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 for general real or complex sparse linear systems with condition numbers 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. In contrast to the previous approach 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 slower than that in Part I, however, a little more stable. 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 with a very efficient Matlab implementation.

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. [26, 41, 29] and the literature cited over there. Many verification algorithms are included in INTLAB [39], the Matlab/Octave toolbox for reliable computing.

As mentioned in Part I, a verification method for sparse linear systems is part of the *Grand challenges* [27]. Although we cannot expect a general purpose algorithm, competitive known attempts such as in [43] work only for symmetric positive definite matrices.

In Part I we presented the splitting of the input matrix A in two factors $A \approx L_1L_2$

*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).

based on some LDL^T-decomposition. A key to a successful verification method is to compute accurate of residuals, here $\|A - L_1L_2\|_2$. The advantage of the splitting into two factors is that each entry of the residual $A - L_1L_2$ is a dot product, so that fast and accurate methods to compute accurate bounds for the residual norm can be used.

The methods in Part I and II explore the ideas in [37, 38, 40] published in the 1990's. For the time being the algorithms for LDL^T-decomposition were not stable enough to allow for good verification methods. Nowadays good scaling and equilibration routines are available [8, 9] making those methods attractive. That was observed by Terao and Ozaki [46] and triggered both parts of this note.

One key of our methods is the following theorem [38, 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 as well.

That approach for symmetric (positive definite) A was further explored in [43]. 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 2.5 and Corollary 2.6 in Part I of this note we improve the bound ϱ by using linear estimates on the rounding error of dot products [16, 17, 18] and a special application of Perron-Frobenius Theory.

Another application [40, 46] 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 $(-n, 0, n)$ for nonsingular A. 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 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

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

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, where both are 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 a little more often successful. We elaborate on that in several numerical examples in Section 9.

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. Besides 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 IEEE754 floating-point standard [13]. 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 [39] the command `setround(-1)` switches the rounding to downwards. That means that henceforth the result of all floating-point operations is executed in rounding downwards. That includes in particular vector and matrix operations. Similarly, `setround(1)` switches the rounding to upwards.

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.

Part II	Part I	description
(1.5)	(1.10)	$A^T = A \;\Rightarrow\; \lvert\lambda_k(A+E) - \lambda_k(A)\rvert \leqslant \lVert E\rVert_2$
(1.6)	(3.2)	equilibration of a symmetric matrix
(1.7)	(3.3)	equilibration of a general matrix
(1.8)	(3.5)	`[L,D,p] = ldl(A,thresh,'vector');`
(1.9)	(3.7)	remedy for LDL^T-decomposition
(1.10)	(7.1)	decomposition of D
(1.11)	(2.10)	norm of residual using a priori bounds
(1.12)	(3.9)	approximation of smallest singular value

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

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

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 often and for all entries maximally accurate.

After discussing how to compute the inertia of the block matrix D of an LDL^T-decomposition we explain our alternative method for symmetric and for general input matrix; our approach for symmetric (positive) definite matrices does not change. 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 `verifySparselss0` 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 square sparse linear systems, we present Algorithm `verifySparseNlss` for computing error bounds for the solution of a system of nonlinear equations.

Numerical examples for test matrices out of [5], for randomly generated matrices and nonlinear systems are shown. We close this note with concluding remarks and further open problems.

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. The are Matlab implementations, however, they suffer from interpretation overhead, in particular for sparse data. We used Advanpix [12] in Part I 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} according to the IEEE754 standard [13]. That implementation respects the specified rounding mode, for the arithmetic operations as well as for the type cast `double(·)` from `mp` to double precision. Thus the code

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

computes a floating-point matrix `Q` such that $|AB - C| \leqslant$ `Q` is true for the real matrix $AB - C$ using entrywise absolute value and comparison, see Lemma 2.4 in Part I of this note.

The main reason to use the toolbox Advanpix [12] in Part I was to show a fair comparison with [46] because it was also used in there. However, in this Part II we use higher precision to achieve even more accurate bounds. That seems not possible in [12] because except extended precision as by `mp.Digits(34)` it is not clear whether different rounding modes are treated correctly.

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 [19, 20] on a new algorithm improving on [32]. 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 [47] an *absolute splitting* of vectors was introduced, following the scheme

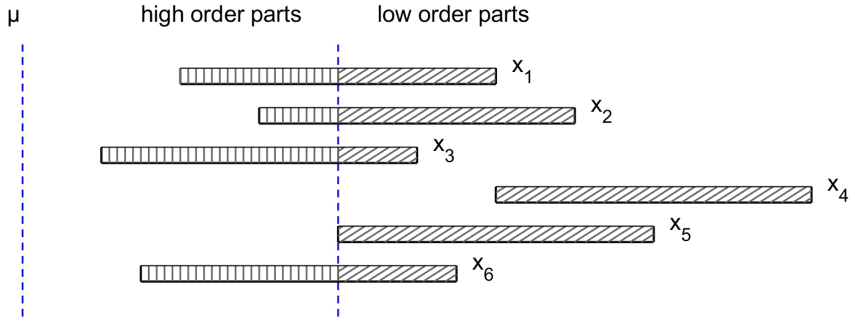

FIG. 1. *The Zielke/Drygalla scheme to extract high and low order parts*

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 [44] 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 [30, 34, 35] with improvements in [31, 32]. 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. 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] Ozaki et al. [34, 33, 32] 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, 25, 7, 28, 44]. 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 [34], one can expect the error to be roughly

[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.

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` [28], 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 [19, 20] we discuss various improvements to the original Ozaki scheme. The most important for this note is to specify a precise splitting point. 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 implementation in Algorithm `prodK` is pure Matlab code and due to Marko Lange [19, 20]. Despite the interpretation overhead it is faster than the mex-files used in Advanpix [12]. 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. As can be seen, for full matrices

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

`vpa` is much slower than `mp`, and for little larger dimension `prodK` is significantly faster than `mp`.

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

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)`.

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

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

For the first pairs of input parameters $p_1, q_1, p_2, q_2, \ldots$ the value $\sum p_i q_i$ will be computed, where the 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 (1.4) in Part I we introduced a notation for the approximation and inclusion of a residual $Ax - b$ with sample Matlab/INTLAB code in (1.5). 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` imply a result "as if" evaluated in $k/2+1$-fold precision. Therefore using `spProdK` sample Matlab/INTLAB code is

$$
\begin{array}{ll}
[\![expr]\!]_{k,1} & \texttt{res = spProdK(A,x,-1,b,2*(k-1));} \\
\langle\!\langle expr \rangle\!\rangle_{k,1} & \texttt{[res,err] = spProdK(A,x,-1,b,2*(k-1));} \\
 & \texttt{res = midrad(res,err);}
\end{array} \tag{2.4}
$$

For compatible matrices A, B, C we borrow the function `NormBnd` in (1.8) and the code in (2.16) of Lemma 2.7 in Part I to bound $\|AB - C\|_2$:

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

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 (2.17) of Lemma 2.7 in Part I is now replaced by

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

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 $A \approx L_1 L_2$ so that the entries of the residual $L_1 L_2 - A$ 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, C_2, E_1 with

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

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

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

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 and computes α_1 with $\||C| + E_2\|_2 \leqslant \alpha_1$ so that 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_2 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 1 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 [36] and store an approximation in 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. Using twice the working precision is not sufficient, i.e., 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 1 in Part I of this note. Algorithm `ErrorBound3` is given in Table 4. If necessary, the steps 6 and 7 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	$[\tilde{x}, \tilde{y}, \tilde{z}] = \text{spProdK}(1, \tilde{x}, 1, \tilde{y}, 1, \tilde{z}, 4)$	% $A^{-1}b \approx \tilde{x} + \tilde{y} + \tilde{z}$
7	$\tilde{z} = \text{solve}(A, [\![b - A\tilde{x} - A\tilde{y} - A\tilde{z}]\!]_{3,1})$	% $A^{-1}b \approx \tilde{x} + \tilde{y} + \tilde{z}$
8	setround(-1); $\varrho = \text{abs}\,([\![A\tilde{x} + A\tilde{y} + A\tilde{z} - b]\!]_{3,1})$	
9	setround(+1); $\varrho = \max(\,\varrho\,, \text{abs}\,([\![A\tilde{x} + A\tilde{y} + A\tilde{z} - b]\!]_{3,1})\,)$	
11	$\delta = \lvert\tilde{y}\rvert + \texttt{vecnorm}(\varrho)/s$	

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. Inertia of a 2×2 Hermitian 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, the number of negative, zero and positive eigenvalues, is $\iota(M) = (1, 0, 1)$. If $\det(M) > 0$, then $\iota(M) = (0, 0, 2)$ if $\text{trace}(M) > 0$ and $\iota(M) = (2, 0, 0)$ otherwise.

We suppose a floating-point computation in some nearest rounding barring over- and underflow. A nearest rounding is defined by a rounding function $\text{fl} : \mathbb{R} \to \mathbb{F}$. For $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$ that means that 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.,

$$x, y \in \mathbb{R}: \quad \mathrm{fl}(x) < \mathrm{fl}(y) \Rightarrow x < y \qquad \text{and} \qquad x < y \Rightarrow \mathrm{fl}(x) \leqslant \mathrm{fl}(y) \ . \tag{3.1}$$

Since zero is a floating-point number, it follows

$$a, c \in \mathbb{F}: \quad \mathrm{fl}(a+c) < 0 \ \Leftrightarrow \ a + c < 0 \ . \tag{3.2}$$

Here $\Rightarrow$ is clear, and for $\Leftarrow$ note that $\mathrm{fl}(a+c) = 0$ is only possible if $a+c$ is below the smallest denormalized floating-point number. However, in that case $\mathrm{fl}(a+c) = a+c$, cf. [24].

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 (3.1) implies

$$p - q < 0 \ \Rightarrow \ ac < b^2 \ \Leftrightarrow \ \det(M) < 0 \tag{3.3}$$

and similarly for $p - q > 0$. It remains the case $p = q$. Since p, q are computed in floating-point, still $\det(M) \neq 0$ is possible and the sign has to be decided. In that rare case we us the error-free transformation `TwoProduct` [14, 44, 24]. 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

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

Then

$$p = q \quad \Rightarrow \quad \det(M) = ac - bd = e - f$$

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

The Algorithm `NumPosEV` in Table 5 is executable Matlab/INTLAB code and computes the number of positive eigenvalues of a symmetric matrix $M :=$ `[a b;b c]`. The first line sets the rounding mode to nearest [39]. 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.

4. Symmetric matrices. We show in Table 6 a general outline of our modified subalgorithm "verifySparseSym0" to compute verified bounds for the solution of a sparse linear system with symmetric matrix.

Our second method explores on Theorem 1.1 published in [38, Theorem 1.1]; the difference to the method on Part I of this note will be explained at the end of this section. The original method in [38, 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 where $A_s - LDL^T$ 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.

Lines 2 – 4 are as in subalgorithm `VerifySparseSym` in Part I of this note. In Line 5 the approximate decomposition of A is used to compute s, an anticipated lower bound on the smallest singular value of A.

```
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 = sign(a+c);
  end
end  % function NumPosEV
```

TABLE 5
Computing the number `p` *of positive eigenvalues of* $M :=$ `[a b;b c]`.

In order to distinguish the factors, we denote A_s in Lines 6 and 14 by $\mathtt{As}_-$ and $\mathtt{As}_+$, respectively. The matrix $\mathtt{As}_-$ in Line 6 is computed in rounding downwards and therefore a lower bound on $A - sI$, i.e., $\mathtt{As}_- = A - sI - \Delta_-$ for a diagonal and nonnegative matrix Δ_-, and similarly for $\mathtt{As}_+$.

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

$$\|\mathtt{As}_- - P_- Q_- P_-^T\|_2 \leqslant \alpha_- \qquad \text{and} \qquad \|\mathtt{As}_+ - P_+ Q_+ P_+^T\|_2 \leqslant \alpha_+. \tag{4.1}$$

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 (1.5) implies

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

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

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

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

$$\sigma_{\min}(A) = \min\left(-\lambda_{k+1}(A), \lambda_k(A)\right) \geqslant s - \max(\alpha_-, \alpha_+) \ . \tag{4.2}$$

Now in Step 7–8 an approximate decomposition $\mathtt{As}_- \approx L_1 S L_1^T$ 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

1 function $[x,\delta]$ = verifySparseSym0(A,b)
2 Equilibrate A by (1.6)
3 Compute $LDL^T(A)$ by (1.8)
4 If D is singular, verification failed, $[x,\delta]$ = verifySparseGen0(A,b), return
5 Compute $\tilde{s}(A,L,D)$ by (1.12) and set $s := 0.9\tilde{s}$, $\Phi = true$
6 Rounding downwards, $A_s := A - sI$ and compute $L_s D_s L_s^T(A_s)$ by (1.8)
7 Compute approximate splitting $D_s \approx \widehat{D_s} S \widehat{D_s}^T$ according to (1.10)
8 Compute $L_1 \approx LD_s$ and $L_2 = SL_1^T$
9 Use (1.11) to compute α_- with $\|A_s - L_1 L_2\|_2 \leqslant \alpha_-$
10 If $\alpha_- \geqslant s$, improve α_- by (2.5)
11 If $\alpha_- < s$, ν_- = `sum(S)` > 0, goto Step 13
12 Compute α_- with $\|A_s - L_s D_s L_s^T\|_2 \leqslant \alpha_-$ as in Table 3, $\nu_- = \pi(D_s)$
13 If $\alpha_- \geqslant s$, first verification failed, go to Step 22
14 Rounding upwards, $A_s := A + sI$ and compute $L_s D_s L_s^T(A_s)$ by (1.8)
15 Compute approximate splitting $D_s \approx \widehat{D_s} S \widehat{D_s}^T$ according to (1.10)
16 Compute $L_1 \approx LD_s$ and $L_2 = SL_1^T$
17 Use (1.11) to compute α_+ with $\|A_s - L_1 L_2\|_2 \leqslant \alpha_+$
18 If $\alpha_+ \geqslant s$, improve α_+ by (2.5)
19 If $\alpha_+ < s$, ν_+ = `sum(S)` > 0, goto Step 21
20 Compute α_+ with $\|A_s - L_s D_s L_s^T\|_2 \leqslant \alpha_+$ as in Table 3, $\nu_+ = \pi(D_s)$
21 Set $\alpha = \max(\alpha_-, \alpha_+)$, if $\alpha < s$, go to Step 23
22 If Φ, $\Phi = false$, $s = s/5$, goto Step 6, else $\nu_- = 0$
23 If $\nu_- \neq \nu_+$, verification failed, $[x,\delta]$ = verifySparseGen0(A,b), return
24 $[x,\delta]$ = ErrorBound$(B, [0;b], s-\alpha,$ "$solve$") using LDL^T for solve

TABLE 6
Verified error bounds for $A^{-1}b$ for general sparse input matrix A.

α_- is computed and possibly improved in Step 10 such that $\|\texttt{As}_- - L_1 S L_1^T\| \leqslant \alpha_-$. If $\alpha_- < s$ in Step 11, 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 11, we set $P_- := L_s$ and $Q_- := D_s$ and compute the upper bound α_1 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 5. 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 22 to try once more with decreased s.

The computations in Lines 14 – 20 are similar to those in Lines 6 – 12 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 21 and $\nu_- = \nu_+$ in Step 23, then $k = \ell$ and (4.2) implies $\sigma_{\min}(A) \geqslant s - \alpha > 0$.

If $\alpha := \max(\alpha_-, \alpha_+) \geqslant s$ in Step 21, then as before a reason may be that s is too

large. In that case we reduce s and try the verification from Lines 6 – 21 again. If still $\alpha \geqslant s$ or $\nu_- \neq \nu_+$ in Step 23, then verification failed and we turn to subalgorithm "verifySparseGen0".

If the verification was successful, the positive lower bound $s - \alpha$ 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 "verifySparseSym" 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 $s-\alpha$ 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. As a consequence "verifySparseSym0" is slower but seems a little more stable.

In "verifySparseSym" 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 stable. See the numerical results in Section 9.

5. General matrices. As in [38, 40] our method for linear systems with general matrix uses the augmented matrix

$$B := \begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix} \tag{5.1}$$

the singular values of which are ± the eigenvalues of A. This matrix is used in [46] as well.

As in the symmetric case we explore on Theorem 1.1 published in [38, Theorem 1.1]. The original method relied 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 is known beforehand. That is at least true for nonsingular matrix A. We do not assume nonsingularity of A beforehand but prove it a posteriori so that all deductions are true.

Rather than LDL^T as in [38, 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, were $L_2 = SL_1^T$ for a signature matrix S. That implies the same advantages as in the symmetric case.

In contrast to [38, 40, 46] 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 (1.8). 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, and we cure it as in (1.7).

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 (1.12) to compute s as a lower bound of $\sigma_{\min}(B)$ rather than of $\sigma_{\min}(A)$ is more stable due to the symmetry of B.

A splitting (1.10) 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 "verifySparseSym" in Part I of this note the factors L_1, L_2 have identical sets of singular values.

1 function $[x,\delta]$ = verifySparseGen0(A,b)
2 Equilibrate A by (1.7)
3 Let B the augmented matrix (5.1)
4 Compute $LDL^T(B)$ by (1.8)
5 If $nnz(D) < 2n$, compute $LDL^T(B)$ by (1.9)
6 If $nnz(D) < 2n$, verification failed, return
7 Compute $\tilde{s}(B,L,D) \lesssim \sigma_{\min}(B)$ by (1.12) and set $s := 0.9\tilde{s}$, $\Phi = true$
8 Rounding downwards, $B_s := B - sI$ and compute $L_s D_s L_s^T(B_s)$ by (1.8)
9 Compute approximate splitting $D_s \approx \widehat{D_s} S \widehat{D_s}^T$ according to (1.10)
10 Compute $L_1 \approx L D_s$ and $L_2 = S L_1^T$
11 Use (1.11) to compute α with $\|B_s - L_1 L_2\|_2 \leqslant \alpha$
12 If $\alpha < s$, improve α by (2.5)
13 If $\alpha < s$, ν = `sum(Ds)` > 0, else improve α by (2.6), $\nu = \pi(D_s)$
14 If $\alpha < s$, go to Step 16
15 If Φ, $\Phi = false$, $s = s/5$, goto Step 8, else $\nu = 0$
16 If $\nu \neq n$, verification failed, return
17 $[x,\delta]$ = ErrorBound$(B, [0;b], s-\alpha$, "$solve$") using LDL^T for solve

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

The remaining of the subalgorithm `verifySparseGen0` is identical to subalgorithm `verifySparseGen` in Table 5 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

$$\begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \tag{5.2}$$

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

As for "verifySparseSym0" the difference is that "verifySparseGen0" shifts the augmented matrix and computes a lower bound for $\sigma_{\min}(B)$ using Sylvester's law of inertia. In contrast, "verifySparseGen" 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 "verifySparseGen0" does. In general, however, "verifySparseGen0" seems slower because the decomposition of the shifted causes additional fill-in, see the computational results in Section 9.

6. 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])[3]

$$\text{(6.1)} \qquad \begin{pmatrix} 0 & A^H \\ A & -I_m \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \quad \Rightarrow \quad A^H y = 0 \ \text{ and } \ Ax - y = b \ ,$$

where I_m denotes the $m \times m$ identity matrix. Multiplying the second equation by A^H yields $A^H Ax = A^H b$. 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 (6.1) is symmetric indefinite, so our subalgorithms "verifySparseSym" and "verifySparseSym0" are applicable. In [42] 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 [42] that the computed inclusions are sometimes more accurate than using (6.1). However, for our present approach we have to 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

$$\text{(6.2)} \qquad \begin{pmatrix} -I_n & A^H \\ A & 0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \quad \Rightarrow \quad Ax = b \ \text{ and } \ A^H y = x \ ,$$

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 $\|x\|_2$.

The linear systems in (6.1) and (6.2) can be solved by "verifySparseSym" or "verifySparseSym0". However, in Algorithm `verifySparselss0` in Table 10 we call recursively "verifySparselss0". Since the augmented matrix is square, that leads directly to the case distinctions for real or complex matrices.

In subalgorithms "verifySparseSym" or "verifySparseSym0" the symmetric equilibration (1.7) (which is (3.3) in Part I of this note) is applied, i.e., two steps of 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 produces stable results.

When computing error bounds for the square linear systems (6.1) or (6.2) 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.

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

[3] We may use $+I_m$ or $-I_m$ in the lower right corner of the system matrix; in order to cover complex matrices and keep the algorithm to be presented in Section 8 simple, we use $-I_m$ because "verifySparseSPD" recognizes immediately that the system matrix cannot be positive definite.

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.

As a consequence we cannot compare our results with that of Matlab's backslash operator.

7. Systems of nonlinear equations. In this section we need some more details on interval operations, in particular the use of INTLAB [39]. 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` 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 derivative of a function. If the argument is of type `intval`, then a mathematically rigorous inclusion is computed. For details, see [39, 41].

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 a Matlab program `f` to be 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 $\xi_1, \ldots, \xi_n \in x \underline{\cup} \tilde{x}$, the convex union of x and $\tilde{x}$, with

$$f(x) = f(\tilde{x}) + \begin{pmatrix} \nabla f_1(\xi_1) \\ \cdots \\ \nabla f_n(\xi_n) \end{pmatrix} \left(x - \tilde{x}\right) \tag{7.1}$$

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 (7.1) is only rowwise equal to some Jacobian J_f of f.

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

$$\texttt{Y = f(gradientinit(X))} \tag{7.2}$$

computes `Y` such that `Y.x` $\in \mathbb{IF}^n$ is an interval vector with $\{f(x) : x \in \mathbf{X}\} \subseteq$ `Y.x`, and `Y.dx` is an interval matrix `Y.dx` $\in \mathbb{IF}^{n \times n}$ with $\{\nabla f_k(\xi) : \xi \in$ `X`$\} \subseteq$ `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

$$\mathrm{hull}(X) := \bigcap \{\mathbf{Z} \in \mathbb{IR}^n : X \subseteq \mathbf{Z}\} \ . \tag{7.3}$$

For $x, \tilde{x} \in \mathbf{D}$ also $\mathbf{X} := \mathrm{hull}(x \underline{\cup} \tilde{x}) \subseteq \mathbf{D}$, and (7.2) implies

$$\begin{pmatrix} \nabla f_1(\xi_1) \\ \cdots \\ \nabla f_n(\xi_n) \end{pmatrix} \in \texttt{Y.dx} \tag{7.4}$$

for all $\xi_1, \ldots, \xi_n \in \mathbf{X}$. Therefore [41, Theroem 13.1], using interval operations the Mean Value Theorem can be written in the following elegant way.

THEOREM 7.1. *Let continuously differentiable* $f : \mathbf{D} \to \mathbb{R}^n$ *with* $\mathbf{D} \in \mathbb{IR}^n$ *and* `x,xs` $\in \mathbf{D} \cap \mathbb{F}^n$ *be given. Define* `Y = f(gradientinit(hull(x,xs)))`. *Then*

$$f(x) \in f(\tilde{x}) + \texttt{Y.dx}\big(x - \tilde{x}\big) \ . \tag{7.5}$$

Using this we can formulate [41, 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 7.2. *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*

$$S(\mathbf{X}, \tilde{x}) := -Rf(\tilde{x}) + \big\{I - RJ_f(\tilde{x} + \mathbf{X})\big\}\mathbf{X} \ \subseteq \ int(\mathbf{X}) \tag{7.6}$$

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 7.2 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 (7.6) of Theorem 7.2 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 [41, Section 13, page 87] and compute an inclusion $\mathbf{J} \in \mathbb{IF}^{n \times n}$ of $J_f(\tilde{x}+\mathbf{X})$ as in (7.2). Hence (7.4) 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 7.1 is applicable. Denote $\check{C} = \mathrm{mid}(\mathbf{J})$ and $\Delta := \mathrm{rad}(\mathbf{J})$. Assume that $\check{C}$ is nonsingular and suppose

$$\{y \ : \ \check{C}y = -f(\tilde{x}) - \varrho x, \ -\Delta \leqslant \varrho \leqslant \Delta, \ x \in \mathbf{X}\} \subseteq \mathbf{Y} \ . \tag{7.7}$$

for $\mathbf{Y} \in \mathbb{IF}^n$. Then $\mathbf{Y} \subset \mathrm{int}(\mathbf{X})$ implies (7.6). To see this set $R := \check{C}^{-1}$ and observe

$$-\check{C}^{-1}f(\tilde{x}) + \big\{I - \check{C}^{-1}[\check{C} + \varrho]\big\}x = \check{C}^{-1}\big(- f(\tilde{x}) - \varrho x\big)$$

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 (7.6) for $R := \check{C}^{-1}$. 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. Note that (7.6) proves the nonsingularity of $\check{C}$ as well.

Now $\mathbf{X}$ is an anticipated inclusion of the difference of the true solution $\widehat{x}$ 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 $\widehat{x} - \tilde{x} \in \mathbf{Y}$. 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 (7.7) by using the magnitudes[4] $\overline{X}$ 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 $\mathbf{Y} \subset \mathrm{int}(\mathbf{X})$.

[4]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.

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:

$$\Sigma(A,\mathbf{b}) := \{x \in \mathbb{R}^n : \exists b \in \mathbf{b} \text{ with } Ax = b\} \ . \tag{7.8}$$

To that end we use Algorithm "verifySparselss" as in Table 6 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 "verifySparseSPD", "verifySparseSym" and "verifySparseGen" by the call of "ErrorBoundI" as shown in Table 8.

```
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 8

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 [45] that implies backward stability of the approximate solution $\tilde{x}$ of the midpoint equation $Ax = \mu$. Line 5 computes an inclusion `rho ± err` of the residual $|A\tilde{x}-\mu|$, such that in particular $|A\tilde{x}-\mu| \leqslant |\texttt{rho}|+\texttt{err}$. Now `delta` in Line 7 is computed in rounding upwards, and with the lower bound s on $\sigma_{\min}(A)$ it follows

$$\begin{aligned} |A^{-1}b - \tilde{x}| &\leqslant |A^{-1}|\,|b - A\tilde{x}| \\ &\leqslant |A^{-1}|\,(|\mu - A\tilde{x}| + r) \\ &\leqslant \|A^{-1}\,(|\mu - A\tilde{x}| + r)\,\|_\infty \mathbf{e} \\ &\leqslant \|A^{-1}\|_2 \||\texttt{rho}| + \texttt{err} + r\|_2 \mathbf{e} \\ &\leqslant \delta \end{aligned} \tag{7.9}$$

for all $b \in \mathbf{b}$. Let `J` $\in \mathbb{IF}^{n\times n}$ be an inclusion of $J_f(\tilde{x} + \mathbf{X})$ computed as in (7.2) and consider

```
y = -f(intval(xs));
setround(1)
b = midrad( y.mid , y.rad + J.rad * mag(X) );
```
(7.10)

```
function [X,kxs,kY] = verifySparseNlss(f,xs)
  setround(0)
  n = size(xs,1); phi = 1e-14*sqrt(n);
  dxs = abs(xs); kxs = 0;
  while ( kxs < 15 )           % at most 10 Newton iterations
    kxs = kxs + 1; xsold = xs;
    y = f(gradientinit(xs));   % function value and gradient
    xs = xs - y.dx\y.x;        % approximate Newton iteration
    d = abs(xs-xsold);
    if all(d<.5*abs(xs)) && ( norm(d,inf)<=phi*norm(xs,inf) )
      break
    end
  end
  ys = -f(intval(xs));          % inclusion of f(xs)
  Y = mag(ys);                 % magnitude of ys
  kY = 0; setround(1)
  while ( kY < 10 )
    kY = kY + 1;
    X = 1.01*Y + realmin;      % epsilon-inflation
    JJ = f(gradientinit(midrad(xs,X)));
    J = JJ.dx;                 % inclusion of Jacobian
    b = midrad( ys.mid , ys.rad + J.rad*X );
    [Ys,delta] = verifySparse(J.mid,b);
    Y = abs(Ys) + delta;       % r.h.s. of (7.7)
    if all( Y < X )
      X = midrad(xs,Y);        % inclusion successful
    return
  end
  X = intval(NaN(size(xs)));   % inclusion failed
end  % function verifySparseNlss
```

TABLE 9

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

The first line computes an inclusion $\mathtt{y} \in \mathbb{IF}^n$ of $-f(\tilde{x})$ with $-f(\tilde{x}) \in \mathtt{y.mid} \pm \mathtt{y.rad}$. The second statement switches the rounding to upwards, and finally $\mathbf{b}$ is an inclusion of $\mathtt{y.mid} \pm \varrho$ for all $|\varrho| \leqslant \mathtt{y.rad + J.rad*mag(X)}$. 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 $\check{C}$ and right hand side $\mathbf{b}$ satisfies (7.7). 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 $\mathbf{b}$ is 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 of $\mathbf{b}$, which in turn increases with the sensitivity of the problem. Therefore, there is no need to improve an approximate solution of $\check{C}y = \mathbf{b}$ by a residual iteration and we may apply algorithms "verifySparselss" or "verifySparselss0" with using Algorithm "ErrorBoundI" as in Table 8 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 9. 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 [39]. 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 – 28 are an interval iteration adapted to the description in [41]. 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 [36]. 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. The success of the epsilon-inflation can be analyzed theoretically, see [41]. 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, according to (7.9) around $-f(\tilde{x})$, but that 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[5] `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 (7.10), and the next line an inclusion $Ys \pm \delta$ of the linear system with matrix $\check{C} = \mathrm{mid}(\mathbf{J})$ 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 little widened Y. In some way these are also Newton steps. 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. Moreover, the solution of a sparse system of linear equations with interval matrix and interval right hand side is required. That introduces overestimations, and eventually the condition `Y < X` in line 25 of Algorithm `verifySparseNlss` in Table 9 is not satisfied for too large and too ill-conditioned nonlinear systems. Nevertheless the method works well in a number of examples, see the test results in Section 9.

8. 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 `verifySparselss` for our algorithm presented in Part I (also called "new" in there) and use `verifySparselss0` for the algorithm presented in this Part II (henceforth called "new0"). The latter is identical to

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

```
function [xs,delta] = verifySparselss0(A,b,acc)
% Approximate solution xs of Ax=b with error bound delta
  [m,n] = size(A);
  if m>n                    % least squares problem
    B = [ sparse(n,n) A' ; A -speye(m) ];
    [xs,delta] = verifySparselss0(B,[zeros(n,size(b,2));b],acc);
    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] = verifySparselss0(B,[zeros(n,size(b,2));b],acc);
    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
      symm = isequal(A',A);
      if symm               % A symmetric
        [xs,delta] = verifySparseSPD(A,b);
      end
      if ( ~symm ) || isnan(xs(1))   % A unsymm. or SPD failed
        [xs,delta] = verifySparseGen0(A,b);
      end
    else                    % A real, b complex
      [xs,delta] = verifySparselss0(A,[real(b) imag(b)]);
      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] = verifySparselss0(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 verifySparselss0
```

TABLE 10

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.

the former except replacing subalgorithms "verifySparseSym" and "verifySparseGen" by "verifySparseSym0" and "verifySparseGen0", respectively. Executable code of Algorithm `verifySparselss0` including least squares problems and underdetermined linear systems is presented in Table 10.

The algorithm first checks for the type of problem, namely $m > n$ for a least squares problem and $m < n$ for an underdetermined system of equations. In either case Algorithm `verifySparselss0` is called using (6.1) or (6.2), respectively. If $m = n$, verified error bounds for a linear system with square matrix are computed with code identical to Algorithm `verifySparselss` in Table 6 in Part I of this note. The subalgorithm "verifySparseSPD" in Table 3 of Part I of this note is used except that in case of failure in lines $2, 5$ and 12 subalgorithm "verifySparseSym0" is called instead of "verifySparseSym".

The algorithm in Part I of this note is adapted to least squares problems and underdetermined linear systems similar to Algorithm `verifySparselss0` by replacing subalgorithms "verifySparseSym0" and "verifySparseGen0" by "verifySparseSym" and "verifySparseGen", respectively.

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. As has been mentioned, we restrict computational tests to least squares problems because Matlab does not compute an approximation of A^+b for underdetermined linear systems.

9. 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 2023b [21] under Windows 10. Henceforth we call Algorithm `verifySparselss` "new" as in Part I, and Algorithm `verifySparselss0` "new0".

We use the same set of test matrices from the Suite Sparse Matrix Collection [5] with the interface [15] as in Part I, namely we treat all real and complex square matrices with dimension

$$10^3 \leqslant n \leqslant 10^5 \quad \text{and} \quad 10^{10} \leqslant \texttt{condest(A)} \leqslant 10^{16} \quad \text{and} \quad \texttt{nnz(A)} \leqslant 10^6 \ . \tag{9.1}$$

Test matrices with symmetric positive definite input matrix are omitted because the corresponding subalgorithms in `verifySparselss` and `verifySparselss0` coincide.

That resulted in totally 284 tests displayed in Table 11. The first column indicates the structure indicated by [5], namely symmetric indefinite, general real, all test matrices out of [46], complex Hermitian positive definite and general complex. Our first Algorithm `verifySparselss` in Part I of this note computed verified bounds in 301 out of the 306 test cases, whereas Algorithm `verifySparselss0` presented here failed in only one test case, namely number 1247 in [5]. For that case Algorithm `verifySparselss` failed as well. We discuss that case later.

The dimension, number of nonzero elements and condition number of all 284 test cases is shown in Figure 2. The dimensions vary between 1019 and $682,862$ and

TABLE 11
Test sets and success rate.

structure	success `verifySparselss`			success `verifySparselss0`		
sym	45	out of	48	47	out of	48
gen	210	out of	211	211	out of	211
[46]	20	out of	20	20	out of	20
complex spd	1	out of	1	1	out of	1
complex gen	3	out of	4	4	out of	4

the number of nonzero elements between 3562 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.

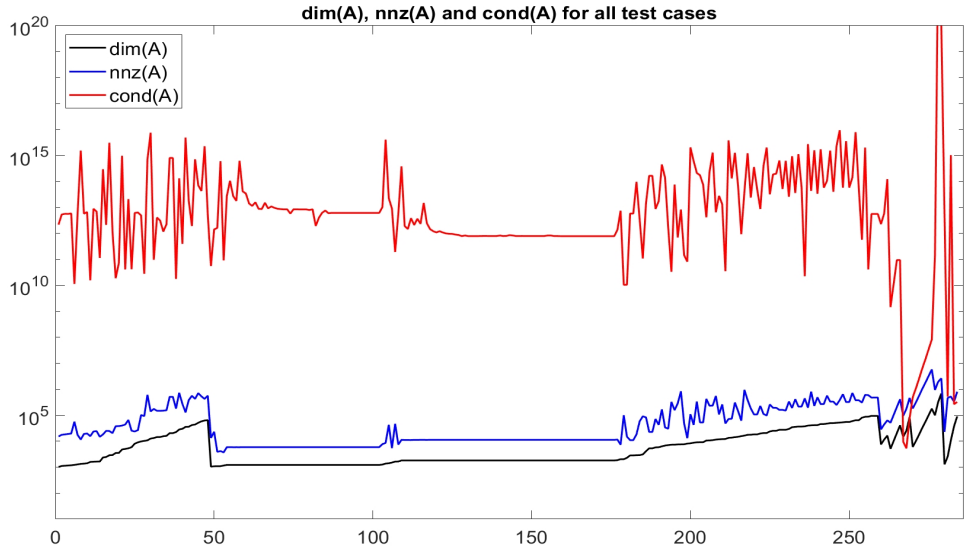

FIG. 2. *Dimension, number of nonzero elements and condition number of all test matrices.*

In Figure 3 we show for all tests the ratio of computing times of Algorithm `verifySparselss0` (henceforth also called "new0") divided by that of Algorithm `verifySparselss` (henceforth also called "new"). The ratios are displayed if "new" (and therefore also "new0") is successful. That explains the gap at case 30. A number less than 1 means that "new0" is faster than "new". That is rarely the case. In the median over all examples Algorithm `verifySparselss` from Part I of this note is faster than `verifySparselss0` by a factor 1.22, at most by a factor 5.2. Conversely, "new0" is faster than "new" by at most a factor 2.6.

In some way Algorithm `verifySparselss0` is simpler than `verifySparselss`, so we may ask why it is slower. Both algorithm start with computing some factor L_1, both for symmetric as for general matrices. However, "new" computes for symmetric input matrix A a factor of A, but "new0" of A shifted by s. Similarly, "new" computes a factor of the augmented matrix B, but "new0" of B shifted by s for general input

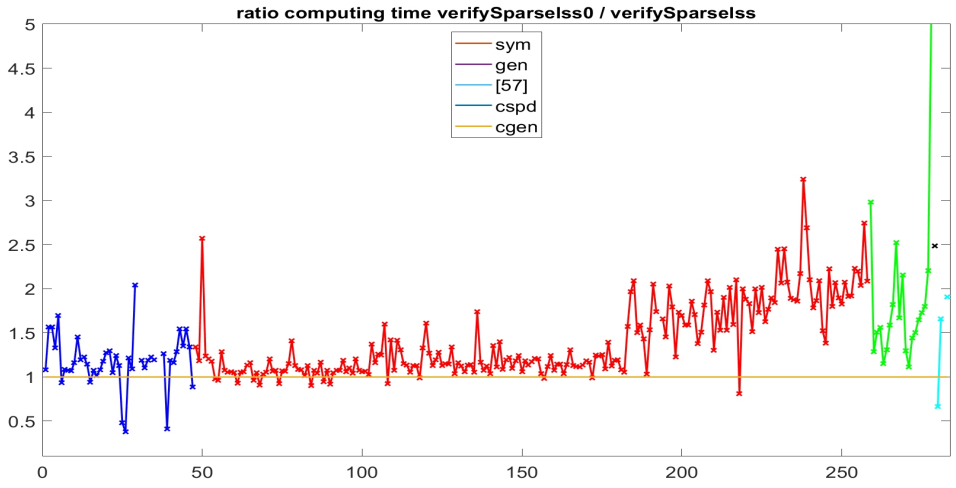

FIG. 3. *Ratios of computing times* $t_{\texttt{verifySparselss0}}/t_{\texttt{verifySparselss}}$.

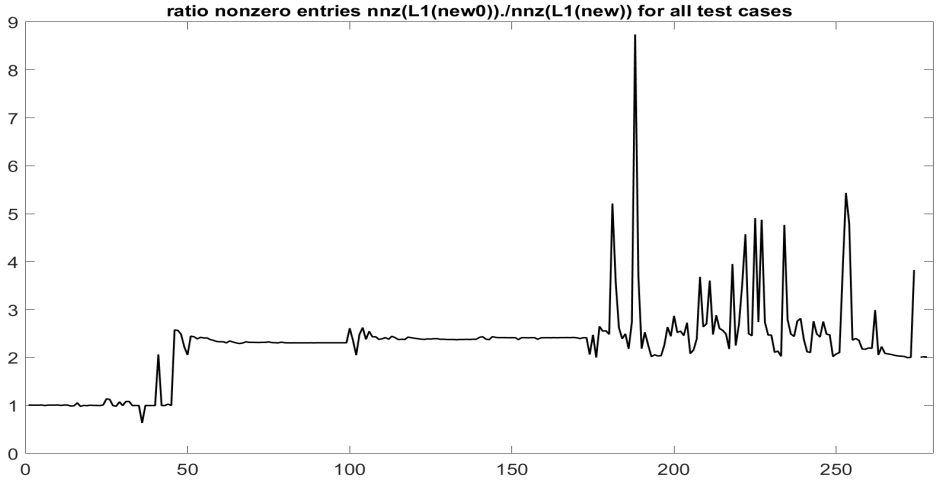

FIG. 4. *Ratio of number of nonzero entries of* L_1 *in "new0" divided by that of "new".*

matrix A. That causes a significant fill-in for method "new0". In Figure 4 we display the ratio of the number of nonzero entries of the factor L_1 in `verifySparselss0` divided by that of `verifySparselss`. Hence a value greater than 1 means that "new0" has more fill-in than "new".

The median ratio of fill-in over all examples is 2.4, and maximally the factor L_1 by "new0" has 8.7 times more elements than that of L_1 by "new". That is true although we reduced the number of elements as explained in (3.5)ff in Part I of this note by setting entries in L smaller than 10^{-30} in magnitude to zero in case the first LDL^T-decomposition failed due to singular D.

Next we show in Figure 5 a rough image of the median relative error of the

Algorithms `verifySparselss` and `verifySparselss0`. As can be seen in both cases

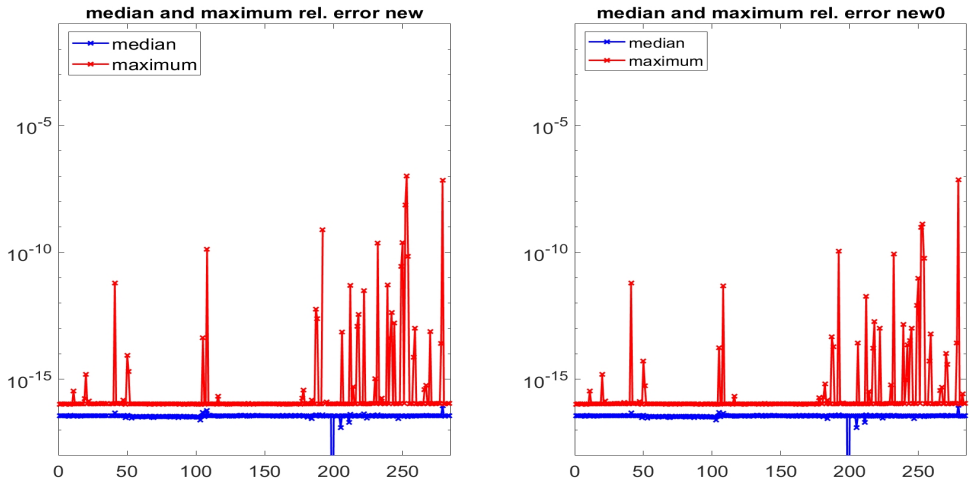

FIG. 5. *Median of relative errors of* `verifySparselss` *and* `verifySparselss0`.

usually almost maximally accurate approximations are computed. In the median the relative error of all entries of the inclusion computed by Algorithms `verifySparselss` and `verifySparselss0` is $3.6 \cdot 10^{-17}$, the maximum relative error over all entries of the inclusions is around 10^{-7}.

We discuss some details of our Algorithm `verifySparselss0` on the several improvement steps in the subalgorithms "verifySparseSym0" and "verifySparseGen0". 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 begin with subalgorithm "verifySparseSym0". The security measure on singular D in step 4 occurred occasionally while developing Algorithm `verifySparselss0`, in the sym tests with (9.1) it did not happen. The improvement of α in line 10 was used 10 times, the second improvement in line 11 was used in 4 out of the 48 tests. For one test case the value s was decreased in line 22. Failure in line 23 occurred in 4 out of the 48 sym tests and Algorithm `verifySparselss` called subalgorithm "verifySparseGen0". It succeeded in all but one case. As in Part I the reason seems that subalgorithm "verifySparseGen0" performs an unsymmetric equilibration by (1.7).

Secondly, some details on the performance of subalgorithm "verifySparseGen0" for the 211 "gen" test cases plus the 20 tests from [46]. The second call of LDL^T in step 5 was necessary in 53 out of 231 cases due to singularity of the factor D. As explained in Part I of this note there seems room for improvement for the Matlab routine `ldl` for an augmented matrix of type (5.1). With the trick in (1.9) the LDL^T-decomposition never produced a singular D.

The improvement of α in step 12 of subalgorithm "verifySparseGen0" was called in 58 cases, and the second improvement in line 13 was never used in the 231 tests. The decrease of s in step 15 was necessary once.

Algorithm `verifySparselss0` failed once in all 306 test cases including the symmetric positive definite matrices, namely matrix 1247 in [5]. The condition number

of that matrix is $7.6 \cdot 10^{15}$, but the estimate s in Step 5 of "verifySparseSym0" for the smallest singular was $4.5 \cdot 10^{-19}$. This is far too small for a successful verification. In this example even artificially setting s to a value slightly below $\sigma_{\min}(A)$ did not help, the residuals where too large for both Algorithm `verifySparselss` and `verifySparselss0`.

We present some detailed data in Tables 13 - 14. To present all data is too much for this note, so we put the results for all 284 test cases at the `url` in (9.2).

(9.2) `https://www.tuhh.de/ti3/rump/sparselssAllResultsII.pdf`

Here NaN in the columns for the relative error indicate failure of verification, the sixth column displays the ratio $\varrho = t_{new0}/t_{new}$. A ratio $\varrho > 1$ indicates that Algorithm `verifySparselss` of Part I of this note is faster than `verifySparselss0` presented here. Otherwise, the columns are self-explaining.

In order to reduce space for the results to be displayed in this note, we considered the 20 tests in [46] together with the 264 examples in (9.1) satisfying all properties listed in Table 12. That fills 2 pages of computational results; all results can be found

TABLE 12
Displayed tests extracted from the 306 *tests in Table 11.*

- all tests where "new" failed
- all tests where "new0" failed
- all tests where the maximal relative error by "new" is larger than 10^{-15}
- all tests where the maximal relative error by "new0" is larger than 10^{-15}
- all tests where the computing time ratio t_{new0}/t_{new} is larger than 1.64

at the `url` in (9.2). The curios ratio 1.64 of computing time t_{new0}/t_{new} is tuned to fill 2 pages of results. The horizontal lines separate symmetric, general, [46], Hermitian positive definite and general complex matrices.

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 estimated condition number over the 100 tests was $4.0 \cdot 10^{15}$. The results of this test are reported in Table 15.

The median condition number $4.0 \cdot 10^{15}$ of our samples is boarder line in the sense that a verification algorithm might just succeed to compute verified bounds. Still, "new" succeeds in 96 cases "new0" succeeds in all cases. For randomly generated examples there is not much difference in the accuracy of the bounds, but "new" is mostly more than twice as fast as "new0". In Figure 6 we show the ratio of computing times of Algorithm `verifySparselss0` divided by that of Algorithm `verifySparselss`. Algorithm "new" from Part I of this note is always faster than "new0". As explained before that is related to the number of nonzero elements of the matrices L_1.

We tested Algorithm `verifySparselss0` for complex data as well. Some data is shown in the `url` in (9.2). As there were no surprises we refrain, as in Part I of this note, from extending our already shown computational data.

Next we show computational results for rectangular input matrix. As has been mentioned, Matlab chooses to minimize the number of nonzero elements of the solution rather than computing A^+b. Therefore we show only data for least squares problems. Since the matrix in (6.2) is a permutation of that in (6.1) this is gives information of

TABLE 13
Timing and accuracy for sparse linear systems in [5] satisfying the conditions in (9.1).

matrix				*times*		*relerr new*		*relerr news*	
#	n	$nnz(A)$	cnd	$t_{\texttt{new}}$	ϱ	*median*	*max*	*median*	*max*
2742	1180	19674	5.9e12	0.11	1.70	3.7e-17	1.0e-16	3.7e-17	1.0e-16
2672	3543	38136	7.4e10	0.12	1.28	3.7e-17	1.6e-15	3.7e-17	1.5e-15
2221	10798	608540	7.0e14	13.92	2.04	3.6e-17	1.1e-16	3.6e-17	1.1e-16
1247	12546	140034	7.6e15	24.97	2.53	NaN	NaN	NaN	NaN
1210	20360	509866	8.1e14	387.20	1.16	NaN	NaN	3.8e-17	1.2e-16
1451	20360	509866	8.1e14	375.45	1.13	NaN	NaN	3.7e-17	1.1e-16
1218	35543	128115	4.9e15	1.12	1.16	4.7e-17	6.3e-12	4.7e-17	6.3e-12
243	1080	23094	1.4e12	0.77	2.57	3.8e-17	8.9e-15	3.8e-17	5.3e-15
450	1089	3895	1.6e12	0.03	1.24	3.6e-17	2.0e-15	3.5e-17	5.7e-16
420	1409	42760	2.3e13	0.13	1.26	5.0e-17	4.4e-14	4.9e-17	1.8e-14
893	1650	7419	5.6e12	0.04	0.93	5.9e-17	1.3e-10	4.5e-17	4.7e-12
1012	1813	11246	8.0e11	0.11	1.74	3.8e-17	1.1e-16	3.8e-17	1.1e-16
465	2904	58142	3.5e12	0.78	1.97	2.8e-17	1.5e-16	2.8e-17	1.5e-16
439	3096	90841	1.1e11	1.92	2.09	3.7e-17	1.1e-16	3.7e-17	1.1e-16
837	5308	22680	1.7e14	0.18	1.59	3.9e-17	5.9e-13	3.8e-17	4.7e-14
1536	5308	22592	1.7e14	0.17	1.43	3.9e-17	2.5e-13	3.7e-17	2.0e-14
818	6316	167178	4.5e14	2.67	2.05	3.6e-17	1.1e-16	3.6e-17	1.1e-16
568	6774	33744	1.4e14	0.68	1.74	4.0e-17	8.0e-10	4.0e-17	1.1e-10
934	7055	30082	1.7e12	1.06	0.99	NaN	NaN	3.7e-17	1.1e-16
446	7320	324772	3.3e10	8.35	1.66	3.7e-17	1.1e-16	3.7e-17	1.1e-16
920	7500	283992	7.0e11	34.48	2.03	3.7e-17	1.1e-16	3.7e-17	1.1e-16
1395	7548	834222	8.3e12	33.81	1.79	3.7e-17	1.1e-16	3.7e-17	1.1e-16
1170	8081	13036	8.2e10	0.12	1.73	2.2e-26	1.1e-16	6.3e-27	1.1e-16
2814	8256	109368	2.1e15	11.91	1.70	3.6e-17	1.1e-16	3.6e-17	1.1e-16
580	9129	52883	1.7e14	4.18	1.86	3.7e-17	1.1e-16	3.7e-17	1.1e-16
581	9129	52883	7.5e13	4.14	1.71	3.7e-17	1.1e-16	3.7e-17	1.1e-16
741	10672	232633	2.3e14	3.54	1.51	3.7e-17	7.4e-14	3.7e-17	2.7e-14
743	10964	233741	1.3e15	6.76	1.81	3.7e-17	1.1e-16	3.7e-17	1.1e-16
921	11532	551184	6.5e12	176.83	2.09	3.7e-17	1.1e-16	3.7e-17	1.1e-16
550	11790	107383	2.8e13	8.14	1.97	3.5e-17	1.1e-16	3.5e-17	1.1e-16
372	12127	48137	3.5e10	0.40	1.73	2.0e-17	1.1e-16	2.0e-17	1.1e-16
461	13436	71594	3.8e15	0.91	1.53	4.1e-17	5.1e-12	4.0e-17	1.9e-12
570	13694	72734	1.3e14	3.49	1.90	3.6e-17	1.1e-16	3.6e-17	1.1e-16
551	14760	145157	6.7e13	13.98	2.02	3.5e-17	1.1e-16	3.5e-17	1.1e-16
922	16428	948696	4.2e13	454.32	2.10	3.7e-17	1.3e-13	3.6e-17	1.7e-14
747	17576	381975	1.2e15	28.01	0.81	3.8e-17	3.6e-13	3.8e-17	1.9e-13
553	17730	183325	2.4e13	25.34	2.00	3.5e-17	1.1e-16	3.5e-17	1.1e-16
582	18289	106803	3.6e14	12.90	1.88	3.7e-17	1.1e-16	3.7e-17	1.1e-16
583	18289	106803	5.1e13	12.43	1.83	3.7e-17	1.1e-16	3.7e-17	1.1e-16
431	19716	227872	8.8e12	6.60	1.51	4.4e-17	3.1e-12	3.9e-17	1.0e-13
572	20614	111903	3.9e14	6.52	2.00	3.6e-17	1.1e-16	3.6e-17	1.1e-16
1376	20640	97353	2.0e15	0.94	1.73	3.0e-17	1.1e-16	3.0e-17	1.1e-16
555	23670	259648	3.2e13	46.14	2.02	3.5e-17	1.1e-16	3.5e-17	1.1e-16
1111	25187	193216	2.0e14	4.51	1.77	3.7e-17	1.1e-16	3.7e-17	1.1e-16
584	27449	160723	6.4e14	23.46	1.90	3.8e-17	1.1e-16	3.8e-17	1.1e-16
585	27449	160723	5.1e13	23.23	1.84	3.6e-17	1.1e-16	3.6e-17	1.1e-16

TABLE 14
Timing and accuracy for sparse linear systems in [5] satisfying the conditions in (9.1).

matrix				*times*		*relerr new*		*relerr news*	
#	n	$nnz(A)$	cnd	$t_{\texttt{new}}$	ϱ	*median*	*max*	*median*	*max*
574	27534	151063	6.3e14	12.09	2.45	3.6e-17	1.1e-15	3.5e-17	6.0e-16
557	29610	335972	2.6e13	58.95	2.07	3.5e-17	1.1e-16	3.5e-17	1.1e-16
576	34454	190224	9.4e14	16.48	2.45	4.0e-17	2.3e-10	3.9e-17	8.5e-11
559	35550	412306	3.6e13	84.16	2.08	3.5e-17	1.1e-16	3.5e-17	1.1e-16
586	36609	214643	1.1e15	34.72	1.89	3.7e-17	1.1e-16	3.7e-17	1.1e-16
587	36609	214643	5.8e13	33.61	1.87	3.7e-17	1.8e-16	3.7e-17	1.1e-16
1316	37261	443573	2.2e10	43.49	1.86	3.7e-17	1.1e-16	3.7e-17	1.1e-16
1371	39899	195429	2.8e15	2.33	2.17	3.5e-17	1.1e-16	3.5e-17	1.1e-16
2815	40816	803978	4.4e13	747.99	3.24	3.5e-17	1.1e-16	3.5e-17	1.1e-16
578	41374	229385	1.6e15	30.19	2.69	3.7e-17	5.2e-12	3.6e-17	1.5e-13
561	41490	488633	3.4e13	130.49	2.10	3.5e-17	1.1e-16	3.5e-17	1.1e-16
588	45769	268563	1.7e15	47.45	1.78	3.7e-17	4.2e-14	3.7e-17	9.4e-16
589	45769	268563	5.8e13	45.09	1.87	3.7e-17	4.3e-13	3.7e-17	2.3e-14
563	47430	564952	1.1e14	130.93	2.09	3.5e-17	1.1e-16	3.5e-17	1.1e-16
1413	49702	333029	1.5e15	41.20	1.52	3.7e-17	1.7e-13	3.7e-17	3.4e-14
1414	49702	332807	4.0e13	17.53	1.38	3.7e-17	1.1e-16	3.7e-17	1.1e-13
1375	51032	247528	2.3e15	3.50	2.22	3.5e-17	1.1e-16	3.5e-17	1.1e-16
983	51993	380415	9.4e15	222.05	1.80	2.9e-17	1.1e-16	2.9e-17	1.1e-16
565	53370	641290	5.8e13	162.11	2.07	3.5e-17	1.1e-16	3.5e-17	1.1e-16
590	54929	322483	3.6e15	60.60	1.90	3.7e-17	2.8e-11	3.7e-17	8.2e-13
591	54929	322483	5.8e13	59.22	1.83	3.7e-17	2.4e-10	3.7e-17	9.4e-12
567	59310	717620	1.4e14	219.13	2.07	3.5e-17	1.1e-16	3.5e-17	1.1e-16
592	64089	376395	7.9e15	73.48	1.91	3.7e-17	7.3e-9	3.7e-17	9.9e-10
593	64089	376395	5.8e13	72.98	1.92	3.7e-17	1.0e-7	3.7e-17	1.3e-9
373	80209	307604	5.7e11	8.25	2.23	3.4e-17	6.9e-11	3.3e-17	5.8e-11
1374	87190	606489	2.0e15	14.38	2.20	3.7e-17	1.1e-16	3.7e-17	1.1e-16
2657	87936	593276	4.1e10	14.48	2.04	3.6e-17	1.1e-16	3.6e-17	1.1e-16
1343	94294	476766	5.6e12	13.86	2.75	3.7e-17	1.1e-16	3.7e-17	1.1e-16
1344	94294	479246	5.6e12	20.09	2.09	3.7e-17	7.5e-15	3.7e-17	5.2e-15
1345	94294	479151	5.6e12	16.12	2.98	3.7e-17	1.0e-13	3.7e-17	6.1e-14
2567	40948	412148	9.4e10	5.40	1.82	3.7e-17	4.0e-16	3.7e-17	3.6e-16
288	14734	95053	9.6e3	3.56	2.53	3.8e-17	5.8e-16	3.8e-17	4.8e-16
289	25228	175027	5.3e3	6.61	1.67	3.7e-17	1.1e-16	3.7e-17	1.1e-16
290	84617	463625	7.5e4	19.04	2.16	3.7e-17	1.1e-16	3.7e-17	1.1e-16
2820	6005	182168	4.9e5	0.77	1.30	3.7e-17	7.5e-14	3.5e-17	1.1e-14
2821	10142	312814	1.1e6	1.61	1.11	3.6e-17	1.1e-16	3.8e-17	3.9e-15
2824	56021	1797934	1.4e7	23.89	1.65	3.6e-17	1.1e-16	3.6e-17	1.1e-16
2825	100037	3226066	3.4e7	55.05	1.73	3.7e-17	1.1e-16	3.7e-17	1.1e-16
2826	178437	5778545	8.2e7	135.27	1.80	3.7e-17	1.1e-16	3.7e-17	1.1e-16
1415	99340	940621	1.5e11	21.28	2.21	3.7e-17	1.1e-16	3.7e-17	1.1e-16
1417	321821	1931828	5.1e22	101.00	5.18	3.7e-17	2.6e-14	3.7e-17	2.7e-14
1419	682862	2638997	9.5e19	746.79	2.49	9.2e-17	6.9e-8	9.4e-17	7.2e-8
326	2534	463360	5.2e5	20.31	1.66	3.6e-17	1.1e-16	3.6e-17	1.1e-16
1407	10605	522387	1.0e15	51.05	1.87	NaN	NaN	3.6e-17	2.7e-16
2555	37365	330633	2.7e5	57.70	1.90	3.7e-17	1.1e-16	3.7e-17	1.1e-16
2556	90249	803173	3.2e5	207.38	1.98	3.7e-17	1.1e-16	3.7e-17	1.1e-16

TABLE 15
Results for 100 *randomly generated ill-conditioned test cases.*

	"new"	"new0"
inclusions	failed in 4 out of 100 tests	failed in 0 out of 100 tests
median relative error	$3.7 \cdot 10^{-17}$	$3.7 \cdot 10^{-17}$
maximal relative error	$5.0 \cdot 10^{-4}$	$1.2 \cdot 10^{-4}$

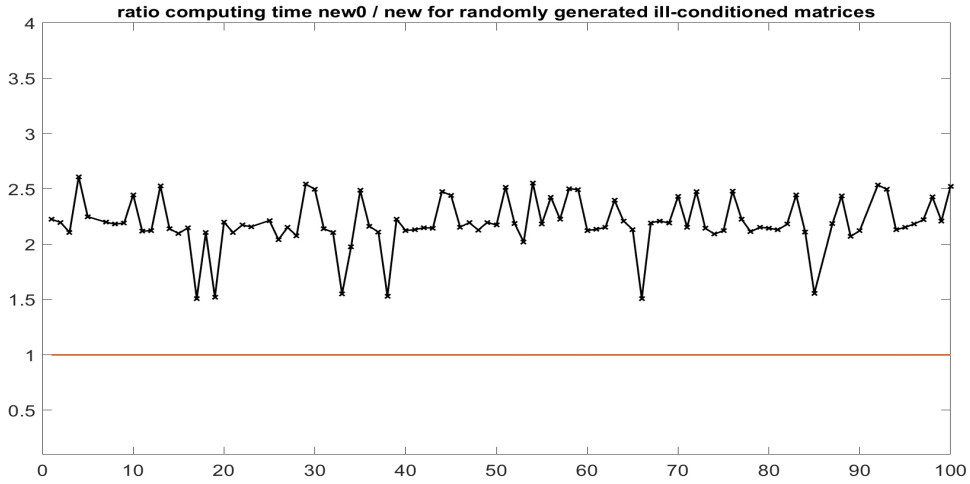

FIG. 6. *Ratios of computing times* $t_{\texttt{verifySparselss0}}/t_{\texttt{verifySparselss}}$.

the underdetermined cases as well. If a test matrix A in [5] has more columns than rows we use A^H.

We use all matrices from the Suite Sparse Matrix Collection [5] with dimensions

$$10^3 \leqslant m, n \leqslant 10^5 \quad \text{and} \quad 10^7 \leqslant cnd \leqslant 10^{16} \quad \text{and} \quad \texttt{nnz(A)} \leqslant 10^6 \ . \tag{9.3}$$

The condition number of a rectangular matrix with respect to a least squares problem is a bit tricky. Here *cnd* denotes the estimated condition number of the augmented matrix in (6.1).

There were no complex examples in [5] satisfying (9.3). The conditions in (9.3) lead to 26 test cases because most of the examples where either well-conditioned or extremely ill-conditioned, often with condition number ∞. The results are displayed in Table 16. As can be seen Algorithm `verifySparselss` failed for the two cases 1950 and 2026 of [5], Algorithm `verifySparselss0` failed only for the last case 2026.

There is not too much difference in computing times for "new" and "new0". In the median the computing times are almost the same, in the worst case "new" is 2.2 times faster than "new0", and "new0" is 1.4 times faster than "new".

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 the majority of diagonal elements of the augmented matrix (6.1) are already nonzero.

There is quite a spread in computing time between `lu` and our new algorithms, and surprisingly Matlab's `lu` is in general slower than our verification. In the median "new" is 1.7 times faster than `lu`, and in the worst case "new" 335 times faster than

`lu`. However, `lu` may be also up to 279 times faster than "new".

Both Algorithms "new" and "new0" compute always inclusions with maximal accuracy for all entries of the solution. In contrast, the approximations by Matlab's `lu` are significantly less accurate. The median and maximum relative errors of the approximation by `lu` and "verifySparselss0" are displayed in Figure 7. As can be seen in the median some 13 figures of the approximation by `lu` are correct, but in one case only 6 digits of at least one entry of the approximation. In contrast, "verifySparselss0" (and also "verifySparselss") compute almost always maximally accurate inclusions for all entries.

Out of the ill-conditioned test cases satisfying (9.3) there were 37 matrices with zero columns. That implies that the matrix is rank-deficient. When deleting those 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 [23]. The source code for 23 examples can be found 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 example $p09$, $p13$ and $p14$ we list computational results for different dimensions.

The results for Algorithm "verifySparseNlss" are shown in Table 17. We compare three algorithm. The first is "verifySparseNlss" listed in Table 9 and called "new" in Table 17. It calls the modified Algorithm "verifySparselss" as in Table 6 in Part I of this note to solve the linear system with interval right hand side. Secondly, we use Algorithm "verifySparselss0" as in Table 10 as the linear system solver. In Table 17 it is called "new0". As a third algorithm we use the built-in Matlab routine `fsolve`.

The columns are self-explaining except "iter" for "new" and "new0" which displays the number `kxs` of (approximate) Newton iterates and the number `kY` of interval iterates.

All routines have as input parameters a reference to the function in use as well as an initial approximation. For the functions $p09$, $p13$ and $p14$ we use the starting values specified in [23]. Except for $p09$ and AB we treat dimensions from 10^3 to 10^7.

INTLAB contains Algorithm `verifynlss` for solving systems of nonlinear equations. It is based on Theorem 7.2 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.

To further investigate the performance of our algorithms, we consider two other examples with specifiable dimension. The first [22], abbreviated by MC, is a discretization of

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

and initial approximation $x_k = t_k(t_k - 1)$ for $t_k = k/(n+1)$. The second example [1], abbreviated by AB, is a discretization of

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

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

TABLE 16
Timing and accuracy for sparse linear systems in [5] satisfying the conditions in (9.1).

	matrix				*times [sec]*			*relerr* `lu`		*relerr new*		*relerr new0*	
# matrix	*m*	*n*	*nnz(A)*	*cnd*	$t_{\texttt{lu}}$	t_{new}	t_{new0}	*median*	*max*	*median*	*max*	*median*	*max*
155	10595	4929	46591	2.1e12	0.194	0.304	0.314	1.7e-13	3.6e-9	3.9e-17	1.1e-16	3.9e-17	1.1e-16
615	5831	2171	33081	2.5e7	0.146	0.333	0.379	3.0e-15	8.6e-12	3.9e-17	1.1e-16	3.9e-17	1.1e-16
628	1706	1309	6937	3.7e8	0.020	0.046	0.050	2.9e-14	6.3e-11	3.9e-17	1.1e-16	3.9e-17	1.1e-16
697	23541	16675	72721	4.0e7	70.895	2.299	3.159	3.1e-14	4.0e-10	3.9e-17	1.1e-16	3.9e-17	1.1e-16
981	29493	11822	117954	1.7e10	21.635	10.895	13.520	1.7e-9	3.2e-7	3.9e-17	1.1e-16	3.9e-17	1.1e-16
1708	38602	24617	156466	1.3e9	160.976	496.919	1077.091	3.1e-14	8.8e-9	3.8e-17	1.1e-16	3.8e-17	1.1e-16
1713	16369	10099	44825	1.7e10	8.240	0.717	0.819	4.7e-15	1.5e-10	3.8e-17	1.1e-16	3.8e-17	1.1e-16
1731	16819	4400	150372	4.5e7	0.330	1.537	1.730	1.4e-14	1.3e-10	3.8e-17	1.1e-16	3.8e-17	1.1e-16
1737	8734	2301	68225	4.3e7	0.586	0.442	0.448	5.2e-15	1.4e-11	3.6e-17	1.1e-16	3.6e-17	1.1e-16
1750	7967	2095	19826	6.4e8	0.766	0.073	0.089	1.2e-15	3.8e-13	3.9e-17	1.1e-16	3.9e-17	1.1e-16
1756	1900	1650	8897	3.8e7	0.036	0.053	0.066	2.4e-14	3.2e-11	3.8e-17	1.1e-16	3.8e-17	1.1e-16
1775	63076	3173	491336	4.4e7	19.545	1.879	2.179	1.1e-13	3.9e-10	3.8e-17	1.1e-16	3.8e-17	1.1e-16
1779	46937	6590	164538	4.3e11	49.096	2.944	3.733	2.6e-15	3.7e-12	3.8e-17	1.1e-16	3.8e-17	1.1e-16
1816	6654	3170	15397	5.7e7	0.007	0.095	0.139	6.6e-15	4.4e-11	3.8e-17	1.1e-16	3.8e-17	1.1e-16
1818	8617	4282	20635	1.1e8	0.014	0.215	0.213	8.9e-15	2.1e-11	3.9e-17	1.1e-16	3.9e-17	1.1e-16
1827	13847	9743	35885	6.1e7	2.859	0.267	0.256	1.2e-15	4.7e-11	3.6e-17	1.1e-16	3.6e-17	1.1e-16
1829	46679	32847	120141	6.5e8	343.471	1.665	1.857	1.9e-15	1.8e-11	3.9e-17	1.1e-16	3.9e-17	1.1e-16
1834	27691	14364	58439	5.7e8	163.160	0.487	0.599	1.8e-15	1.2e-11	3.9e-17	1.1e-16	3.9e-17	1.1e-16
1870	26722	11028	102432	1.6e7	35.288	0.774	0.958	1.2e-12	6.7e-9	3.9e-17	1.1e-16	3.9e-17	1.1e-16
1871	14318	11028	57376	6.3e7	33.257	0.478	0.697	1.5e-11	3.9e-7	3.9e-17	1.1e-16	3.9e-17	1.1e-16
1872	28634	11028	115262	1.6e7	35.063	0.791	1.030	1.2e-11	3.4e-7	3.8e-17	1.1e-16	3.8e-17	1.1e-16
1947	1302	1121	11185	5.0e7	0.030	0.592	0.682	2.2e-13	4.2e-11	3.9e-17	1.0e-16	3.9e-17	1.0e-16
1948	3160	2644	29862	3.1e8	0.140	6.614	8.105	2.7e-12	1.4e-7	3.7e-17	1.1e-16	3.7e-17	1.1e-16
1949	7742	6334	80057	2.3e9	1.093	305.128	217.607	2.3e-11	1.9e-8	3.9e-17	1.1e-16	3.9e-17	1.1e-16
1950	19321	15437	216173	1.5e10	15.374	6051.139	2005.582	9.7e-11	1.1e-6	failed		3.8e-17	1.1e-16
2026	15120	5040	30240	9.0e14	1.062	33.085	82.147	?	?	failed		failed	

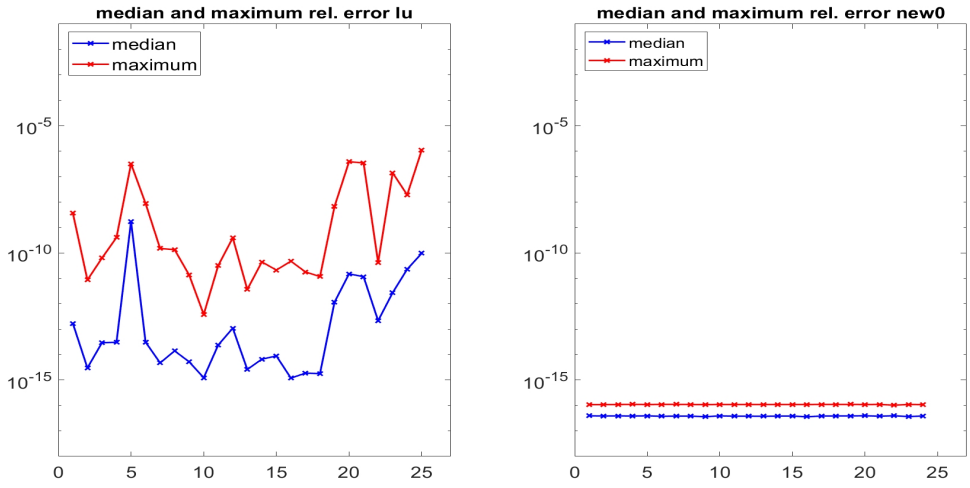

FIG. 7. *Median of relative errors of* `verifySparselss` *and* `verifySparselss0`.

The results for Algorithm "verifySparseNlss" are shown in Table 17. We compare three algorithm. The first is "verifySparseNlss" listed in Table 9 and called "new" in Table 17. It calls the modified Algorithm "verifySparselss" as in Table 6 in Part I of this note to solve the linear system with interval right hand side. Secondly, we use Algorithm "verifySparselss0" as in Table 10 as the linear system solver. In Table 17 it is called "new0". As a third algorithm we use the built-in Matlab routine `fsolve`.

The columns are self-explaining except "iter" for "new" and "new0" which displays the number `kxs` of (approximate) Newton iterates and the number `kY` of interval iterates.

All routines have as input parameters a reference to the function in use as well as an initial approximation. For the functions $p09$, $p13$ and $p14$ we use the starting values specified in [23]. Except for $p09$ and AB we treat dimensions from 10^3 to 10^7.

INTLAB contains Algorithm `verifynlss` for solving systems of nonlinear equations. It is based on Theorem 7.2 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`. As can be seen in Table 17, Algorithm `fsolve` computes an approximation for dimensions up to 10^4; for larger dimensions it fails with error "out of memory". For 1000 unknowns and problem "MC" the approximation is in the median accurate to some 7 decimal digits, for problem $p09$ only one figure is correct.

Our algorithms for a nonlinear system with sparse Jacobian work successfully up dimension 10^7. For the problems $p13$ and $p14$, "new" based on the linear system solver "verifySparselss" in Part I of this note computes verified bounds successfully for $n \leqslant 10^7$, while "new0" based on "verifySparselss0" presented in this note fails for problems $p13$ and $p14$ and dimension $n = 10^7$. Moreover, "new0" is much slower than "new" for problem $p14$.

TABLE 17
Timing and accuracy for systems of nonlinear equations with sparse Jacobian.

	problem		*times [sec]*			*iter and relerr new*				*iter and relerr new0*				*relerr* `fsolve`	
	n	*cond*	t_{new}	t_{new0}	$t_{\texttt{fsolve}}$	*iter*	*median*	*max*	$\|\cdot\|_2$	*iter*	*median*	*max*	$\|\cdot\|_2$	*median*	*max*
*p*09	1,000	3.9e5	0.1	0.0	0.2	5/2	9.3e-10	4.8e-7	4.8e-10	5/2	9.3e-10	4.8e-7	4.8e-10	0.19	0.33
	10,000	3.9e7	0.1	0.2	4.9	5/2	2.9e-7	1.5e-3	1.5e-7	5/3	2.9e-7	1.5e-3	1.5e-7	0.33	0.50
	100,000	3.9e9	1.7	1.6		15/3	9.9e-5	1.5	5.1e-5	15/2	9.9e-5	1.5	5.1e-5	out of memory	
*p*13	1,000	3.5	0.1	0.1	0.3	6/3	1.1e-14	2.7e-14	5.5e-15	6/2	8.8e-15	2.1e-14	4.4e-15	8.5e-15	8.2e-14
	10,000	3.5	0.2	0.2	20.7	6/2	3.0e-14	7.5e-14	1.5e-14	6/2	2.4e-14	6.0e-14	1.2e-14	6.0e-13	1.9e-11
	100,000	3.5	1.6	2.1		6/2	9.5e-14	2.4e-13	4.8e-14	6/2	7.6e-14	1.9e-13	3.8e-14	out of memory	
	1,000,000	3.5	17.5	31.3		6/2	3.0e-13	7.4e-13	1.5e-13	6/3	2.4e-13	5.9e-13	1.2e-13	out of memory	
	10,000,000	3.5	208.9	93.6		6/2	9.4e-13	2.3e-12	4.7e-13	6/3		failed		out of memory	
*p*14	1000	2.2	0.7	0.3	1.2	7/2	1.5e-14	3.1e-14	7.6e-15	7/2	1.6e-14	3.3e-14	7.9e-15	8.6e-15	7.0e-9
	10,000	2.2	0.8	0.8	127.3	7/3	1.2e-13	2.6e-13	6.2e-14	7/2	4.7e-14	1.0e-13	2.4e-14	6.2e-14	1.3e-13
	100,000	2.2	16.2	48.1		7/3	3.8e-13	8.0e-13	1.9e-13	7/3	3.8e-13	8.0e-13	1.9e-13	out of memory	
	1,000,000	2.2	152.6	3408.7		7/2	1.2e-12	2.6e-12	6.0e-13	7/3	1.2e-12	2.5e-12	6.0e-13	out of memory	
	10,000,000	2.2	7224.1	0.5		7/2	3.8e-12	8.0e-12	1.9e-12	7/3		failed		out of memory	
MC	1,000	3.9e5	1.7	0.1	0.4	4/3	5.0e-12	5.1e-6	2.6e-12	4/2	3.9e-12	4.0e-6	2.0e-12	1.4e-3	1.7e-3
	10,000	3.9e7	0.2	0.5	24.5	4/2	2.5e-9	0.23	1.3e-9	4/3	2.0e-9	0.19	1.0e-9	1.1e-3	0.13
	100,000	3.9e9	2.5	13.6		4/3	2.6e-7	2.0	1.3e-7	4/2	2.6e-7	2.0	1.3e-7	out of memory	
	1,000,000	3.9e11	23.6	3844.2		5/3	8.5e-6	2.0	4.4e-6	5/4	6.9e-6	2.0	3.5e-6	out of memory	
	10,000,000	∞	543.5	0.2		11/4		failed		5/3		failed		out of memory	
AB	100	5.7e3	0.1	0.1	0.0	10/3	1.0e-14	2.1e-13	4.8e-15	10/2	6.0e-15	1.2e-13	2.8e-15	1.1e-12	1.2e-11
	1,000	5.3e5	0.1	0.1	0.8	12/2	2.6e-13	3.0e-11	1.2e-13	12/2	1.4e-13	1.6e-11	6.4e-14	2.1e-9	2.2e-7
	5,000	1.3e7	0.5	0.5		14/5	2.4e-12	8.9e-10	1.1e-12	14/3	1.2e-12	4.4e-10	5.4e-13	out of memory	
	7,000	2.6e7	0.6	0.6		14/4	2.6e-12	1.3e-9	1.3e-12	14/3	1.3e-12	6.2e-10	5.9e-13	out of memory	
	10,000	5.3e7	2.0	2.2		15/10		failed		15/8	2.7e-12	3.5e-9	1.3e-12	out of memory	
	15,000	1.2e8	142.1	144.5		15/10		failed		15/8		failed		out of memory	

Contrary, for problem AB "new0" is successful for larger dimensions than "new". For problem AB, with increasing dimension the increasing difficulty of "new0" to compute verified bounds can be seen in Table 17. The number `kxs` of approximate Newton iterates increases to the limit, and eventually also the number of interval iterations. As explained in Section 7 the verified solution of a system of nonlinear equations requires to solve a sparse linear system with interval matrix and interval right hand side. That limits the range of applicability.

Finally note that the median and maximal entrywise relative errors over all entries of the solution are displayed. Naturally, these become weak for entries small in absolute value. We may also judge an inclusion as the error $\delta \in \mathbb{R}_+$ of an approximation $\tilde{x}$ in 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 17. That ratio does not exceed 10^{-5} for all examples.

10. Conclusion and an open problem. In this Part II of our note we discussed a second Algorithm for computing verified error bounds for a linear system with sparse input matrix. The bounds are correct with mathematical certainty including the proof of nonsingularity of the input matrix. As the method in Part I it is applicable to real and complex data including data afflicted with tolerances.

The second algorithm is usually slower than the first one presented in Part I of this note, but seems a little more stable. Our methods are usually slower than Matlab's built-in solver `lu`, but sometimes faster by two orders of magnitude.

Moreover, 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 is often sparse. In that case our method is superior to existing algorithms such as Algorithm `verifynlss` in INTLAB. Computational tests show that the new method is successful on our small laptop for dimensions up to 10^7.

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 to obtain even more accurate error bounds for the solution of linear systems such that almost always error bounds with maximal accuracy are delivered for all entries.

The methods in Part I and II of this note are based on a matrix decomposition. There are numerous iterative methods to compute an approximation of a sparse linear system, and many people think that is at least an attractive way to attack sparse systems. These approximations may be used for a verification method, but the computation of rigorous bounds based on an iterative method is completely open. There are error estimates, but those are qualitative and/or theoretical and not computable. Up to now some factorization is the only way for the step from a small residual to a verified inclusion.

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, 2019.

[13] IEEE Standard for Floating-point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[14] D. E. Knuth: *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, 1969.

[15] 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.

[16] 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.

[17] 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.

[18] M. Lange, S.M. Rump. Sharp estimates for perturbation errors in summations. *Math.Comp.*, 88:349–368, 2019.

[19] M. Lange and S.M. Rump. Floating-point matrix products with improved accuracy part I: theoretical background. to appear.

[20] M. Lange and S.M. Rump. Floating-point matrix products with improved accuracy part II: Schemes for matrix products. to appear.

[21] MATLAB. User's Guide, Version 2023b, the MathWorks Inc., 2023.

[22] J.J. Moré and M.Y. Cosnard. Numerical solution of non-linear equations. *ACM Trans. Math. Software*, 5:64–85, 1979.

[23] 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.

[24] 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.

[25] A. Neumaier. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *Zeitschrift für Angew. Math. Mech. (ZAMM)*, 54:39–51, 1974.

[26] A. Neumaier: *Interval methods for systems of equations*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1990.

[27] A. Neumaier. Grand challenges and scientific standards in interval analysis. *Reliable Computing*, 8(4):313–320, 2002.

[28] T. Ogita, S. M. Rump, S. Oishi: Accurate sum and dot product. *SIAM Journal on Scientific Computing (SISC)*, 26(6):1955–1988, 2005.

[29] 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].

[30] 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.

[31] 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.
[32] 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.
[33] 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.
[34] 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.
[35] 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.
[36] S.M. Rump. *Kleine Fehlerschranken bei Matrixproblemen*. PhD thesis, Universität Karlsruhe, 1980.
[37] 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.
[38] S.M. Rump. Verified Computation of the Solution of Large Sparse Linear Systems. *Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, 75:S439–S442, 1995.
[39] S. M. Rump: INTLAB – INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Springer Netherlands, Dordrecht, 1999.
[40] 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.
[41] S. M. Rump: Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.
[42] S.M. Rump. Improved componentwise verified error bounds for least squares problems and underdetermined linear systems. 66:309–322, 2013.
[43] 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).
[44] 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.
[45] R. Skeel. Iterative Refinement Implies Numerical Stability for Gaussian Elimination. *Math. Comp.*, 35(151):817–832, 1980.
[46] Terao T., K. Ozaki. Method for verifying solutions of sparse linear systems with general coefficients. 2024. `https://arxiv.org/abs/2406.02033`.
[47] G. Zielke, V. Drygalla. Genaue Lösung linearer Gleichungssysteme. *GAMM Mitt. Ges. Angew. Math. Mech.*, 26:7–108, 2003.