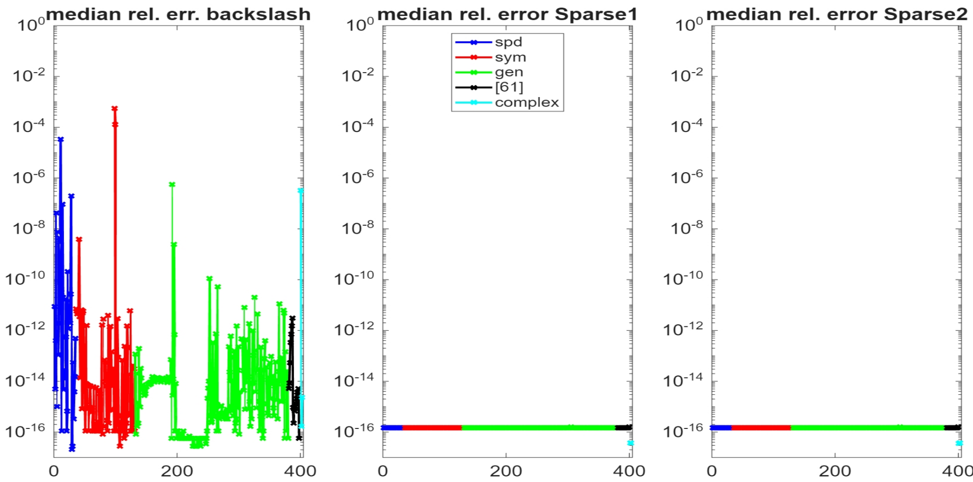

FIG. 6. *Median of relative errors of backslash,* `verifySparselss1` *and* `verifySparselss2`

in the median approximations with some 15 correct figures, whereas in the median all inclusions computed by Sparse1 and Sparse2 are maximally accurate. We may ask for the maximal relative error as well. The results are displayed in Figure 7.

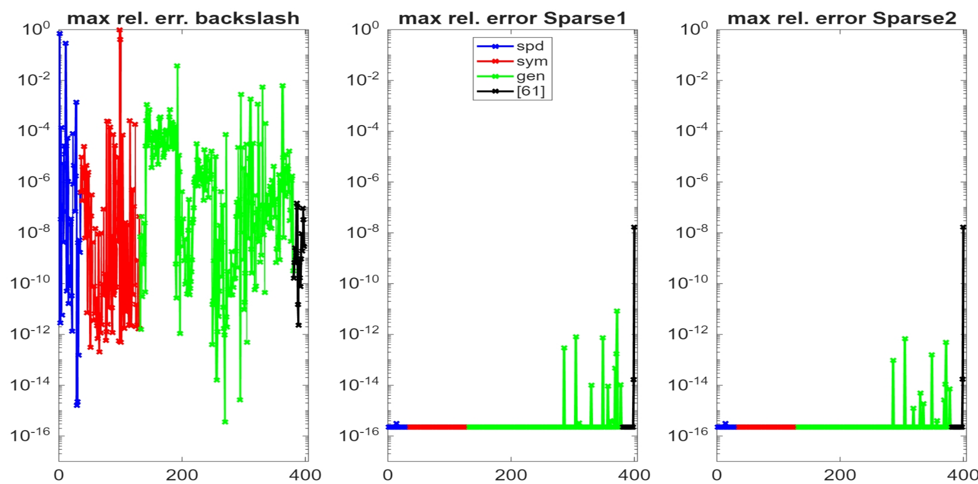

FIG. 7. *Maximum of relative errors of backslash,* `verifySparselss1` *and* `verifySparselss2`

As can be seen usually some entries of the approximation by "backslash" have some 8 correct digits or less. Having only the approximation at hand that information is not available. The verified inclusions are mostly maximally accurate for all entries, in few examples less. The reason is that we used the residual iteration in Algorithm `ErrorBound3` in Table 4 which stores the approximation in 3 parts. That requires the computation of residuals in higher than extended precision and is therefore out of the scope of Advanpix [12]. In Part II we use INTLAB's `spProdK` which offers that possibility.

We discuss some details of our Algorithm Sparse2 on the several improvement steps in the subalgorithms `verifySparseNearSym2` and `verifySparseGen2`. As has been mentioned, our first priority is the successful computation of verified bounds, and to that end there are several measures in the subalgorithms to avoid failure. Secondly, we aim to compute highly accurate bounds. One might introduce options to change these priorities.

We omit to display the behaviour of subalgorithm `verifySparseSPD2` because it is basically the same `verifySparseSPD1`. Next subalgorithm `verifySparseNearSym2`, results shown in Table 15. Note that the time consuming step 26 was not called. The security measure on singular D in Step 5 is no longer needed from Matlab's Version 2026a on.

TABLE 15
Detailed analysis of "verifySparseNearSym2' for the 96 *sym test cases satisfying* (10.1)

action	step	number of times
s too small	8	1
improve α_- first time	13	4
improve α_- second time	15	0
decrease s	16	0
improve α first time	21	4
improve α second time	23	0
verification failed, call verifySparseGen2	26	0

TABLE 16
Detailed analysis of "verifySparseGen2' for the gen test cases satisfying (10.1) *and in [48]*

action	step	number of times
improve α first time	12	63
improve α second time	13	1
decrease s	15	2
verification failed	16	0

As in Part I of this note in 1 out of the 96 samples the anticipated lower bound s for $\sigma_{\min}(A)$ was so small that there was no hope to compute an inclusion successfully. In those cases "verifySparseGen2" was called in Step 8 and computed an inclusion successfully. The potential first improvement of α in Line 13 was used 4 times. Neither the costly second improvement in Line 15 was necessary nor the decrease of s in Step 16. Otherwise the initial estimate on α was mostly sufficiently accurate, and the verification never failed in Step 26, i.e., it was not necessary to call subalgorithm verifySparseGen2". In particular the time consuming steps 15 and 23 were not necessary.

Secondly, some details on the performance of subalgorithm `verifySparseGen2` for the 249 gen test cases plus the 20 tests from [48] and the 3 complex test cases. The improvement of α in Step 12 was called in 63 cases, and the second improvement in Line 13 had to be used once in the 269 tests. The decrease of s in Step 15 was necessary 2 times. Algorithm Sparse2 never failed in all test cases.

We present some selected computational results in Tables 18 - 20; all results are in [50]. A NaN in the columns for the relative error indicates failure of verification, the last column displays the ratio $\varrho = t_{Sparse2}/t_{Sparse1}$. A ratio $\varrho > 1$ indicates that Algorithm Sparse1 is faster than Sparse2. Otherwise, the columns are self-explaining.

In order to reduce space for the results to be displayed in this note, we select all initial 403 tests in (10.1) according to Table 17. That fills just 3 pages. Note that

TABLE 17
Displayed tests extracted from the 403 tests in Table 13

- all tests where Matlab's "backslash", Sparse1 or Sparse2 failed
- all tests where the median relative error by "backslash" is larger than 10^{-4}
- all tests where the maximal relative error by "backslash" is larger than 0.5
- all tests where the maximal relative error by Sparse1 is larger than 10^{-14}
- all tests where the maximal relative error by Sparse2 is larger than 10^{-14}
- all tests where one of the ratios $t_{Sparse1}/t_\backslash$ or $t_\backslash/t_{Sparse1}$ is larger than 26.4
- all tests where one of the ratios $t_{Sparse2}/t_\backslash$ or $t_\backslash/t_{Sparse2}$ is larger than 52.8

samples where the *median* relative error by "backslash" is larger than 10^{-4} as well as the *maximum* relative error by Sparse1 or Sparse2 is larger than 10^{-14} are printed. Furthermore, the timing where "backslash" failed or is 26.4 time slower than Sparse1 or is 52.8 time slower than Sparse2 are printed in bold-face. Algorithm Sparse2 never failed, and the ratio of $t_{Sparse2}/t_{Sparse1}$ is only printed when Sparse1 does not fail. For the symmetric test cases it occurred 3 times that Algorithms "verifySparseNearSym1/2" failed and Sparse1 and Sparse2 had to call "verifySparseGen1/2", respectively. The corresponding times for number $1210, 1451$ and 2536 are appended with a †. In the first two cases the sign of some entries of the approximation by "backslash" is incorrect.

TABLE 18
Timing and accuracy for sparse linear systems in [5] satisfying the conditions in (10.1) $[\ \varrho := t_{Sparse2}/t_{Sparse1}\ ]$

		matrix			*times* [*sec*]			*relerr backslash*		*relerr Sparse1*		*relerr Sparse2*		
# *matrix*		n	$nnz(A)$	$\kappa_2(A)$	$t_{backslash}$	$t_{Sparse1}$	$t_{Sparse2}$	*median*	*max*	*median*	*max*	*median*	*max*	ϱ
spd	358	1050	26198	4.6e15	0.00	0.32	0.15	8.8e-12	0.691	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.45
	427	1821	52685	1.7e10	0.00	0.09	0.08	7.5e-9	5.3e-6	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.90
	408	2548	57308	5.5e11	0.00	0.09	0.08	2.9e-9	1.3e-5	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.00
	411	2568	75628	1.2e15	0.00	0.21	0.19	3.4e-5	0.293	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.90
	1623	3200	18316	1.6e13	0.00	0.02	0.03	1.1e-16	5.0e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.14
	440	3363	99471	1.2e13	0.00	0.16	0.14	9.3e-8	5.4e-5	1.5e-16	3.2e-16	1.5e-16	3.2e-16	0.93
	1625	4800	27520	8.2e13	0.00	0.03	0.03	1.1e-16	3.3e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.06
	1606	5489	263351	1.8e8	0.02	0.43	0.41	1.2e-12	8.1e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.96
	1607	5489	262943	1.8e10	0.02	0.55	0.52	1.5e-11	4.6e-6	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.94
	1610	5489	217669	2.5e10	0.02	0.42	0.46	2.7e-11	1.8e-6	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.08
	1406	10605	144579	1.9e7	0.00	0.14	0.14	2.1e-17	1.6e-15	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.99
	1409	10605	144579	1.3e11	0.00	0.14	0.14	2.8e-17	2.3e-15	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.01
sym	2693	10260	92703	3.5e9	**12.09**	0.23	0.26	6.6e-15	6.2e-9	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.16
	2214	14454	147972	8.3e11	**151.27**	0.51	0.69	1.4e-12	2.8e-5	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.34
	2231	14454	147972	8.3e11	**151.19**	0.53	0.66	1.5e-12	9.7e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.25
	2694	15561	149532	3.0e11	**48.90**	0.45	0.50	6.4e-15	4.3e-9	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.12
	2695	16011	155246	5.4e11	**56.81**	0.47	0.53	2.7e-14	3.5e-8	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.14
	1210	20360	509866	6.1e14	0.22	$129.45^{\dagger}$	$127.60^{\dagger}$	5.5e-4	1.000	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.99
	1451	20360	509866	6.1e14	0.26	$139.03^{\dagger}$	$133.63^{\dagger}$	1.3e-4	0.402	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.96
	2229	28216	730080	3.1e13	**883.85**	20.82	9.00	2.9e-12	7.0e-5	1.5e-16	2.2e-16	1.5e-16	2.2e-16	0.43
	1572	30235	355139	3.8e7	**95.70**	1.27	1.63	2.2e-16	6.8e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.28
	1574	35910	380240	1.2e13	**98.66**	1.62	2.23	1.5e-16	2.6e-8	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.37
	1575	37833	403373	1.8e8	**250.77**	1.14	1.49	4.4e-16	3.0e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.30
	1577	43618	310016	1.8e8	**85.70**	2.37	3.31	1.1e-16	1.7e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.39
	1576	43640	298570	6.1e7	**104.28**	2.97	4.08	5.6e-17	2.4e-12	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.37
	2536	43887	426898	5.9e14	0.06	$3.76^{\dagger}$	$4.88^{\dagger}$	2.4e-13	2.6e-4	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.30
	1578	48066	360428	1.1e9	**74.82**	1.82	2.49	1.1e-16	1.6e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.37

TABLE 19
Timing and accuracy for sparse linear systems in [5] satisfying the conditions in (10.1) $\left[\ \varrho := t_{Sparse2}/t_{Sparse1}\ \right]$

	matrix			*times [sec]*			*relerr backslash*		*relerr Sparse1*		*relerr Sparse2*		
# *matrix*	n	$nnz(A)$	$\kappa_2(A)$	$t_{backslash}$	$t_{Sparse1}$	$t_{Sparse2}$	*median*	*max*	*median*	*max*	*median*	*max*	ϱ
951	49989	444853	1.1e8	**178.46**	2.33	3.16	8.3e-17	2.2e-12	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.35
1588	49989	444851	1.4e7	**179.78**	2.94	4.21	2.2e-16	3.9e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.43
950	51035	707985	2.7e12	**389.68**	2.73	4.19	2.1e-13	5.1e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.54
1587	51035	707601	5.5e7	**390.14**	2.57	3.63	5.6e-16	3.7e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.41
952	57975	530229	1.9e13	**483.97**	2.07	2.86	1.9e-14	1.1e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.38
1589	57975	530583	3.4e9	**492.66**	1.86	2.48	4.4e-16	8.6e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.34
1225	64810	565996	3.2e12	**697.21**	17.10	22.47	1.1e-16	2.0e-12	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.31
1226	67458	623914	4.1e8	**625.15**	8.31	11.17	1.1e-16	1.5e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.34
1227	68924	658986	1.6e9	**658.79**	16.04	19.83	1.1e-16	2.4e-12	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.24
1229	84064	707546	3.6e8	**1049.17**	13.83	17.93	1.1e-16	1.7e-12	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.30
gen 438	1633	46626	8.3e10	0.01	0.45	0.78	2.5e-9	1.1e-5	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.71
258	1856	11360	4.4e9	0.01	0.25	0.36	2.2e-15	1.5e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.47
259	1856	11550	1.4e11	0.01	0.24	0.38	5.3e-15	2.1e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.57
439	3096	90841	4.6e10	0.02	1.33	2.36	5.3e-11	1.2e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.77
235	3140	543160	4.8e9	0.73	64.29	118.26	1.1e-16	1.2e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.84
839	4134	94408	2.1e7	0.02	0.49	0.68	1.1e-15	5.0e-12	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.37
157	4929	33044	1.0e8	**0.29**	0.17	0.19	1.0e-14	1.7e-10	1.5e-16	3.0e-13	1.5e-16	9.6e-14	1.09
322	5000	19996	3.7e7	0.00	0.06	0.08	5.9e-13	1.9e-9	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.41
818	6316	167178	3.9e14	0.01	1.27	2.74	1.1e-16	2.7e-15	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.16
934	7055	30082	2.8e13	0.02	0.82	1.89	3.3e-16	0.003	NaN	NaN	1.5e-16	2.2e-16	
446	7320	324772	2.0e10	0.11	4.92	8.79	1.5e-12	1.0e-8	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.79
920	7500	283992	5.7e11	0.46	11.07	25.64	2.7e-14	8.8e-10	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.32
1404	8765	42471	4.9e14	0.04	0.11	0.19	0.0e0	5.0e-13	1.6e-16	8.1e-13	1.6e-16	6.8e-13	1.75
581	9129	52883	8.4e13	0.09	2.55	4.11	4.4e-15	1.6e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.61
921	11532	551184	4.3e12	1.45	78.97	177.03	1.4e-13	2.3e-9	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.24
415	12005	259577	2.2e12	0.12	3.50	4.83	8.3e-16	3.3e-5	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.38
922	16428	948696	2.6e13	3.78	326.24	489.12	1.7e-13	1.2e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.50

TABLE 20
Timing and accuracy for sparse linear systems in [5] satisfying the conditions in (10.1) $\left[\ \varrho := t_{Sparse2}/t_{Sparse1}\ \right]$

	matrix			times [sec]			relerr backslash		relerr Sparse1		relerr Sparse2		
# matrix	n	$nnz(A)$	$\kappa_2(A)$	$t_{backslash}$	$t_{Sparse1}$	$t_{Sparse2}$	median	max	median	max	median	max	ϱ
582	18289	106803	6.8e13	0.21	6.88	12.98	3.1e-14	8.4e-9	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.89
583	18289	106803	5.5e13	0.20	6.54	12.49	1.6e-15	1.4e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.91
431	19716	227872	2.2e12	0.13	4.77	9.95	4.2e-16	2.0e-4	1.5e-16	2.2e-16	1.5e-16	1.9e-15	2.09
584	27449	160723	1.1e14	0.31	12.46	23.62	3.7e-14	1.5e-8	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.90
585	27449	160723	5.5e13	0.32	11.98	22.61	1.3e-15	1.2e-6	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.89
576	34454	190224	3.8e14	1.21	6.73	16.35	4.4e-16	4.2e-6	1.5e-16	7.4e-13	1.5e-16	1.6e-13	2.43
559	35550	412306	1.0e13	1.50	39.20	80.12	1.3e-14	1.3e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.04
586	36609	214643	2.7e14	0.47	18.25	35.02	5.5e-14	4.0e-8	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.92
587	36609	214643	5.6e13	0.42	17.98	33.90	2.1e-15	2.2e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.89
561	41490	488633	9.6e12	1.92	60.54	124.21	8.2e-15	9.4e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.05
589	45769	268563	5.6e13	0.53	24.12	44.41	1.9e-15	4.3e-7	1.5e-16	4.0e-16	1.5e-16	2.2e-16	1.84
563	47430	564952	3.1e13	2.21	60.98	128.36	1.1e-14	6.3e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.11
565	53370	641290	1.8e13	2.63	76.64	158.15	1.1e-14	8.9e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.06
591	54929	322483	5.6e13	0.63	31.88	58.09	1.6e-15	1.2e-5	1.5e-16	4.8e-14	1.5e-16	2.6e-15	1.82
567	59310	717620	4.0e13	2.90	105.82	223.45	1.1e-14	8.6e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.11
592	64089	376395	5.0e15	1.55	38.47	72.98	8.4e-14	4.5e-6	1.5e-16	1.7e-13	1.5e-16	1.1e-14	1.90
593	64089	376395	5.6e13	0.74	38.65	72.19	1.7e-15	3.2e-6	1.5e-16	8.4e-12	1.5e-16	4.8e-13	1.87
1374	87190	606489	1.8e15	**385.90**	10.47	14.01	5.6e-17	3.0e-8	NaN	NaN	1.5e-16	2.2e-16	
2657	87936	593276	6.7e8	**1850.68**	7.72	14.48	1.4e-13	7.9e-8	1.5e-16	2.2e-16	1.5e-16	2.2e-16	1.88
1343	94294	476766	5.6e12	**442.39**	5.49	13.85	1.1e-16	4.7e-9	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.53
1344	94294	479246	5.6e12	**82.65**	6.70	15.98	1.4e-14	1.1e-6	1.5e-16	1.0e-14	1.5e-16	7.2e-15	2.38
[48] 1415	99340	940621	7.9e9	**1324.81**	9.78	20.23	5.6e-17	3.0e-9	1.5e-16	2.2e-16	1.5e-16	2.2e-16	2.07
1417	321821	1931828	5.1e22	**memory**	23.65	91.25			1.5e-16	1.7e-14	1.5e-16	1.7e-14	3.86
1419	682862	2638997	8.1e19	**crash**	316.30	640.49			1.6e-16	1.7e-8	1.6e-16	1.7e-8	2.02
cmplx 1407	10605	522387	5.5e14	9.19	21.72	46.95	3.2e-7	0.509	NaN	NaN	3.6e-17	1.1e-16	
1621	1280	22778	4.7e12	0.00	0.08	0.07	1.7e-16	2.8e-12	3.7e-17	1.1e-16	3.7e-17	1.1e-16	0.90
792	8184	127762	3.4e15	0.20	12.04	21.37	2.3e-15	1.1e-11	3.8e-17	1.1e-16	3.8e-17	1.1e-16	1.78

As in Part I of this note we give some additional test results for randomly generated ill-conditioned sparse matrices using `A = sprand(n,n,dens,1/cnd)` with dimension $n = 10^4$, density 0.001 and `cnd=1e15`. The resulting matrices have some 100,000 nonzero elements each, and the median 2-norm condition number over the 100 tests was $1.9 \cdot 10^{15}$. The results of this test are reported in Table 21.

TABLE 21
Results for 100 randomly generated ill-conditioned test cases

	Sparse1	Sparse2
inclusions	failed in 1 out of 100 tests	failed in 0 out of 100 tests
median relative error	1.5e-16	1.5e-16
maximal relative error	1.5e-16	1.5e-16

The median condition number $1.9 \cdot 10^{15}$ of our samples is boarder line in the sense that a verification algorithm might just succeed to compute verified bounds. Still, Sparse1 succeeds in 99 cases, and Sparse2 in all cases to compute maximally accurate bounds for all entries. For randomly generated examples there is not much difference in the accuracy of the bounds, but Sparse1 is often twice as fast as Sparse2. In Figure 8 we show the ratio of computing times of Algorithm Sparse2 divided by that of Sparse1. As explained before that is related to the number of nonzero elements of the

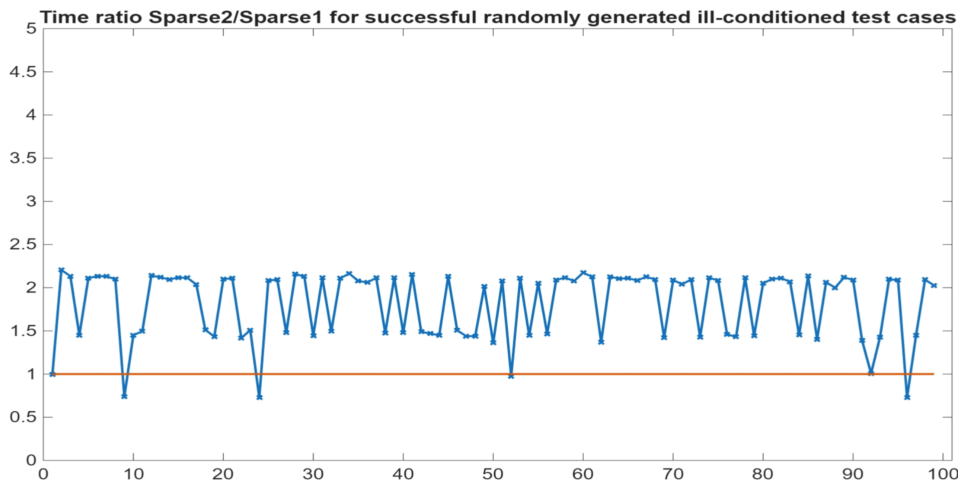

FIG. 8. *Ratios of computing times* $t_{\texttt{verifySparselss2}}/t_{\texttt{verifySparselss1}}$

matrices L_1, which is almost always 2.5 times as much for Sparse2.

We tested Algorithms Sparse1 and Sparse2 for complex data as well. All data is shown in [50]. There is one failure of Sparse1, namely the spd matrix number 1407 in [5]. The sum $\alpha + \beta + \gamma$ of the error terms in Step 26 of Algorithm "verifySparseGen1" is $2.39 \cdot 10^{-15}$ and slightly larger than the anticipated lower bound $s = 2.03 \cdot 10^{-15}$ of the smallest singular value. As there were no other surprises we refrain from showing computational data here but put it into [50].

Next we show computational results for rectangular input matrix. Our algorithms verify an inclusion of $A^{+}b$ which is computed in Matlab by the backslash operator for least squares problems and `lsqminnorm` for underdetermined systems. As has been

mentioned, we tested all matrices from the Suite Sparse Matrix Collection [5] with

$$10^3 \leqslant \max(m,n) \leqslant 10^5 \quad \text{and} \quad 10^7 \leqslant cnd \leqslant 10^{16} \quad \text{and} \quad \texttt{nnz(A)} \leqslant 10^6 \ . \tag{10.3}$$

The condition number of a rectangular matrix with respect to a least squares problem is a bit tricky. Here $\kappa_2(A)$ denotes the 2-norm condition number of the augmented matrix in (7.1) which coincides with that of the matrix in (7.2).

There were no complex examples in [5] satisfying (10.3). The conditions in (10.3) lead to 29 test cases, two of them least squares problems. In order to increase the number of tests, we used the original samples and tested in addition A^T leading to two underdetermined and 27 least squares problems. All result are listed in [50].

Here we display some selected results in one Table 22, for the original matrices in the upper half and for A^T below. The timing for “backslash” and `lsqminnorm` is printed bold-face if slower than Sparse1 or Sparse2. Algorithm Sparse2 never failed, and Sparse1 in one case number 1950 in [5].

Table 22
Selected rectangular matrices in [5] satisfying the conditions in (10.1) with results for the original matrix in the upper half, for the transposed below

	matrix					times [sec]			relerr backslash		relerr Sparse1		relerr Sparse2	
type	# mat	m	n	$nnz(A)$	$\kappa_2(A)$	$t_{backslash}$	$t_{Sparse1}$	$t_{Sparse2}$	median	max	median	max	median	max
lsq	169	1033	320	4719	2.1e8	0.010	0.059	0.049	1.4e-15	1.7e-12	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	981	29493	11822	117954	9.5e8	**20.150**	6.957	3.171	7.3e-12	3.7e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16
under	1948	2644	3160	29862	6.3e7	0.191	5.339	7.276	5.1e-14	1.3e-10	1.6e-16	2.2e-16	1.6e-16	2.2e-16
	1949	6334	7742	80057	4.0e8	2.961	239.717	186.615	2.7e-13	2.3e-9	1.6e-16	2.2e-16	1.6e-16	2.2e-16
	1731	4400	16819	150372	1.4e7	0.111	1.347	1.735	7.8e-16	1.2e-10	1.6e-16	2.2e-16	1.6e-16	2.2e-16
	1950	15437	19321	216173	2.7e9	45.184	7807.761	1772.134	3.3e-13	8.4e-9	NaN	NaN	1.6e-16	2.2e-16
	1834	14364	27691	58439	3.9e7	**6.719**	0.608	0.545	3.6e-16	4.3e-12	1.6e-16	2.2e-16	1.6e-16	2.2e-16
	1708	24617	38602	156466	1.3e9	51.435	412.024	938.336	1.0e-15	9.6e-8	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1829	32847	46679	120141	8.3e7	**1.300**	1.253	1.238	4.4e-16	5.4e-11	1.6e-16	2.2e-16	1.6e-16	2.2e-16
	1779	6590	46937	164538	1.5e11	0.137	1.274	1.511	5.6e-17	7.5e-14	1.6e-16	2.2e-16	1.6e-16	2.2e-16
	1775	3173	63076	491336	1.0e7	0.065	1.786	2.091	2.1e-16	1.1e-11	1.6e-16	2.2e-16	1.6e-16	2.2e-16
under	169	320	1033	4719	2.1e8	0.004	0.036	0.033	1.4e-14	5.1e-12	1.6e-16	2.2e-16	1.6e-16	2.2e-16
	981	11822	29493	117954	9.5e8	0.039	6.609	2.564	7.9e-13	3.1e-8	1.6e-16	2.2e-16	1.6e-16	2.2e-16
lsq	1948	3160	2644	29862	6.3e7	0.109	4.884	6.232	2.1e-13	5.5e-10	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1949	7742	6334	80057	4.0e8	0.959	232.729	171.551	2.1e-13	8.9e-10	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1713	16369	10099	44825	1.2e10	**8.053**	0.698	0.837	5.1e-15	6.9e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1731	16819	4400	150372	1.4e7	0.270	1.679	1.996	8.3e-14	2.0e-9	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1950	19321	15437	216173	2.7e9	14.861	7683.927	1865.704	1.3e-13	3.5e-9	NaN	NaN	1.5e-16	2.2e-16
	1834	27691	14364	58439	3.9e7	**163.291**	0.667	0.628	7.8e-16	2.3e-11	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1708	38602	24617	156466	1.3e9	156.976	425.328	514.906	1.1e-13	4.8e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1829	46679	32847	120141	8.3e7	**317.512**	1.383	1.465	8.3e-16	6.8e-12	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1779	46937	6590	164538	1.5e11	**47.858**	1.628	1.824	7.7e-12	2.5e-7	1.5e-16	2.2e-16	1.5e-16	2.2e-16
	1775	63076	3173	491336	1.0e7	**18.462**	3.027	3.379	2.1e-15	1.0e-10	1.5e-16	2.2e-16	1.5e-16	2.2e-16

Except for number 1950 there is not too much difference in computing times for Sparse1 and Sparse2. For case 1950 Sparse1 needed more than 4 times the time of Sparse2 to finally fail, where Sparse2 succeeded to compute an inclusion. In the median the computing times for the remaining cases are almost the same, in the worst case Sparse1 is 2.6 times faster than Sparse2, and Sparse2 is 1.3 times faster than Sparse1.

A reason is that, in contrast to the square case, there is not much difference in the fill-in of the factor L_1 because half of the diagonal elements of the augmented matrix (7.1) are already nonzero.

There is quite a spread in computing time between backslash and our new algorithms, and surprisingly Matlab's backslash or `lsqminnorm` is often slower than our verification. In the median backslash is 6.4 times faster than Sparse1, in the worst case Sparse1 is 245 times faster than backslash. However, backslash may be also up to 243 times faster than Sparse1.

Both Algorithms Sparse1 and Sparse2 compute always inclusions with maximal accuracy for all entries of the solution. In contrast, the approximations by Matlab's backslash or `lsqminnorm` are less accurate. The median and maximum relative errors of the approximation by backslash or `lsqminnorm`, and by Sparse1 and Sparse2 are displayed in Figure 9. As can be seen, in the median some 13 figures of Matlab's approximation are correct, but in two cases no digit is correct of at least one entry of the approximation. In contrast, Sparse1 and Sparse2 compute always maximally accurate inclusions for all entries due to the improved residual iteration `ErrorBound3` in Table 4.

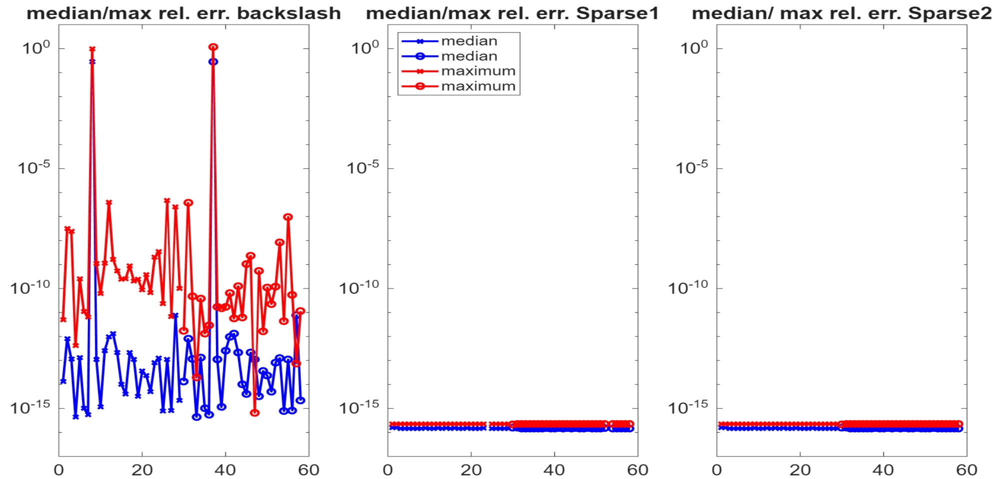

FIG. 9. *Median of relative errors of Matlab's backslash or* `lsqminnorm`*, Sparse1 and Sparse2*

Out of the ill-conditioned test cases satisfying (10.3), i.e., condition number exceeding 10^{16}, there were 37 matrices with zero rows or columns implying that the matrix is rank-deficient. When deleting those rows or columns there was a dichotomy. Either the matrices became well-conditioned, i.e., condition number less than $5 \cdot 10^7$, or, the matrices were still extremely ill-conditioned, i.e., condition number larger than $3 \cdot 10^{20}$. In the former case it was no problem to compute verified inclusions, the latter cases are out of the scope of verification methods. Therefore, we refrain from giving additional computational results for those.

We finally show some test results for systems of nonlinear equations. The first source of test examples stems from the MINPACK project [25]. The source code for 23 examples is available at

`https://people.sc.fsu.edu/~jburkardt/m_src/test_nonlin/test_nonlin.html`

In 4 examples the dimension can be freely specified. In the first example $p01$ the floating-point Newton iteration did not converge. For the other three examples $p09$, $p13$ and $p14$ we list computational results for different dimensions.

The results for Algorithm `verifySparseNlss` are shown in Table 23. We compare three algorithms. The first is `verifySparseNlss` listed in Table 10 calling Sparse1 in Step 23 with the modification as in Algorithm "verifySparselss1" as in Table 7 in Part I of this note to solve the linear system with interval right hand side. Secondly, we use Algorithm Sparse2 in Step 23 as the linear system solver. As a third algorithm we use the Matlab routine `fsolve` which is part of Matlab's Optimization toolbox. That comparison is biased because `fsolve` does not use derivatives.

All routines have as input parameters a reference to the function in use as well as the starting values specified in [25] as an initial approximation.

To further investigate the performance of our algorithms, we consider two other examples with specifiable dimension. Both use second order finite differences to discretize first an example in [24], abbreviated by MC,

$$MC: \qquad u'' = .5 * (u + t + 1)^3 \quad \text{with} \;\; u(0) = u(1) = 0$$

with initial approximation $x_k = t_k(t_k - 1)$ for $t_k = k/(n+1)$, and second an example in [1], abbreviated by AB,

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

with true solution $20x^{3/4}$. The initial approximation specified in [1] is `10*ones(n,1)`.

INTLAB contains Algorithm `verifynlss` for solving systems of nonlinear equations. It is based on Theorem 8.2 but using an approximate inverse R of the Jacobian at $\tilde{x}$ which is, in general, a full matrix. For dimension $n = 10^4$ that requires, for example, some 800 megabytes of memory. For small dimension, all of our 5 examples are solved successfully by `verifynlss`, but larger dimensions are prohibitive for our laptop. The same seems to apply to Matlab's `fsolve`.

Computational results are shown in Table 23. The columns are self-explaining except "iter" for Sparse1 and Sparse2 which displays the number `kxs` of approximate Newton iterates and the number `kY` of interval iterates in Algorithm "verifySparseNlss" in Table 10. For an inclusion vector $\mathbf{X} \in \mathbb{IR}^n$ the column $\|\cdot\|_2$ shows $\|\text{rad}(\mathbf{X})\|_2/\|\text{mid}(\mathbf{X})\|_2$, see below.

TABLE 23
Timing and accuracy for systems of nonlinear equations with sparse Jacobian

problem			*times [sec]*			*iter and relerr Sparse1*				*iter and relerr Sparse2*				*relerr* `fsolve`	
	n	*cond*	$t_{Sparse1}$	$t_{Sparse2}$	$t_{\texttt{fsolve}}$	*iter*	*median*	*max*	$\lVert\cdot\rVert_2$	*iter*	*median*	*max*	$\lVert\cdot\rVert_2$	*median*	*max*
p09	1,000	4.0e5	0.041	0.049	0.100	4/2	1.3e-8	6.8e-6	1.4e-8	4/2	1.3e-8	6.8e-6	1.4e-8	0.107	0.195
	10,000	4.0e7	0.146	0.146	3.18	4/3	1.3e-5	0.068	1.4e-5	4/3	1.3e-5	0.068	1.4e-5	0.200	0.364
	20,000	1.6e8	0.356	0.329		4/5	1.2e-4	0.606	1.2e-4	4/5	1.2e-4	0.604	1.2e-4	out of memory	
	30,000	3.6e8	0.557	0.547		4/6	NaN	NaN	NaN	4/6	NaN	NaN	NaN	out of memory	
	200,000	*1.6e10*	*3.63*	*3.61*		*5/5*	*2.5e-16*	*1.5e-13*	*3.5e-16*	*5/5*	*2.5e-16*	*1.5e-13*	*3.5e-16*	*out of memory*	
p13	1000	2.5	0.088	0.054	0.182	5/5	5.1e-13	1.3e-12	5.1e-13	5/2	4.0e-13	1.0e-12	4.0e-13	2.5e-13	6.3e-13
	10,000	2.5	0.206	0.166	19.6	5/3	5.2e-12	1.3e-11	5.2e-12	5/2	4.1e-12	1.0e-11	4.1e-12	2.9e-12	1.2e-11
	100,000	2.5	1.628	1.803		5/2	5.2e-11	1.3e-10	5.2e-11	5/2	4.1e-11	1.0e-10	4.1e-11	out of memory	
	1,000,000	2.5	18.21	20.06		5/2	5.2e-10	1.3e-9	5.2e-10	5/2	4.1e-10	1.0e-9	4.1e-10	out of memory	
	10,000,000	2.5	484.4	87.6		5/5	5.6e-9	1.4e-8	5.6e-9	5/2	NaN	NaN	NaN	out of memory	
p14	1,000	1.8	0.725	0.284	1.458	7/2	2.4e-13	5.0e-13	2.3e-13	7/3	2.4e-13	5.1e-13	2.4e-13	1.2e-13	3.5e-9
	10,000	1.8	0.617	0.726	128.0	6/2	6.2e-12	1.3e-11	6.2e-12	6/2	2.3e-12	4.9e-12	2.3e-12	1.2e-12	2.5e-12
	100,000	1.8	11.6	31.5		6/2	6.0e-11	1.3e-10	6.0e-11	6/2	5.9e-11	1.3e-10	5.9e-11	out of memory	
	1,000,000	1.8	207.9	3408		6/3	6.0e-10	1.3e-9	6.0e-10	6/3	6.0e-10	1.3e-9	6.0e-10	out of memory	
MC	1,000	4.0e5	0.080	0.076	0.182	4/3	4.8e-12	4.9e-6	4.9e-12	4/2	3.8e-12	3.9e-6	4.0e-12	6.8e-4	8.3e-4
	10,000	4.0e7	0.258	0.296	15.9	4/3	2.3e-9	0.23	2.3e-9	4/2	1.8e-9	0.18	1.8e-9	5.3e-4	0.13
	100,000	4.0e9	3.17	26.4		4/4	2.7e-7	0.014	2.8e-7	4/4	2.7e-7	0.014	2.8e-7	out of memory	
	1,000,000	4.0e11	31.5	2935		4/3	NaN	NaN	NaN	4/3	NaN	NaN	NaN	out of memory	
AB	1,000	5.3e5	0.157	0.116	0.587	12/3	3.9e-13	4.5e-11	3.7e-13	12/2	2.1e-13	2.3e-11	1.9e-13	1.0e-9	1.1e-7
	3,000	4.7e6	0.384	0.261	5.06	13/8	1.9e-12	9.7e-10	1.7e-12	13/4	7.4e-13	3.8e-10	6.9e-13	4.6e-8	1.2e-5
	4,000	8.4e6	0.322	0.482	9.97	13/5	NaN	NaN	NaN	13/7	1.2e-12	4.0e-10	1.2e-12	out of memory	
	5,000	1.3e7	0.246	0.506		13/3	NaN	NaN	NaN	13/6	NaN	NaN	NaN	out of memory	
	90,000	*4.3e9*	*3.59*	*16.1*		*18/5*	*2.5e-16*	*4.3e-16*	*3.5e-16*	*18/4*	*2.5e-16*	*3.4e-16*	*3.5e-16*	*out of memory*	

For example, for $p13$ algorithm Sparse1 works successfully for dimension 10^7 while Sparse2 fails. Conversely, for problem AB Sparse2 is successful for larger dimensions than Sparse1.

For the first example $p09$ and the last AB we show the effectiveness of improving the approximation $\tilde{x}$ in extended precision as explained at the end of Section 8 for $n = 200,000$ and $n = 90,000$, respectively. Only the approximation part in executed in extended precision (where we increased the maximum number of iterations), the verification part still uses double precision. The results are display in italic in Table 23. As can be seen, much larger dimensions are successfully treated and very accurate inclusions computed in a reasonable computing time.

Note that the median and maximal entrywise relative errors over all entries of the solution are displayed. Naturally, those become weak for entries small in absolute value. We may also judge an inclusion by the error $\delta \in \mathbb{R}_+$ of an approximation $\tilde{x}$ in the n-dimensional space, so that the inclusion is the ball $\{x \in \mathbb{R}^n : |x - \tilde{x}| \leqslant \delta\}$. The ratio $\delta/\|\tilde{x}\|_2$ is shown in the columns "$\|\cdot\|_2$" in Table 23. That ratio does not exceed about 10^{-4} for all examples.

A typical application of our sparse linear solver in the realm of verification methods is the following. The verification of solutions to nonlinear partial differential equations (PDEs) is typically carried out via fixed-point theorems such as the Newton–Kantorovich theorem. To validate the contraction property of the associated mapping and thereby ensure the existence of a fixed point, all relevant quantities must be explicitly evaluated and/or rigorously estimated. This includes an estimate of the discretization error arising from the projection onto a finite-dimensional subspace of the original infinite-dimensional function space. As a result, the existence of the continuous solution of the original PDE within explicitly computed error bounds to a certain approximate solution is proved. Some details on verified inclusions of ODEs and PDEs can be found, for example, in Part 3 of [43].

Xuefeng Liu [21] communicated that in the context of finite element methods (FEM) substantial progress has been made, and several verification schemes have been developed. Notably, a priori error estimates based on the hypercircle method have proven effective for bounding discretization errors, particularly in the rigorous treatment of the divergence-free condition in 3D domains [22]. Since the matrices generated by FEM are typically sparse, there is a strong need for verification algorithms that effectively exploit this sparsity structure. However, as for larger Reynolds numbers the current computational capabilities are still far from our target. From Liu's perspective, the efficiency of rigorous sparse linear solvers is one of the major bottlenecks in verified computation.

We close this note with an explicit example communicated by Xuefeng Liu [21] arising in this area, the verification of an eigenproblem of a three dimensional Navier-Stokes equation using mixed finite elements on a cube domain. The resulting sparse linear system had $30,424$ unknowns with $3,056,247$ nonzero elements, and in a finer discretization $247,956$ unknowns with $28,167,243$ nonzero elements, see Table 24.

For the smaller problem Matlab's "backslash" operator needed 95 seconds to compute an accurate approximation in the median and 3 correct digits for some entries. Our verification Algorithm Sparse1 produced verified bounds in 16 seconds with all entries almost maximally accurate. Some 52 entries of Matlab's approximation have incorrect sign, however, the size of those entries is below 10^{-16}. Algorithm Sparse2 produced verified bounds in 21 seconds with all entries almost maximally accurate as well.

For the larger problem Sparse1 needed 78 seconds and Sparse2 115 seconds to

compute inclusions with median relative error $1.5 \cdot 10^{-16}$. Both algorithm guarantee at least 11 correct digits. The built-in "backslash" operator in Matlab finished after some 12 hours with "out of memory".

Table 24
Detailed results for the sparse linear system arising in the verification of an eigenproblem of a three dimensional Navier-Stokes equation communicated by Xuefeng Liu [21]

	small problem			large problem		
	backslash	Sparse1	Sparse2	backslash	Sparse1	Sparse2
n		30,424			3,056,247	
nnz(A)		247,956			28,167,243	
$\text{cond}_2(A)$		8.1e4			5.7e5	
median rel. err.	5.6e-17	1.6e-16	1.6e-16	-	1.5e-16	1.5e-16
max rel. err.	0.004	2.2e-15	1.5e-15	-	2.3e-10	1.7e-10
time [sec]	95.3	16.4	21.2	>12 hours	78.4	115.3
time Gen [sec]		251.6	462.4		311.3	503.6
median rel. err.		1.6e-16	1.6e-16		1.5e-16	1.5e-16
max rel. err.		8.5e-15	1.8e-15		3.9e-11	9.8e-12
symmetric		no			no	
A^{sym} s.p.d.		no			no	
$\|A^T - A\|_\infty$		9.9e-17			1.1e-16	

For both Sparse1 and Sparse2 and both examples it was not necessary to improve any of α, β, γ or α_-, α_+, respectively, or to decrease the guess s for the smallest singular value.

Both examples demonstrate the effectiveness to treat nearly symmetric matrices. For both the small and large problem the input matrix should be symmetric but due to rounding errors the symmetry is lost. As a fact of the matter, `isequal(A',A)` is *false*. That might be one reason for the poor behaviour of Matlab's backslash operator.

For both problems the matrices are normed to about 1. The last line in Table 24 shows that in both cases A is a tiny perturbation of a symmetric matrix. The symmetrized matrix A^{sym} is not definite, so that for both problems subalgorithms "verifySparseNearSym1/2" are called, respectively. That improved the performance significantly. Without the method to treat nearly symmetric matrices introduced in Section 3, Algorithms Sparse1/2 would call "verifySparseGen1/2", the performance of which is shown in line "time Gen" in Table 24. The accuracy of the results is similar, however, about 15 and 22 times more computing time was necessary for the verification of the small example for Sparse1 and Sparse2, respectively, and about 4 times as much for the large.

11. Conclusion and an open problem. In this Part II of our note we discussed a second Algorithm for computing verified error bounds for systems of equations with sparse input matrix. The bounds are correct with mathematical certainty including the proof of existence and uniqueness of the solution. Like the method in Part I our algorithms are applicable to real and complex data as well as data afflicted with tolerances.

The second algorithm is often a little slower than the first one presented in Part I of this note, but seems a little more robust as it did not fail in any our examples. Our methods are usually slower than Matlab's built-in solver "backslash", but sometimes faster by some two orders of magnitude.

We gave algorithms to compute verified bounds for least squares problems as well as for underdetermined linear systems. Computational evidence suggests that even for very ill-conditioned problems accurate bounds are computed.

As an application of the solution of linear systems the data of which are afflicted with tolerances we described a method to compute verified error bounds for a system of real or complex nonlinear equations. The nonlinear problem is transformed into a linear system with point matrix and interval right hand side. In practical applications the Jacobian of a nonlinear system is often sparse. In that case our method is, provided that the system is not too ill-conditioned, superior to existing algorithms such as Algorithm `verifynlss` in INTLAB where the Jacobian is treated as a full matrix.

The norm of choice for interval linear systems is the ∞-norm. A major drawback of our methods is that it is based on the 2-norm of the residual $A\tilde{x} - b$, which often overestimates $\|A\tilde{x} - b\|_\infty$ by a factor close to $\sqrt{n}$. For point data that can be compensated by storing an approximation as an unevaluated sum and accurate dot products, for interval data both is not applicable. Thus, for an interval linear system where at least the right hand side is thick, i.e., diameters are nonzero, it remains an open problem to compute accurate error bounds for the solution set (8.8) in reasonable computing time. That is one of Neumaier's Great Challenges [29] posed in 2002. Mathematically the problem is equivalent to find a good upper bound for the ∞-norm condition number of A. With the methods given in the two parts of this note we obtain only sharp estimates of the 2-norm condition number.

The primary goal of our algorithms is to be successful, accepting some penalty in computing time. The second goal is to compute narrow error bounds. To the latter end we described a method in this Part II to obtain error bounds for the solution of linear systems which are almost always maximally accurate for all entries.

Acknowledgement. My special thanks go to the reviewers for their extremely valuable and constructive comments. They helped to improve the paper significantly.

REFERENCES

[1] J.P. Abbott and R.P. Brent. Fast local convergence with single and multistep methods for nonlinear equations. *Austr. Math. Soc. 19 (Series B)*, pages 173–199, 1975.
[2] P. Ahrens, J. Demmel, and H.D. Nguyen. Algorithms for efficient reproducible floating-point summation. *ACM TOMS*, 46:1–49, 2020.
[3] I.J. Anderson. A distillation algorithm for floating-point summation. *SIAM J. Sci. Comput.*, 20:1797–1806, 1999.
[4] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Nauman. *Automatic Differentiation of Algorithms – From Simulation to Optimisation*. Springer-Verlag, Berlin, 2002.
[5] T.A. Davis, Y. Hu: The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software 38, 1, Article 1, 2011.
[6] J.B. Demmel. On floating point errors in Cholesky. LAPACK Working Note 14 CS-89-87, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, 1989.
[7] J. Demmel, Y. Hida. Accurate and efficient floating point summation. *SIAM J. Sci. Comput. (SISC)*, 25:1214–1248, 2003.
[8] I.S. Duff, J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications (SIMAX)*, 22 (4):973–996, 2001.
[9] Iain S. Duff. Ma57—a code for the solution of sparse symmetric definite and indefinite systems.

ACM Trans. Math. Softw., 30(2):118–144, 2004.
[10] A. Griewank. A mathematical view of automatic differentiation. In *Acta Numerica*, volume 12, pages 321–398. Cambridge University Press, 2003.
[11] N. J. Higham: *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, 2nd edition, 2002.
[12] P. Holoborodko. *Multiprecision Computing Toolbox for MATLAB 4.6.4.13348*. Advanpix LLC., Yokohama, Japan, 2026.
[13] P. Holoborodko. *private communication, 2025*.
[14] IEEE Standard for Floating-point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.
[15] D. E. Knuth: *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, 1969.
[16] S.P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T.A. Davis, M. Henderson, Y. Hu, R. Sandstrom: The SuiteSparse Matrix Collection Website Interface. Journal of Open Source Software 4, 35, 1244-1248, 2019.
[17] C.-P. Jeannerod, S.M. Rump. Improved error bounds for inner products in floating-point arithmetic. *SIAM J. Matrix Anal. Appl. (SIMAX)*, 34(2):338–344, 2013.
[18] M. Lange and S.M. Rump. Error estimates for the summation of real numbers with application to floating-point summation. *BIT*, 57:927–941, 2017.
[19] M. Lange, S.M. Rump. Sharp estimates for perturbation errors in summations. *Math.Comp.*, 88:349–368, 2019.
[20] M. Lange and S.M. Rump. Accurate floating-point matrix residuals. to appear.
[21] X. Liu. private communication. 2025.
[22] X. Liu, M.T. Nakao, C. You and S. Oishi. Explicit a posteriori and a priori error estimation for the finite element solution of Stokes equations. Japan J. Indust. Appl. Math. 38, 545–559 2021. `https://doi.org/10.1007/s13160-020-00449-5`.
[23] MATLAB. User's Guide, Pre-Release 2026a, the MathWorks Inc., 2025.
[24] J.J. Moré and M.Y. Cosnard. Numerical solution of non-linear equations. *ACM Trans. Math. Software*, 5:64–85, 1979.
[25] J.J. Moré, D.C. Sorensen, K.E. Hillstrom, and B.S. Garbow. The MINPACK project. In W.J. Cowell, editor, *Sources and Development of Mathematical Software*, pages 88–111. Prentice Hall, 1984.
[26] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, R. Revol,, S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2nd edition, 2018.
[27] A. Neumaier. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *Zeitschrift für Angew. Math. Mech. (ZAMM)*, 54:39–51, 1974.
[28] A. Neumaier: *Interval Methods for Systems of Equations*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1990.
[29] A. Neumaier. Grand challenges and scientific standards in interval analysis. *Reliable Computing*, 8(4):313–320, 2002.
[30] T. Ogita, S. M. Rump, S. Oishi: Accurate sum and dot product. *SIAM Journal on Scientific Computing (SISC)*, 26(6):1955–1988, 2005.
[31] S. Oishi, K. Ichihara, M. Kashiwagi, T. Kimura, X. Liu, H. Masai, Y. Morikura, T. Ogita, K. Ozaki, S. M. Rump, K. Sekine, A. Takayasu, N. Yamanaka: *Principle of Verified Numerical Computations*. Corona Publisher, Tokyo, Japan, 2018. [in Japanese].
[32] K. Ozaki, T. Ogita, and S. Oishi. Tight and efficient enclosure of matrix multiplication by using optimized BLAS. *Numerical Linear Algebra with Applications*, 18(2):237–248, 2011.
[33] K. Ozaki, T. Ogita, and S. Oishi. Improvement of error-free splitting for accurate matrix multiplication. *Journal of Computational and Applied Mathematics*, 288:127–140, 2015.
[34] K. Ozaki, T. Ogita, and S. Oishi. Error-free transformation of matrix multiplication with a posteriori validation. *Numerical Linear Algebra with Applications*, 23(5):931–946, 2016.
[35] K. Ozaki, T. Ogita, S.M. Rump, and S. Oishi. Fast algorithms for floating-point interval matrix multiplication. *Journal of Computational and Applied Mathematics*, 236(7):1795–1814, 2012.
[36] K. Ozaki, T. Ogita, S. Oishi, and S.M. Rump. Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. *Numerical Algorithms*, 59(1):95–118, 2012.
[37] K. Ozaki, T. Ogita, S. Oishi, and S.M. Rump. Generalization of error-free transformation for matrix multiplication and its application. *Nonlinear Theory and its Applications*, 4(1):2–11, 2013.
[38] S.M. Rump. *Kleine Fehlerschranken bei Matrixproblemen*. PhD thesis, Universität Karlsruhe,

1980.
[39] S.M. Rump. Validated solution of large linear systems. In R. Albrecht, G. Alefeld, H.J. Stetter, editors, *Validation numerics: theory and applications*, volume 9 of *Computing Supplementum*, pages 191–212. Springer, 1993.
[40] S.M. Rump. Verified computation of the solution of large sparse linear systems. *Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, 75:S439–S442, 1995.
[41] S. M. Rump: INTLAB – INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Springer Netherlands, Dordrecht, 1999.
[42] S.M. Rump. Verified solution of large linear and nonlinear systems. In H. Bulgak, C. Zenger, editors, *Error Control and adaptivity in Scientific Computing*, pages 279–298. Kluwer Academic Publishers, 1999.
[43] S. M. Rump: Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.
[44] S.M. Rump. Improved componentwise verified error bounds for least squares problems and underdetermined linear systems. 66:309–322, 2013.
[45] S.M. Rump, T. Ogita. Super-fast validated solution of linear systems. *Journal of Computational and Applied Mathematics (JCAM)*, 199(2):199–206, 2006. Special issue on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2004).
[46] S.M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput. (SISC)*, 31(1):189–224, 2008.
[47] R. Skeel. Iterative refinement implies numerical stability for Gaussian elimination. *Math. Comp.*, 35(151):817–832, 1980.
[48] Terao T., K. Ozaki. Method for verifying solutions of sparse linear systems with general coefficients. Applied Mathematics and Computation, Volume 490, April 2025, 129204.
[49] G. Zielke, V. Drygalla. Genaue Lösung linearer Gleichungssysteme. *GAMM Mitt. Ges. Angew. Math. Mech.*, 26:7–108, 2003.
[50] Supplementary material to *Verified error bounds for sparse systems Part II*, 2025.