# 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 several efforts, the solution of general sparse linear systems is 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 Jacobi matrix. In all cases the algorithms for 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).



*This manuscript is for review purposes only.*

47 based on some $LDL^T$-decomposition. A key to a successful verification method is to
48 compute accurate of residuals, here $\|A - L_1 L_2\|_2$. The advantage of the splitting into
49 two factors is that each entry of the residual $A - L_1 L_2$ is a dot product, so that fast
50 and accurate methods to compute accurate bounds for the residual norm can be used.
51 The methods in Part I and II explore the ideas in [37, 38, 40] published in the
52 1990's. For the time being the algorithms for $LDL^T$-decomposition were not stable
53 enough to allow for good verification methods. Nowadays good scaling and equilibra-
54 tion routines are available [8, 9] making those methods attractive. That was observed
55 by Terao and Ozaki [46] and triggered both parts of this note.
56 One key of our methods is the following theorem [38, Theorem 1.1]:

57 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*
58 *given. If the inertia of* $\tilde{D}_1$ *and* $\tilde{D}_2$ *are equal, then for any matrix norm*

59 (1.1) $$\sigma_{\min}(A) > \tilde{\lambda} - \max\{\|A - \tilde{\lambda}I - \tilde{L}_1\tilde{D}_1\tilde{L}_1^T\|, \|A + \tilde{\lambda}I - \tilde{L}_2\tilde{D}_2\tilde{L}_2^T\|\}.$$

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

61 (1.2) $$\sigma_{\min}(A) > \tilde{\lambda} - \|A - \tilde{\lambda}I - \tilde{L}_1\tilde{D}_1\tilde{L}_1^T\|.$$

62 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
63 $k \in \{1, 2\}$. We use "tilde" to indicate that approximate factorizations are used.
64 An application to symmetric (positive definite) $A$ sets $\tilde{G} := \tilde{L}_1^T$ and $\tilde{D}_1 = I$, such
65 that (1.2) implies

66 (1.3) $$\sigma_{\min}(A) > \tilde{s} - \|A - \tilde{s}I - \tilde{G}^T\tilde{G}\| =: \varrho$$

67 for an approximate Cholesky decomposition $A - \tilde{s}I \approx \tilde{G}^T\tilde{G}$. This certifies a lower
68 bound $\varrho$ of the smallest singular value $\sigma_{\min}(A)$ based on some approximation $\tilde{s}$. If $\varrho$
69 is positive it proves positive definiteness of $A$ as well.
70 That approach for symmetric (positive definite) $A$ was further explored in [43].
71 It is appealing that a priori bounds for $\|A - \tilde{s}I - \tilde{G}^T\tilde{G}\|_2$ are available at practically
72 no cost solely based on the diagonal of $A$. This is based on [6], see also [11, Theorem
73 10.5]. In Lemma 2.5 and Corollary 2.6 in Part I of this note we improve the bound
74 $\varrho$ by using linear estimates on the rounding error of dot products [16, 17, 18] and a
75 special application of Perron-Frobenius Theory.
76 Another application [40, 46] of Theorem 1.1 gives a lower bound on $\sigma_{\min}(A)$ of a
77 general matrix $A$ by using the augmented matrix $B := \begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix}$. The eigenvalues
78 of $B$ are $\pm\sigma_k(A)$ so that the inertia of $B$ is known to be $(-n, 0, n)$ for nonsingular $A$.
79 Hence

80 (1.4) $$\sigma_{\min}(A) = \sigma_{\min}(B) > \tilde{s} - \|B - \tilde{s}I - \tilde{L}\tilde{D}\tilde{L}^T\| =: \varrho$$

81 for an anticipated lower bound $\tilde{s}$ of $\sigma_{\min}(A) = \sigma_{\min}(B)$ is true if $\tilde{D}$ has $n$ positive
82 eigenvalues for an approximate $LDL^T$-decomposition $B - \tilde{s}I \approx \tilde{L}\tilde{D}\tilde{L}^T$. Note that $\varrho > 0$
83 implies that $B$ has full rank and therefore $A$ is nonsingular.
84 If $\sigma_{\min}(A) \geqslant \varrho > 0$, then for an approximate solution $\tilde{x}$ of a linear system $Ax = b$
85 it follows
86 $$\|A^{-1}b - \tilde{x}\|_\infty \leqslant \|A^{-1}b - \tilde{x}\|_2 \leqslant \varrho^{-1}\|b - A\tilde{x}\|_2$$
87 as noted in Part I. However, for ill-conditioned $A$ that bound may be quite some
88 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 $\pm1$ 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 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 $\mathrm{float}(\cdot)$ 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 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 almost always 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. 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.

Numerical examples for test matrices out of [5] as well as for randomly generated matrices 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)));
```
(163)

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].

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 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 $\mu$ determines the splitting and is chosen such

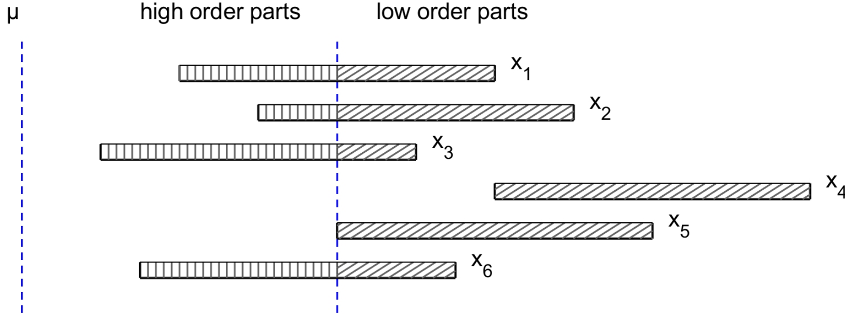


FIG. 1. *The Zielke/Drygalla scheme to extract high and low order parts*

179 that all products $p_i r_i$ and their sum reside in the range of digits of one floating-point
180 number in the given format. That method was analysed in [44] and is also used for
181 reproducible results [2].

182 One specific advantage of the absolute splitting is the applicability to matrix
183 products. The recursive application leads to the following *Ozaki scheme* for the matrix
184 product $AB$ of two floating-point matrices. It was originally published in [30, 34, 35]
185 with improvements in [31, 32]. In the first step $A$ is split into $k+1$ parts

186 (2.1)
$$A = A^{(1)} + A^{(2)} + \ldots + A^{(k)} + \underline{A}^{(k)}$$

187 where each part $A^{(i)}$ holds a limited range of mantissa digits and $\underline{A}^{(k)}$ is the least
188 significant part containing the remainder. A similar splitting is applied to $B$. The
189 ranges for the mantissa digits in $A^{(i)}$ and $B^{(j)}$ are chosen in such a way that all
190 the individual matrix products $A^{(i)}B^{(j)}$ are computed error-free independent of the
191 order of evaluation.[2] Ozaki et al. [34, 33, 32] exploited this by computing $AB$ as the
192 unevaluated sum of $\frac{(k+1)(k+2)}{2}$ individual matrix products

193 (2.2)
$$AB = \sum_{i+j \leqslant k+1} A^{(i)}B^{(j)} + \underbrace{\sum_{i=1}^{k} A^{(i)}\underline{B}^{(k+1-i)} + \underline{A}^{(k)}B}_{\text{remainder terms}}.$$

194 where the sum of these is realized via an accurate summation algorithm, for instance
195 [3, 25, 7, 28, 44]. Then the overall error is determined by the rounding errors in the
196 computation of the $k+1$ remainder terms which are least significant. By using the
197 particular splitting approach proposed in [34], one can expect the error to be roughly
198 of the size $(2n\mathbf{u})^{k/2+1}|A||B|$, where $\mathbf{u}$ denotes the relative rounding error unit. Hence,
199 with increasing $k$ there is a significant increase in the precision.

200 A major advantage of Ozaki's scheme over other approaches for computing ac-
201 curate matrix-matrix products is the efficient use of highly optimized level-3 BLAS
202 routines. For algorithms based on vector transformations, such as `Dot2` [28], reaching

[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.

*This manuscript is for review purposes only.*

203 peak performance is more difficult and requires to perform optimizations by hand. A
204 second benefit of Ozaki's scheme is the relatively low computational complexity for
205 small $k$. The biggest drawback is that the computational complexity and the required
206 memory increase quadratically with $k$.
207 In [19, 20] we discuss various improvements to the original Ozaki scheme. The
208 most important for this note is to specify a precise splitting point. When compared to
209 the original splitting by Ozaki's methods, this yields roughly an additional precision
210 of $k$ digits. Moreover, instead of the infinity norm of the respective column or row
211 vectors, we use the Euclidean norm to determine suitable splitting parameters. This
212 often gives another factor two in precision.
213 The implementation in Algorithm `prodK` is pure Matlab code and due to Marko
214 Lange [19, 20]. Despite the interpretation overhead it is faster than the mex-files used
215 in Advanpix [12]. Timing of `vpa`, `mp` and `prodK` for full matrices is shown in Table
1. As can be seen, for full matrices `vpa` is much slower than `mp`, and for little larger

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 |

216
217 dimension `prodK` is significantly faster than `mp`.
218 For sparse matrices much effort is necessary to ensure an efficient memory man-
219 agement. To that end Marko Lange provided a special implementation `spProdK`.
220 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 |

221
222 For dimension $10,000$ and larger `vpa` stopped with memory problems. However, `vpa`
223 would only be an option to compute accurate approximations, but it is not suitable
224 for verified inclusions because it does not allow the computation of error bounds. The
225 same is true for Advanpix except for extended precision using `mp.Digits(34)`.

226 For prodK and similarly for spProdK typical calls are

227 (2.3)
```
res = prodK(L,U,-1,A,k);                      LU - A ≈ res
[res,err] = prodK(L,U,-1,A,k);                LU - A ∈ res ± err
[res,err] = prodK(A,x,A,y,-1,b,k);            Ax + Ay - b ∈ res ± err
res = prodK(A,x,-1,b,k,'OutputTerms',2);      Ax - b ≈ res{1} + res{2}
```

228 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 com-
229 puted, where the each of the first parameters may be a scalar. For one output pa-
230 rameter $res$ the result will be approximated in about $(k/2+1)$-fold precision. For two
231 output parameters, $res \pm err$ is a correct inclusion, also computed in $(k/2+1)$-fold
232 precision. Finally, $'OutputTerms', m$ specifies that the result is stored in a cell array
233 with $m$ members. That corresponds to an unevaluated sum of $m$ addends.
234 In (1.4) in Part I we introduced a notation for the approximation and inclusion
235 of a residual $Ax - b$ with sample Matlab/INTLAB code in (1.5). Here we extend the
236 notation allowing evaluation in higher precision. The subindices $_{k,1}$ indicate that the
237 expression is evaluated in $k$-fold precision and rounded into working precision. The
238 last parameter $k$ in the calls of prodK and spProdK imply a result "as if" evaluated
239 in $k/2+1$-fold precision. Therefore using spProdK sample Matlab/INTLAB code is

240 (2.4)

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

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

243 (2.5)
```
setround(-1); Q = abs(A*B-C);
setround(+1); Q = max(Q,abs(A*B-C));
beta = NormBnd(Q,symm);
```

244 The second parameter symm in the function NormBnd is chosen to be true if $AB - C$
245 is symmetric/Hermitian. A bound $\|AB - C\|_2 \leqslant \gamma$ computed in higher precision as in
246 (2.17) of Lemma 2.7 in Part I is now replaced by

247 (2.6)
```
[res,err] = spProdK(A,B,-1,C,k);
setround(1)
gamma = NormBnd(abs(res)+err,symm);
```

248 Then $\|AB-C\|_2 \leqslant \gamma$ because the sum abs(res)+err in the last statement is computed
249 in rounding upwards and $\|M\|_2$ is monotone for nonnegative $M$.
250 As explained above we work with a factorization $A \approx L_1 L_2$ so that the entries
251 of the residual $L_1 L_2 - A$ consist of dot products. For ill-conditioned input matrix it
252 might be necessary to compute an upper bound $\alpha$ of the spectral norm of a residual
253 $LDL^T - A$. Here extra care is necessary because now the product of three matrices is
254 involved. The following code in Table 3 computes an upper bound $\alpha$ of $\|LDL^T - A\|_2$.
255
256 The proof of correctness is as follows. The first line yields matrices $C, C_2, E_1$ with

257
$$|C_1 + C_2 - DL^T| \leqslant E_1$$

```
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 $\alpha$ of $\|LDL^T - A\|_2$.*

258 with entrywise absolute value and comparison. The matrix pair $(C_1, C_2)$ approximates
259 $DL^T$ as an unevaluated sum which corresponds to quadruple precision. The matrices
260 $C, E_2$ in the next line satisfy

261
$$|LC_1 + LC_2 - A - C| \leqslant E_2 \ .$$

262 The next line uses Algorithm `NormBnd` from Table 1 in Part I of this note and computes
263 $\alpha_1$ with $\||C| + E_2\|_2 \leqslant \alpha_1$ so that finally

264
$$\begin{aligned} \|LDL^T - A\|_2 &= \|L(DL^T - C_1 - C_2) + C + L(C_1 + C_2) - A - C\|_2 \\ &\leqslant \|L\|_2\|E_1\|_2 + \| \ |C| + |L(C_1 + C_2) - A - C| \ \|_2 \\ &\leqslant \|L\|_2\|E_1\|_2 + \| \ |C| + E_2 \ \|_2 \\ &\leqslant \|L\|_2\|E_1\|_2 + \alpha_1 \end{aligned}$$

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

272 Accurate bounds for matrix residuals are mandatory to compute accurate error
273 bounds for the solution of a linear system. In Section 4 in Part I of this note we
274 introduced in Table 1 the function `ErrorBound`. It stores an approximate solution of
275 $A^{-1}b$ in two parts $\tilde{x}, \tilde{y}$ such that the unevaluated sum $\tilde{x} + \tilde{y}$ produces a small residual
276 $\varrho = \|A\tilde{x} + A\tilde{y} - b\|_2$. The computation of $\varrho$ is very ill-conditioned and requires at least
277 double the working precision. To that end `mp.Digits(34)` is sufficient to improve an
278 approximation and the inclusion.

279 In order to obtain almost always error bounds close to maximal accuracy for all
280 entries of the solution, we follow [36] and store an approximation in three parts $\tilde{x}, \tilde{y}, \tilde{z}$.
281 Then the residual $\varrho = \|A\tilde{x} + A\tilde{y} + A\tilde{z} - b\|_2$ is even more ill-conditioned. Using twice
282 the working precision is not sufficient, i.e., when using `mp.Digits(34)` there would
283 be no improvement whether using two or three parts for the approximation.

284 A higher precision can be specified in `mp`, however, there is not enough information
285 about the arithmetic in use to compute valid error bounds. In contrast, higher pre-
286 cision can be specified in `prodK` and `spProdK` to compute an accurate approximation
287 and with the possibility to obtain verified error bounds. For example, an inclusion of
288 $\|A\tilde{x} + A\tilde{y} + A\tilde{z} - b\|_2$ is computed by

289
```
[c,e] = spProdK(A,xs,A,ys,A,zs,-1,b,k)
```

290 implying that

$$|A\tilde{x} + A\tilde{y} + A\tilde{z} - b - c| \leqslant e \quad (291)$$

292 is satisfied for all entries. The parameter $k$ specifies that $(k/2+1)$-fold precision is
293 used. For an approximation in three parts $k = 4$ corresponding to 3-fold precision
294 is suitable. This leads to an improved and very accurate version `ErrorBound3` of
295 Algorithm `ErrorBound` in Table 1 in Part I of this note. Algorithm `ErrorBound3` is
296 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 | $\delta = |\tilde{y}| + \|\varrho\|_\infty / s$ | |

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

297
298 The proof of correctness is as for `ErrorBound` in Part I of this note because only
299 the approximation was changed from $\tilde{x} + \tilde{y}$ to three parts $\tilde{x} + \tilde{y} + \tilde{z}$. Of course it
300 is possible to split the approximation into an unevaluated sum of even more parts,
301 where increasing the parameter $k$ in `prodK` or `spProdK` would compute the residuals
302 with sufficient accuracy. However, we refrained from doing this because we rarely
303 encountered entries with not maximally accurate inclusion.

304 **3. Inertia of a $2 \times 2$ Hermitian matrix.** For a decomposition $A = LDL^T$ of
305 real $A$ we need the inertia of the block diagonal matrix $D$. Thus we need the inertia
306 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
307 $\lambda_1 + \lambda_2 = \text{trace}(M) = a + c$ and $\lambda_1\lambda_2 = \det(M) = ac - b^2$. The following is true for
308 singular $M$, however, if successful then nonsingularity of $D$ will be proved a posteriori
309 by our verification algorithm.

310 If $\det(M) < 0$, then the inertia, the number of negative, zero and positive eigen-
311 values, is $\iota(M) = (1, 0, 1)$. If $\det(M) > 0$, then $\iota(M) = (0, 0, 2)$ if $\text{trace}(M) > 0$ and
312 $\iota(M) = (2, 0, 0)$ otherwise.

313 We suppose a floating-point computation in some nearest rounding barring over-
314 and underflow. A nearest rounding is defined by a rounding function $\text{fl} : \mathbb{R} \to \mathbb{F}$. For
315 $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$ that means that the floating-point result $\text{fl}(a \circ b)$ satisfies

$$|\text{fl}(a \circ b) - a \circ b| = \min\{|f - a \circ b| : f \in \mathbb{F}\}\ . \quad (316)$$

317 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

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

Then

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

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 $M := [\mathtt{a\ b; b\ c}]$. 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 $\lambda_2$. 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]`.

362 In order to distinguish the factors, we denote $A_s$ in Lines 6 and 14 by $\mathtt{As}_-$ and
363 $\mathtt{As}_+$, respectively. The matrix $\mathtt{As}_-$ in Line 6 is computed in rounding downwards
364 and therefore a lower bound on $A - sI$, i.e., $\mathtt{As}_- = A - sI - \Delta_-$ for a diagonal and
365 nonnegative matrix $\Delta_-$, and similarly for $\mathtt{As}_+$.
366 Suppose matrices $P_-, Q_-, P_+, Q_+$ are given such that

367 $$\|\mathtt{As}_- - P_- Q_- P_-^T\|_2 \leqslant \alpha_- \qquad \text{and} \qquad \|\mathtt{As}_+ - P_+ Q_+ P_+^T\|_2 \leqslant \alpha_+. \tag{4.1}$$

368 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
369 the index of the smallest positive eigenvalue of $Q_-$. Then (1.5) implies

370 $$\lambda_k(A) = \lambda_k(A - sI) + s \geqslant \lambda_k(\mathtt{As}_-) + s \geqslant \lambda_k(P_- Q_- P_-^T) + s - \alpha_- > s - \alpha_- \ .$$

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

373 $$\lambda_{\ell+1}(A) = \lambda_{\ell+1}(A + sI) - s \leqslant \lambda_{\ell+1}(\mathtt{As}_+) - s \leqslant \lambda_{\ell+1}(P_+ Q_+ P_+^T) - s + \alpha_+ \leqslant -s + \alpha_+ \ .$$

374 The smallest singular value of $A$ is equal to the smallest absolute value of an eigenvalue
375 $\lambda_\nu(A)$. If the inertia of $Q_-$ and $Q_+$ coincide, then $k = \ell$ and the ordering of the $\lambda_\nu(A)$
376 implies

377 $$\sigma_{\min}(A) = \min\left(-\lambda_{k+1}(A), \lambda_k(A)\right) \geqslant s - \max(\alpha_-, \alpha_+) \ . \tag{4.2}$$

378 Now in Step 7–8 an approximate decomposition $\mathtt{As}_- \approx L_1 S L_1^T$ is computed. Note that
379 the computation of $L_2 = S L_1^T$ does not cause rounding errors because $S$ is a signature
380 matrix, i.e., diagonal with entries $\pm 1$ on the diagonal. Hence $L_1 L_2 = L_1 S L_1^T$. Then

```
1   function [x, δ] = 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, δ] = verifySparseGen0(A,b), return
5       Compute s̃(A, L, D) by (1.12) and set s := 0.9s̃, Φ = 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 ≈ D̂_s S D̂_s^T according to (1.10)
8       Compute L_1 ≈ L D_s and L_2 = S L_1^T
9       Use (1.11) to compute α_− with ‖A_s − L_1 L_2‖_2 ⩽ α_−
10      If α_− ⩾ s, improve α_− by (2.5)
11      If α_− < s, ν_− = sum(S) > 0, goto Step 13
12      Compute α_− with ‖A_s − L_s D_s L_s^T‖_2 ⩽ α_− as in Table 3, ν_− = π(D_s)
13      If α_− ⩾ 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 ≈ D̂_s S D̂_s^T according to (1.10)
16      Compute L_1 ≈ L D_s and L_2 = S L_1^T
17      Use (1.11) to compute α_+ with ‖A_s − L_1 L_2‖_2 ⩽ α_+
18      If α_+ ⩾ s, improve α_+ by (2.5)
19      If α_+ < s, ν_+ = sum(S) > 0, goto Step 21
20      Compute α_+ with ‖A_s − L_s D_s L_s^T‖_2 ⩽ α_+ as in Table 3, ν_+ = π(D_s)
21      Set α = max(α_−, α_+), if α < s, go to Step 23
22      If Φ, Φ = false, s = s/5, goto Step 6, else ν_− = 0
23      If ν_− ≠ ν_+, verification failed, [x, δ] = verifySparseGen0(A,b), return
24      [x, δ] = ErrorBound(B, [0; b], s − α, "solve") using LDL^T for solve
```

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

381 $\alpha_-$ is computed and possibly improved in Step 10 such that $\|\texttt{As}_- - L_1SL_1^T\| \leqslant \alpha_-$.
382 If $\alpha_- < s$ in Step 11, we set $P_- := L_1$ and $Q_- := S$. Then the number $k$ of positive
383 eigenvalues of $Q_-$ is equal to $\nu_-$ and $\lambda_k(A) > s - \alpha_-$. If $\alpha_- \geqslant s$ in Step 11, we set
384 $P_- := L_s$ and $Q_- := D_s$ and compute the upper bound $\alpha_1$ for $\|A_s - L_sD_sL_s^T\|_2$ using
385 Algorithm `residualBoundLDLT` in Table 3. The number $k$ of positive eigenvalues of
386 $Q_-$ is equal to $\nu_-$ which is computed by $\pi(D)$ based on Algorithm `NumPosEV` in Table
387 5. Hence $\lambda_k(A) > s - \alpha_-$ as well.

388 If $\alpha_- \geqslant s$, the verification is not yet successful for the choice of $s$. In that case we
389 go to Step 22 to try once more with decreased $s$.

390 The computations in Lines 14 – 20 are similar to those in Lines 6 – 12 replacing
391 the subindex "-" by "+". It follows that the number $\ell$ of positive eigenvalues of $Q_+$
392 is equal to $\nu_+$ and that $\lambda_{\ell+1}(A) \leqslant -s + \alpha_+$. If $\alpha := \max(\alpha_-, \alpha_+) < s$ in Step 21 and
393 $\nu_- = \nu_+$ in Step 23, then $k = \ell$ and (4.2) implies $\sigma_{\min}(A) \geqslant s - \alpha > 0$.

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

395 large. In that case we reduce $s$ and try the verification from Lines 6 – 21 again. If
396 still $\alpha \geqslant s$ or $\nu_- \neq \nu_+$ in Step 23, then verification failed and we turn to subalgorithm
397 "verifySparseGen0".
398 If the verification was successful, the positive lower bound $s - \alpha$ on $\sigma_{\min}(A)$
399 verifies that the matrix $A$ is nonsingular, and entrywise bounds for the solution of
400 the linear system are computed by Algorithm `ErrorBound` in Table 1 of Part I of this
401 note. To compute almost always maximally accurate inclusions we may use Algorithm
402 `ErrorBound3` as in Table 4.
403 The difference to Algorithm "verifySparseSym" in Part I of this note is as follows.
404 Here the input matrix is shifted by $s$ to the left and right. If the inertia of the corre-
405 sponding $LDL^T$-decompositions are the same, then $s-\alpha$ is a lower bound for $\sigma_{\min}(A)$
406 subject to the maximum $\alpha$ of the residual bounds. The drawback is some additional
407 fill-in of the factors $L$ of the shifted matrices. As a consequence "verifySparseSym0"
408 is slower but seems a little more stable.
409 In "verifySparseSym" in Part I of this note we decompose $A \approx L_1 L_2$ and estimate
410 the smallest singular value of $L_1$ using a Cholesky decomposition of $L_1 L_1^T$ subject to
411 a norm bound of the residual $A - L_1 L_2$. That turns out to be faster, but in rare cases
412 it is less stable. See the numerical results in Section 9.

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

415 (5.1)
$$B := \begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix}$$

416 the singular values of which are ± the eigenvalues of $A$. This matrix is used in [46] as
417 well.
418 As in the symmetric case we explore on Theorem 1.1 published in [38, Theorem
419 1.1]. The original method relied on approximate $LDL^T$-decompositions of $A \pm sI$ for
420 a shift $s$ being an anticipated lower bound of $\sigma_{\min}(A)$. In contrast to the symmetric
421 case, one shift suffices for the augmented matrix $B$ because the inertia of $B$ is known
422 beforehand. That is at least true for nonsingular matrix $A$. We do not assume
423 nonsingularity of $A$ beforehand but prove it a posteriori so that all deductions are
424 true.
425 Rather than $LDL^T$ as in [38, Theorem 1.1] we use, as in the symmetric case, a
426 decomposition $L_1 L_2$ of $B - sI$ as presented in Part I of this note, were $L_2 = SL_1^T$ for
427 a signature matrix $S$. That implies the same advantages as in the symmetric case.
428 In contrast to [38, 40, 46] we proceed for general matrices as follows. After equili-
429 brating the original matrix $A$ we compute an $LDL^T$-decomposition of the augmented
430 matrix $B$ by (1.8). As has been observed in Part I in some cases the computed $D$
431 is singular, even for moderately conditioned input matrix. That should not happen,
432 and we cure it as in (1.7).
433 Based on the factors $L, D$ we compute in Step 7 an anticipated lower bound $s$ for
434 the smallest singular value of $B$ which is equal to that of $A$. Although $B$ has double
435 the size of $A$, the iteration (1.12) to compute $s$ as a lower bound of $\sigma_{\min}(B)$ rather
436 than of $\sigma_{\min}(A)$ is more stable due to the symmetry of $B$.
437 A splitting (1.10) of $D$ is computed in Step 9, and in Step 10 the factors $L_1, L_2$
438 such that $L_1 L_2 \approx A$. The factor $L_2$ is $L_1$ multiplied by some signature matrix. That
439 computation is error-free, so that as in subalgorithm "verifySparseSym" in Part I of
440 this note the factors $L_1, L_2$ have identical sets of singular values.

```
1  function [x, δ] = 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 s̃(B,L,D) ≲ σ_min(B) by (1.12) and set s := 0.9s̃, Φ = true
8      Rounding downwards, B_s := B − sI and compute L_s D_s L_s^T(B_s) by (1.8)
9      Compute approximate splitting D_s ≈ D̂_s S D̂_s^T according to (1.10)
10     Compute L_1 ≈ 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 ⩽ α
12     If α < s,  improve α by (2.5)
13     If α < s, ν = sum(Ds) > 0, else improve α by (2.6), ν = π(D_s)
14     If  α < s, go to Step 16
15     If Φ, Φ = false, s = s/5, goto Step 8, else ν = 0
16     If ν ≠ n, verification failed, return
17     [x, δ] = ErrorBound(B, [0; b], s − α, "solve") using LDL^T for solve
```

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

441 The remaining of the subalgorithm `VerifySparseGen0` is identical to subalgo-
442 rithm `VerifySparseGen` in Table 5 of Part I of this note. Hence, if successful, $s - \alpha$
443 is a lower bound for $\sigma_{\min}(B) = \sigma_{\min}(A)$.
444 Error bounds for the solution of the original linear system $Ax = b$ use that

445 (5.2)
$$\begin{pmatrix} 0 & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix}$$

446 implies $x = A^{-1}b$ and we proceed as in Part I of this note.
447 As for "verifySparseSym0" the difference is that "verifySparseGen0" shifts the
448 augmented matrix and computes a lower bound for $\sigma_{\min}(B)$ using Sylvester's law
449 of inertia. In contrast, "verifySparseGen" relies on the factorization $L_1L_2$ of the
450 original augmented matrix $B$ without shift and computes a lower bound for $\sigma_{\min}(B)$
451 based on a Cholesky factorization of $L_1L_1^T$. In rare cases that does not allow a
452 verification where "verifySparseGen0" does. In general, however, "verifySparseGen0"
453 seems slower because the decomposition of the shifted causes additional fill-in, see the
454 computational results in Section 9.

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

459 For $A \in \mathbb{C}^{m\times n}$ with $m > n$ and $b \in \mathbb{C}^m$ define (cf. [11, Chapter 20])[3]

460 (6.1)
$$\begin{pmatrix} 0 & A^H \\ A & -I_m \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \quad \Rightarrow \quad A^H y = 0 \text{ and } Ax - y = b \ ,$$

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

465 The system matrix in (6.1) is symmetric indefinite, so our subalgorithms "verifyS-
466 parseSym" and "verifySparseSym0" are applicable. In [42] we published algorithms to
467 compute verified error bounds for least squares problems and underdetermined linear
468 systems with full matrix. In that paper we used

469
$$\begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} .$$

470 Although the system matrix is not Hermitian, we showed numerical evidence in [42]
471 that the computed inclusions are sometimes more accurate than using (6.1). However,
472 for our present approach we have to stick to the Hermitian input matrix.

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

475 (6.2)
$$\begin{pmatrix} -I_n & A^H \\ A & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} \quad \Rightarrow \quad Ax = b \text{ and } A^H y = x \ ,$$

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

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

483 In subalgorithms "verifySparseSym" or "verifySparseSym0" the symmetric equi-
484 libration (1.7) (which is (3.3) in Part I of this note) is applied, i.e., two steps of the
485 Sinkhorn-Knopp algorithm. That means the rows of $A^T$ and rows of $A$ are equili-
486 brated from the left, and similarly from the right. Thus, although $B$ is symmetric,
487 the matrix $A$ is equilibrated independently from the left and right. That produces
488 stable results.

489 When computing error bounds for the square linear systems (6.1) or (6.2) by
490 our algorithms, the nonsingularity of the augmented matrix is verified. In turn, that
491 implies that $A$ has full rank and our conclusions are valid.

492 There are other possibilities to define the solution of an underdetermined linear
493 system. For example, Matlab computes a solution of $Ax = b$ with at most $m$ nonzero
494 entries. This can be done as follows. First, an $LU$-decomposition of $A^H$ is computed
495 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 $p_1, \dots, p_m$ 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 $\xi_1, \dots, \xi_n \in x \underline{\cup} \tilde{x}$, the convex union of $x$ and $\tilde{x}$, with

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

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 $\xi_i$ cannot, in general, be replaced by a single $\xi$, 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 $\mathtt{J} \in \mathbb{F}^{n \times n}$ with $\mathtt{J} \approx J_f(x)$. More important, let $\mathtt{X} \in \mathbb{IF}^n$ be an interval vector with $\mathtt{X} \subseteq \mathbf{D}$. Then the call

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

computes `Y` such that $\mathtt{Y.x} \in \mathbb{IF}^n$ is an interval vector with $\{f(x) : x \in \mathbf{X}\} \subseteq \mathtt{Y.x}$, and `Y.dx` is an interval matrix $\mathtt{Y.dx} \in \mathbb{IF}^{n \times n}$ with $\{\nabla f_k(\xi) : \xi \in \mathtt{X}\} \subseteq \mathtt{Y}_k$ for all $k \in \{1, \dots, n\}$. 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

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

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

*This manuscript is for review purposes only.*

532 THEOREM 7.1. *Let continuously differentiable* $f : \mathbf{D} \to \mathbb{R}^n$ *with* $\mathbf{D} \in \mathbb{IR}^n$ *and*
533 $\mathtt{x}, \mathtt{xs} \in \mathbf{D} \cap \mathbb{F}^n$ *be given. Define* `Y = f(gradientinit(hull(x,xs)))`*. Then*

534 (7.5)
$$f(x) \in f(\tilde{x}) + \mathtt{Y.dx}\big(x - \tilde{x}\big) \; .$$

535 Using this we can formulate [41, Theorem 13.3] the following theorem to compute
536 error bounds for a solution of a system of nonlinear equations $f : \mathbb{R}^n \to \mathbb{R}^n$ based on
537 some approximate solution $\tilde{x} \in \mathbb{R}^n$.

538 THEOREM 7.2. *Let continuously differentiable* $f : D \to \mathbb{R}^n$ *and* $\tilde{x} \in \mathbb{R}^n$, $\mathbf{X} \in \mathbb{IR}^n$,
539 $R \in \mathbb{R}^{n\times n}$ *with* $0 \in \mathbf{X}$ *and* $\tilde{x} + \mathbf{X} \subseteq D$ *be given. Suppose*

540 (7.6)
$$S(\mathbf{X}, \tilde{x}) := -Rf(\tilde{x}) + \big\{I - RJ_f(\tilde{x} + \mathbf{X})\big\}\mathbf{X} \;\subseteq\; int(\mathbf{X})$$

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

543 The bound $\tilde{x} + S(\mathbf{X}, \tilde{x})$ is computable and is mathematically rigorous including the
544 proof of uniqueness of the root $\widehat{x}$ of $f$ in $\tilde{x} + S(\mathbf{X}, \tilde{x})$.
545 A practical application as implemented in Algorithm `verifynlss` in INTLAB
546 uses an approximate inverse $R$ of $J_f(\tilde{x})$ which is, in general, a full matrix. Therefore,
547 an inclusion based on Theorem 7.2 is hardly applicable to large systems of nonlinear
548 equations even if the Jacobian is sparse.
549 In practice, however, often individual variables $x_k$ have few dependencies on other
550 variables. As a consequence, the Jacobian becomes sparse, often a banded matrix.
551 Next we show how the assumption (7.6) of Theorem 7.2 can be verified by solving a
552 linear system with point matrix and interval right hand side. Then our methods for
553 the solution of sparse linear systems are applicable.
554 We follow [41, Section 13, page 87] and compute an inclusion $\mathbf{J} \in \mathbb{IF}^{n\times n}$ of $J_f(\tilde{x}{+}\mathbf{X})$
555 as in (7.2). Hence (7.4) implies that for all $\xi \in \tilde{x} + \mathbf{X}$ and for all $k \in \{1, \dots, n\}$ the
556 gradient $\nabla f_k(\xi)$ is included in the $k$-th row of $\mathbf{J}$, and Theorem 7.1 is applicable.
557 Denote $\check{C} = \mathrm{mid}(\mathbf{J})$ and $\Delta := \mathrm{rad}(\mathbf{J})$. Assume that $\check{C}$ is nonsingular and suppose

558 (7.7)
$$\{y \;:\; \check{C}y = -f(\tilde{x}) - \varrho x,\; -\Delta \leqslant \varrho \leqslant \Delta,\; x \in \mathbf{X}\} \subseteq \mathbf{Y} \; .$$

559 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

560
$$-\check{C}^{-1}f(\tilde{x}) + \big\{I - \check{C}^{-1}[\check{C} + \varrho]\big\}x = \check{C}^{-1}\big( - f(\tilde{x}) - \varrho x\big)$$

561 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)
562 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
563 transforms the problem of computing verified bounds for the solution of a nonlinear
564 system to the solution of a linear system with interval right hand side. Note that
565 (7.6) proves the nonsingularity of $\check{C}$ as well.
566 Now $\mathbf{X}$ is an anticipated inclusion of the difference of the true solution $\widehat{x}$ of
567 the nonlinear system $f(x) = 0$ to the approximate solution $\tilde{x}$. And if successful, i.e.
568 $\mathbf{Y} \subset \mathrm{int}(\mathbf{X})$, then $\widehat{x}{-}\tilde{x} \in \mathbf{Y}$. If $\tilde{x}$ is a good approximation, then $\mathbf{X}$ is small in magnitude
569 and essentially symmetric to the origin. As a consequence we further simplify (7.7) by
570 using the magnitudes[4] $\overline{X}$ and $\overline{Y}$ of $\mathbf{X}$ and $\mathbf{Y}$, and set $\mathbf{X} := [-\overline{X}, \overline{X}]$ and $\mathbf{Y} := [-\overline{Y}, \overline{Y}]$.
571 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.

572 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
573 interested in computing an inclusion of the "outer inclusion set", see (5.1) in Part I
574 of this note:

575 (7.8) $$\Sigma(A,\mathbf{b}) := \{x \in \mathbb{R}^n : \exists b \in \mathbf{b} \text{ with } Ax = b\} \;.$$

576 To that end we use Algorithm "verifySparselss" as in Table 6 in Part I of this note
577 with small modifications. First, we remove the check for least squares and under-
578 determined problems. Furthermore, the only modification is replacing the calls of
579 "ErrorBound" in last line in subalgorithms "verifySparseSPD", "verifySparseSym"
580 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 = (abs(rho) + err + r)/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.*

581 The input parameter $s$ is a lower bound on $\sigma_{\min}(A)$ and `@solve` is some routine
582 delivering an approximate solution of a linear system. As in the original algorithm
583 "ErrorBound" `@solve` is based on the already computed decomposition in each of the
584 subalgorithms.
585 The proof of correctness of Algorithm "ErrorBoundI" is as follows. Let $\mathbf{b} \in \mathbb{IF}^n$
586 be an interval vector. Then $\mu, r \in \mathbb{F}^n$ in Line 2 are computed such that $\mu - r \leqslant b \leqslant \mu + r$
587 for all $b \in \mathbf{b}$. In Line 3 an approximate solution $\tilde{x}$ of the midpoint equation $Ax = \mu$
588 is computed and is improved in Line 4 by one residual iteration. According to [45]
589 that implies backward stability of $\tilde{x}$. Line 5 computes an inclusion `rho` ± `err` of the
590 residual $|A\tilde{x} - \mu|$, such that in particular $|A\tilde{x} - \mu| \leqslant |\texttt{rho}| + \texttt{err}$. Now `delta` in Line 7 is
591 computed in rounding upwards, and with the lower bound $s$ on $\sigma_{\min}(A)$ it follows

592 (7.9) $$\begin{aligned} |A^{-1}b - \tilde{x}| &\leqslant |A^{-1}|\,|b - A\tilde{x}| \\ &\leqslant |A^{-1}|\,(|\mu - A\tilde{x}| + r) \\ &\leqslant \|A^{-1}\|_\infty\,(|\mu - A\tilde{x}| + r) \\ &\leqslant \|A^{-1}\|_2\,(|\texttt{rho}| + \texttt{err} + r) \\ &\leqslant \delta \end{aligned}$$

593 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
594 consider

595 (7.10)
```
y = -f(intval(xs));
setround(1)
b = midrad( y.mid , y.rad + J.rad * mag(X) );
```

596 The first line computes an inclusion `y` $\in \mathbb{IF}^n$ of $-f(\tilde{x})$ with $-f(\tilde{x}) \in$ `y.mid±y.rad`. The
597 second statement switches the rounding to upwards, and finally $\mathbf{b}$ is an inclusion of

```
1   function [X,kxs,kY] = verifySparseNlss(f,xs)
2     setround(0)
3     n = size(xs,1); phi = 1e-14*sqrt(n);
4     dxs = abs(xs); kxs = 0;
5     while ( kxs < 15 )          % at most 10 Newton iterations
6       kxs = kxs + 1; xsold = xs;
7       y = f(gradientinit(xs));   % function value and gradient
8       xs = xs - y.dx\y.x;        % approximate Newton iteration
9       d = abs(xs-xsold);
10      if all(d<.5*abs(xs)) && ( norm(d,inf)<=phi*norm(xs,inf) )
11        break
12      end
13    end
14    ys = -f(intval(xs));          % inclusion of f(xs)
15    Y = mag(ys);                 % magnitude of ys
16    kY = 0; setround(1)
17    while ( kY < 10 )
18      kY = kY + 1;
19      X = 1.01*Y + realmin;      % epsilon-inflation
20      JJ = f(gradientinit(midrad(xs,X)));
21      J = JJ.dx;                 % inclusion of Jacobian
22      b = midrad( ys.mid , ys.rad + J.rad*X );
23      [Ys,delta] = verifySparse(J.mid,b);
24      Y = abs(Ys) + delta;       % r.h.s. of (7.7)
25      if all( Y < X )
26        X = midrad(xs,Y);        % inclusion successful
27      return
28    end
29    X = intval(NaN(size(xs)));   % inclusion failed
30  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.*

598 `y.mid`$\pm\varrho$ for all $|\varrho| \leqslant$ `y.rad + J.rad*mag(X)`. Thus $-f(\tilde{x}) - \varrho x \in \mathbf{b}$ for all $x \in \mathbf{X}$ and
599 $|\varrho| \leqslant \Delta$. It follows that an inclusion $\mathbf{Y}$ of the linear system with matrix $\check{C}$ and right
600 hand side $\mathbf{b}$ satisfies (7.7). As a consequence, $\mathbf{Y} \subseteq \mathrm{int}(\mathbf{X})$ implies $\widehat{x} \in \tilde{x} + \mathbf{Y}$.
601 The algorithm to solve a system of nonlinear equations works as follows. First
602 we apply some Newton iterations to produce a good approximation $\tilde{x}$ of $f(x) = 0$.
603 Then $f(\tilde{x})$ should be small and the magnitude of $\mathbf{b}$ is dominated by the radius $\Delta$ of
604 the inclusion of $J_f(\tilde{x} + \mathbf{X})$. The residual of the linear system cannot become smaller
605 than the magnitude of $\mathbf{b}$, which in turn increases with the sensitivity of the problem.
606 Therefore, there is no need to improve an approximate solution of $\check{C}y = \mathbf{b}$ by a residual
607 iteration and we may apply algorithms "verifySparselss" or "verifySparselss0" with
608 using Algorithm "ErrorBoundI" as in Table 8 rather than "ErrorBound".
609 Executable Matlab/INTLAB code of Algorithm `verifySparseNlss` to compute
610 rigorous error bounds for the solution of a nonlinear system $f(x) = 0$ based on an
611 approximate solution $\tilde{x}$ is given in Table 9. The rationale is as follows. In Line 2 the

612 rounding is set to nearest, and in Lines 5 – 13 some Newton iterations are applied to
613 improve the approximation $\tilde{x}$. The statement `y = f(gradientinit(xs))` in Line 7
614 computes `y` such that `y.x` $\approx f(\tilde{x})$ and `y.dx` is an approximation of the Jacobi matrix
615 of $f$ at $\tilde{x}$ using the gradient toolbox, which in turn is based on forward automatic
616 differentiation [4, 10] and implemented in INTLAB [39]. Therefore Line 8 is one
617 (approximate) Newton step.

618 The quantity `ys` in Line 14 is an inclusion of $-f(\tilde{x})$ and `Y` its magnitude. Lines
619 17 – 28 are an interval iteration adapted to the description in [41]. Recall that `Y` is
620 a positive real vector, and the anticipated inclusion of the error with respect to $\tilde{x}$
621 is the interval vector $[-Y, +Y]$. Line 19 is one step of the so-called epsilon inflation
622 introduced in [36]. The target is $Y < X$, or equivalently $[-Y, +Y] \subseteq \mathrm{int}[-X, +X]$. The
623 inclusion may fail if $[-X, +X]$ is too narrow, so $[-X, +X]$ is intentionally widened.
624 The success of the epsilon-inflation can be analyzed theoretically, see [41]. On the
625 other hand $[-X, +X]$ should not be too wide because that widens the Jacobian and
626 may prevent $Y < X$.

627 The purpose of the epsilon-inflation is to identify a good candidate for inclusion.
628 The right hand side $\mathbf{b}$ should be a narrow interval around $f(\tilde{x})$. More precisely,
629 according to (7.9) around $-f(\tilde{x})$, but that doesn't matter because our inclusion is
630 symmetric to the origin. Therefore, basically $\pm 1.01|f(\tilde{x})|$ is our first choice. We need
631 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
632 $f(z) \in JJ.x$ and the Jacobian of $f$ at $z$ is in $JJ.dx$ for all $z \in \mathbf{Z}$. Hence `J` in Line
633 21 is what we need. The next Line 22 computes $\mathbf{b}$ as in (7.10), and the next line an
634 inclusion $Ys \pm \delta$ of the linear system with matrix $\check{C} = \mathrm{mid}(\mathbf{J})$ and right hand side $\mathbf{b}$.
635 The magnitude of the inclusion is `Y` as in Line 24, and if $Y < X$ is true for all entries
636 then `midrad(xs,Y)` is an inclusion of the solution of the nonlinear system.

637 If $Y_k \geqslant X_k$ for some $k$, then the inclusion is tried again with $X$ replaced by a little
638 widened $Y$. In some way these are also Newton steps. In each step a new Jacobian
639 `J` at $\mathbf{Z}$ is computed, and the widened $Y$ reflects the width of the previous `J`. If not
640 successful after some 10 trials, the verification failed.

641 Unlike for linear systems we cannot expect, in general, maximally accurate inclu-
642 sions because the lack of an accurate residual iteration and, more important, because
643 of nonlinearities of $f$ widening the Jacobi matrix. Nevertheless the method works well
644 in a number of examples, see the test results in Section 9.

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

650 To distinguish our algorithms, we use `verifySparselss` for our algorithm pre-
651 sented in Part I (also called "new" in there) and use `verifySparselss0` for the algo-
652 rithm presented in this Part II (henceforth called "new0"). The latter is identical to
653 the former except replacing subalgorithms "verifySparseSym" and "verifySparseGen"
654 by "verifySparseSym0" and "verifySparseGen0", respectively. Executable code of Al-
655 gorithm `verifySparselss0` including least squares problems and underdetermined
656 linear systems is presented in Table 10.

657 The algorithm first checks for the type of problem, namely $m > n$ for a least
658 squares problem and $m < n$ for an underdetermined system of equations. In either

[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.*

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 added[6] to both algorithms an extra input parameter `acc`. If *true*, then inclusions with improved accuracy as described in Section 2 are computed by storing an approximation by an unevaluated sum of three instead of two parts. In that case we use Algorithm `ErrorBound3` as in Table 4.

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 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.

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

[6] Since it is clear how to do that and in order to keep the codes simple, that is not shown in Table 6 for subalgorithm "verifySparseSym0" and Table 7 for subalgorithm 'verifySparseGen0".

*This manuscript is for review purposes only.*

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 |

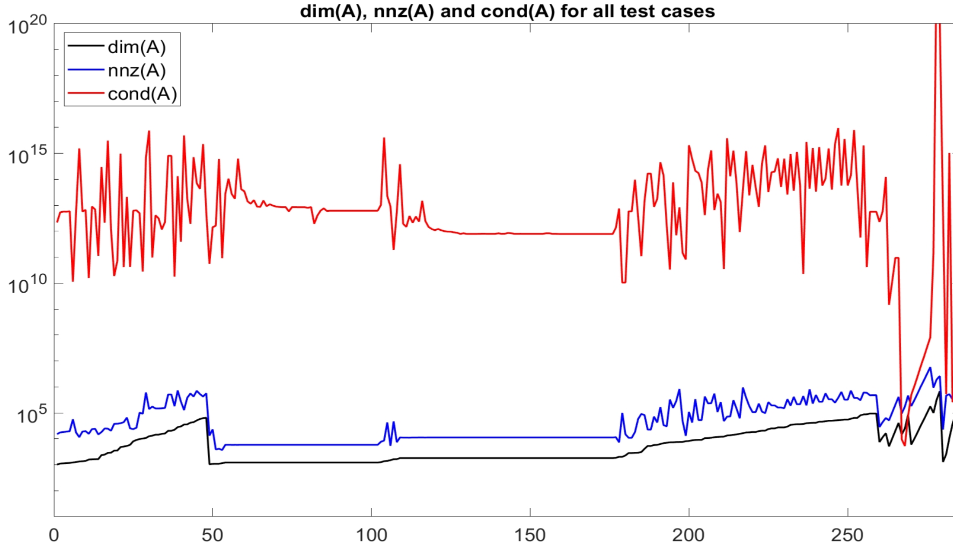


FIG. 2. *Dimension, number of nonzero elements and condition number of all test matrices.*

705 less than 1 means that "new0" is faster than "new". That is rarely the case. In the
706 median over all examples Algorithm `verifySparselss` from Part I of this note is
707 faster than `verifySparselss0` by a factor 1.23, at most by a factor 5.3. Conversely,
708 "new0" is faster than "new" by at most a factor 2.8.

709 In some way Algorithm `verifySparselss0` is simpler than `verifySparselss`, so
710 we may ask why it is slower. Both algorithm start with computing some factor $L_1$,
711 both for symmetric as for general matrices. However, "new" computes for symmetric
712 input matrix $A$ a factor of $A$, but "new0" of $A$ shifted by $s$. Similarly, "new" computes
713 a factor of the augmented matrix $B$, but "new0" of $B$ shifted by $s$ for general input
714 matrix $A$. That causes a significant fill-in for method "new0". In Figure 4 we display
715 the ratio of the number of nonzero entries of the factor $L_1$ in `verifySparselss0`
716 divided by that of `verifySparselss`. Hence a value greater than 1 means that "new0"
717 has more fill-in than "new".

718 The median ratio of fill-in over all examples is 2.4, and maximally the factor
719 $L_1$ by "new0" has 8.7 times more elements than that of $L_1$ by "new". That is true
720 although we reduced the number of elements as explained in (3.5)ff in Part I of this
721 note by setting entries in $L$ smaller than $10^{-30}$ in magnitude to zero in case the first

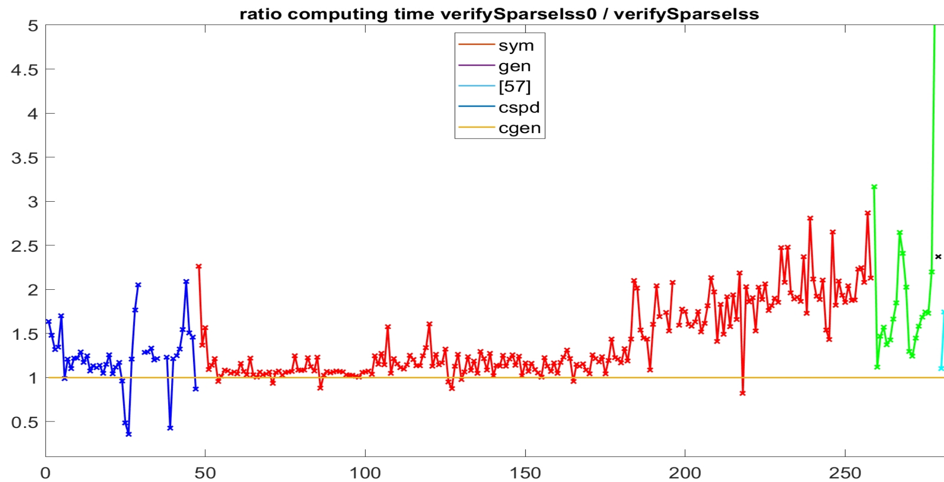


FIG. 3. *Ratios of computing times* $t_{\texttt{verifySparselss0}}/t_{\texttt{verifySparselss}}$.

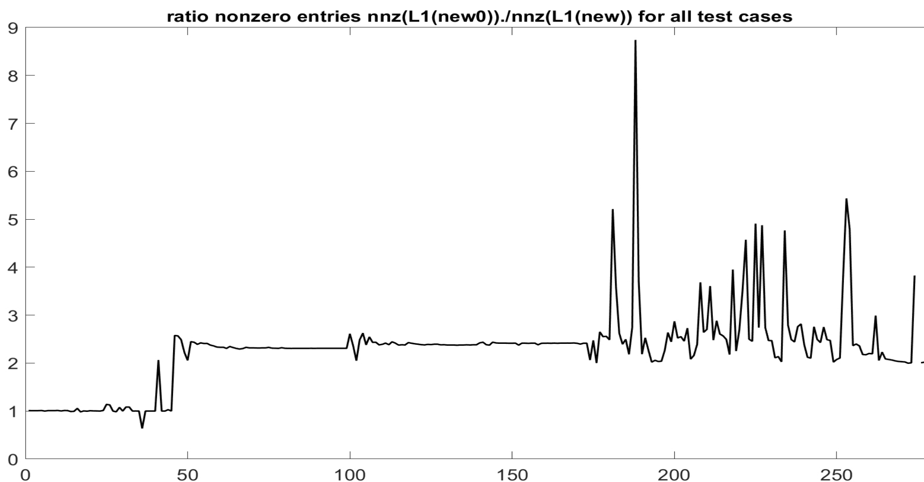


FIG. 4. *Ratio of number of nonzero entries of* $L_1$ *in "new0" divided by that of "new".*

722 $LDL^T$-decomposition failed due to singular $D$.
723 Next we show in Figure 5 a rough image of the median relative error of the
724 Algorithms `verifySparselss` and `verifySparselss0`. As can be seen in both cases
725 usually almost maximally accurate approximations are computed. In the median the
726 relative error of all entries of the inclusion computed by Algorithms `verifySparselss`
727 and `verifySparselss0` is $3.6 \cdot 10^{-17}$, the maximum relative error is around $10^{-4}$.
728 As discussed at the end of Section 2 we may reduce the maximal relative error of
729 the inclusion further by an improved residual iteration, storing an approximate solu-
730 tion by an unevaluated sum of three instead of two parts. In the practical implemen-
731 tation we added an extra parameter `acc` and eventually use Algorithm `ErrorBound3`

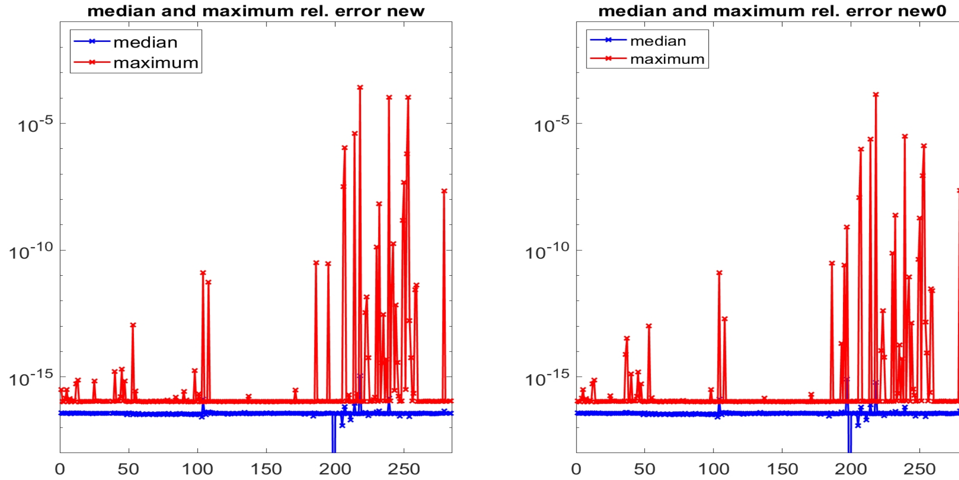


FIG. 5. *Median of relative errors of* `verifySparselss` *and* `verifySparselss0`.

732 as in Table 4. If the maximum relative error of the inclusion is beyond some thresh-
733 old, then we switch from two to three parts for the approximate solution. We used
734 the threshold $10^{-15}$ for the maximal error of all entries of the inclusion. When using
735 `acc`= *true* the relative errors are as shown in Figure 6. Now for most examples the
736 inclusions for all entries of the solution are maximally accurate.

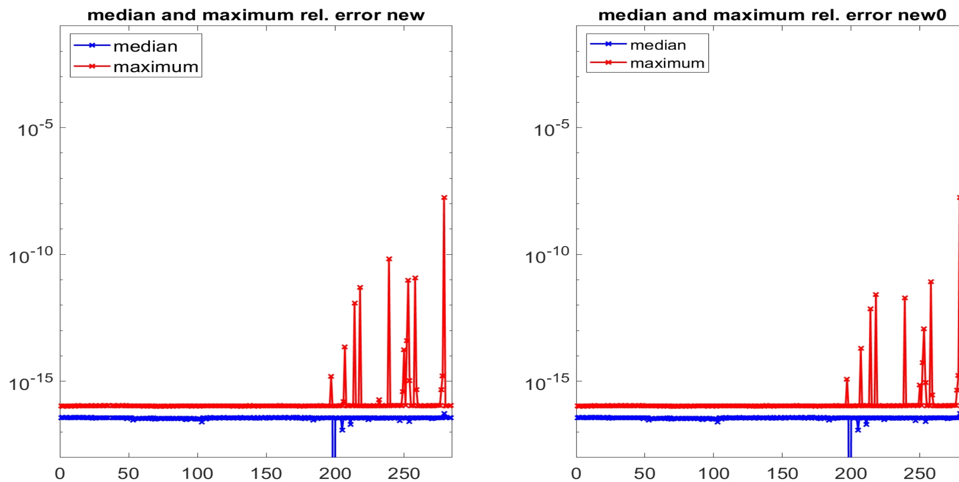


FIG. 6. *Median of relative errors with option* `acc` $=$ *true.*

737 The additional computing time is marginal for a vector right hand side because
738 only some extra $\mathcal{O}(n^2)$ operations are necessary. For multiple right hand sides that
739 changes. As a result, for almost all test cases maximally accurate inclusions for all
740 entries are computed.

741 We discuss some details of our Algorithm `verifySparselss0` on the several im-

provement 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 $\alpha$ 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 $\alpha$ 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 at the `url` in (9.2). The curios ratio 1.43 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,

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 1.43

791 "new" succeeds in 98 cases to compute bounds with at least 11 coinciding figures
792 in each entry, "new0" succeeds in all cases. For randomly generated examples there
793 is not much difference in the accuracy of the bounds, but "new" is mostly more
794 than twice as fast as "new0". In Figure 7 we show the ratio of computing times
795 of Algorithm `verifySparselss0` divided by that of Algorithm `verifySparselss`.
Algorithm "new" from Part I of this note is always faster than "new0". As explained

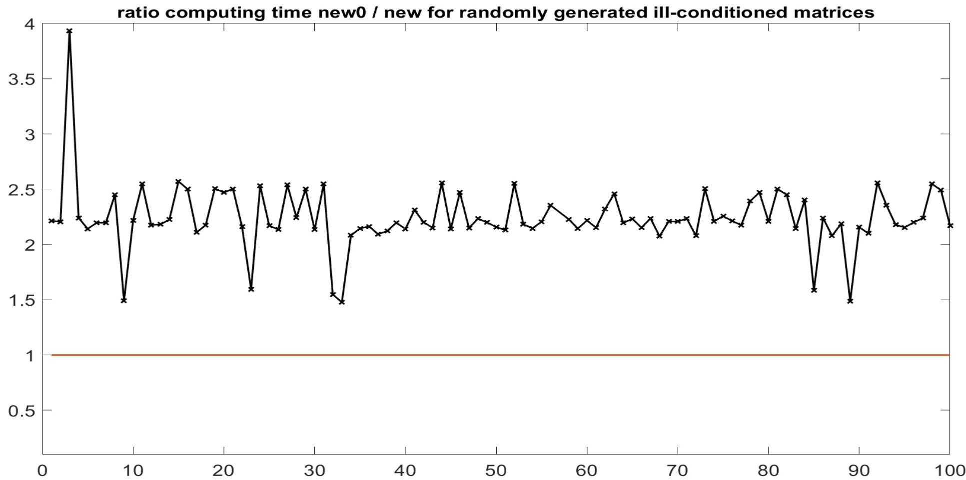


FIG. 7. *Ratios of computing times* $t_{\texttt{verifySparselss0}}/t_{\texttt{verifySparselss}}$.

796
797 before that is related to the number of nonzero elements of the matrices $L_1$.
798 We tested Algorithm `verifySparselss0` for complex data as well. Some data is
799 shown in the `url` in (9.2). As there were no surprises we refrain, as in Part I of this
800 note, from extending our already shown computational data.
801 Next we show computational results for rectangular input matrix. As has been
802 mentioned, Matlab chooses to minimize the number of nonzero elements of the solution
803 rather than computing $A^+b$. Therefore we show only data for least squares problems.
804 Since the matrix in (6.2) is a permutation of that in (6.1) this is gives information of
805 the underdetermined cases as well. If a test matrix $A$ in [5] has more columns than
806 rows we use $A^H$.
807 We use all matrices from the Suite Sparse Matrix Collection [5] with dimensions

808 $$(9.3) \qquad 10^3 \leqslant m,n \leqslant 10^5 \quad \text{and} \quad 10^7 \leqslant cnd \leqslant 10^{16} \quad \text{and} \quad \texttt{nnz(A)} \leqslant 10^6 \ .$$

809 The condition number of a rectangular matrix with respect to a least squares problem

*This manuscript is for review purposes only.*

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 | $t_{\mathtt{new}}$ | $\varrho$ | median | max | median | max |
| 2221 | 10798 | 608540 | 7.0e14 | 13.97 | 1.87 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1247 | 12546 | 140034 | 7.6e15 | 26.47 | 2.58 | NaN | NaN | NaN | NaN |
| 1210 | 20360 | 509866 | 8.1e14 | 396.42 | 1.18 | NaN | NaN | 4.2e-17 | 1.9e-13 |
| 1451 | 20360 | 509866 | 8.1e14 | 391.55 | 1.14 | NaN | NaN | 4.0e-17 | 4.3e-15 |
| 2229 | 28216 | 730080 | 1.3e14 | 9.07 | 1.32 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 949 | 41731 | 559341 | 1.9e12 | 210.97 | 1.57 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 950 | 51035 | 707985 | 7.1e13 | 4.64 | 1.32 | 3.7e-17 | 5.8e-15 | 3.7e-17 | 4.7e-15 |
| 1225 | 64810 | 565996 | 5.3e12 | 24.58 | 1.37 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 243 | 1080 | 23094 | 1.4e12 | 0.63 | 1.44 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 1074 | 1220 | 5892 | 8.6e12 | 0.21 | 1.41 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 438 | 1633 | 46626 | 1.9e11 | 0.87 | 1.43 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 465 | 2904 | 58142 | 3.5e12 | 0.98 | 1.50 | 2.8e-17 | 1.1e-16 | 2.8e-17 | 1.1e-16 |
| 439 | 3096 | 90841 | 1.1e11 | 2.18 | 1.70 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 548 | 5850 | 42568 | 1.8e13 | 2.29 | 1.51 | 3.4e-17 | 1.1e-16 | 3.4e-17 | 1.1e-16 |
| 818 | 6316 | 167178 | 4.5e14 | 2.97 | 1.76 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 934 | 7055 | 30082 | 1.7e12 | 1.59 | 1.12 | NaN | NaN | 4.2e-17 | 5.6e-14 |
| 446 | 7320 | 324772 | 3.3e10 | 8.71 | 1.68 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 739 | 7337 | 156508 | 7.6e13 | 2.40 | 1.35 | 3.8e-17 | 2.0e-11 | 3.8e-17 | 1.8e-11 |
| 920 | 7500 | 283992 | 7.0e11 | 34.95 | 2.00 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1395 | 7548 | 834222 | 8.3e12 | 33.43 | 0.68 | NaN | NaN | 1.3e-15 | 4.6e-10 |
| 2814 | 8256 | 109368 | 2.1e15 | 11.91 | 1.69 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 448 | 9035 | 335472 | 2.1e14 | 4.87 | 1.49 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 580 | 9129 | 52883 | 1.7e14 | 4.72 | 1.65 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 581 | 9129 | 52883 | 7.5e13 | 4.60 | 1.64 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 1405 | 10605 | 424587 | 4.2e12 | 2.87 | 1.38 | 1.2e-17 | 1.1e-16 | 1.2e-17 | 1.1e-16 |
| 741 | 10672 | 232633 | 2.3e14 | 3.96 | 1.48 | 3.9e-17 | 2.2e-9 | 3.8e-17 | 8.0e-10 |
| 743 | 10964 | 233741 | 1.3e15 | 6.70 | 1.62 | 6.7e-17 | 5.9e-6 | 6.5e-17 | 5.0e-6 |
| 921 | 11532 | 551184 | 6.5e12 | 174.38 | 2.16 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 550 | 11790 | 107383 | 2.8e13 | 8.87 | 1.79 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 570 | 13694 | 72734 | 1.3e14 | 3.98 | 1.69 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 745 | 14270 | 307858 | 1.3e15 | 8.04 | 1.49 | 9.3e-17 | 2.7e-6 | 8.2e-17 | 1.6e-6 |
| 551 | 14760 | 145157 | 6.7e13 | 14.37 | 1.90 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 922 | 16428 | 948696 | 4.2e13 | 471.99 | 2.22 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 747 | 17576 | 381975 | 1.2e15 | 27.89 | 0.79 | 1.1e-15 | 4.2e-5 | 5.9e-16 | 2.2e-5 |
| 553 | 17730 | 183325 | 2.4e13 | 26.55 | 1.91 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 582 | 18289 | 106803 | 3.6e14 | 14.49 | 1.81 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 583 | 18289 | 106803 | 5.1e13 | 13.93 | 1.81 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 431 | 19716 | 227872 | 8.8e12 | 6.83 | 1.32 | 4.0e-17 | 1.5e-12 | 3.8e-17 | 4.8e-14 |
| 572 | 20614 | 111903 | 3.9e14 | 7.41 | 1.85 | 3.7e-17 | 3.1e-13 | 3.6e-17 | 9.1e-14 |
| 555 | 23670 | 259648 | 3.2e13 | 46.80 | 2.02 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 1109 | 25187 | 193276 | 1.9e14 | 5.52 | 1.48 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 1111 | 25187 | 193216 | 2.0e14 | 5.73 | 1.50 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 584 | 27449 | 160723 | 6.4e14 | 24.71 | 1.87 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 585 | 27449 | 160723 | 5.1e13 | 24.26 | 1.88 | 3.8e-17 | 5.9e-15 | 3.8e-17 | 4.3e-16 |
| 574 | 27534 | 151063 | 6.3e14 | 12.88 | 2.29 | 4.0e-17 | 2.4e-10 | 4.0e-17 | 1.4e-10 |
| 557 | 29610 | 335972 | 2.6e13 | 59.97 | 2.00 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |

*This manuscript is for review purposes only.*

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* | $t_{\mathtt{new}}$ | $\varrho$ | *median* | *max* | *median* | *max* |
| 576 | 34454 | 190224 | 9.4e14 | 17.73 | 2.26 | 4.6e-17 | 9.1e-9 | 4.4e-17 | 3.3e-9 |
| 559 | 35550 | 412306 | 3.6e13 | 83.83 | 2.00 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 586 | 36609 | 214643 | 1.1e15 | 36.29 | 1.83 | 3.8e-17 | 6.5e-14 | 3.7e-17 | 4.0e-15 |
| 587 | 36609 | 214643 | 5.8e13 | 35.07 | 1.87 | 3.7e-17 | 2.1e-12 | 3.7e-17 | 1.3e-13 |
| 1316 | 37261 | 443573 | 2.2e10 | 43.76 | 1.86 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 1371 | 39899 | 195429 | 2.8e15 | 2.93 | 1.58 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 2815 | 40816 | 803978 | 4.4e13 | 411.55 | 1.76 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 578 | 41374 | 229385 | 1.6e15 | 31.55 | 2.48 | 2.5e-16 | 7.4e-4 | 6.8e-17 | 2.1e-5 |
| 561 | 41490 | 488633 | 3.4e13 | 131.50 | 2.10 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 588 | 45769 | 268563 | 1.7e15 | 48.49 | 1.89 | 3.7e-17 | 2.8e-11 | 3.7e-17 | 2.6e-12 |
| 589 | 45769 | 268563 | 5.8e13 | 46.75 | 1.84 | 3.7e-17 | 6.4e-10 | 3.7e-17 | 3.3e-11 |
| 563 | 47430 | 564952 | 1.1e14 | 130.58 | 2.03 | 3.5e-17 | 1.1e-16 | 3.5e-17 | 1.1e-16 |
| 1413 | 49702 | 333029 | 1.5e15 | 41.28 | 1.43 | 3.7e-17 | 4.2e-13 | 3.7e-17 | 8.4e-14 |
| 1414 | 49702 | 332807 | 4.0e13 | 17.81 | 1.39 | 3.7e-17 | 2.7e-15 | 3.7e-17 | 2.7e-16 |
| 1375 | 51032 | 247528 | 2.3e15 | 3.87 | 1.97 | 3.5e-17 | 2.6e-15 | 3.5e-17 | 2.9e-15 |
| 983 | 51993 | 380415 | 9.4e15 | 221.62 | 1.80 | 2.9e-17 | 1.1e-16 | 2.9e-17 | 1.1e-16 |
| 565 | 53370 | 641290 | 5.8e13 | 162.73 | 2.05 | 3.5e-17 | 1.4e-16 | 3.5e-17 | 1.1e-16 |
| 590 | 54929 | 322483 | 3.6e15 | 62.44 | 1.89 | 3.7e-17 | 1.3e-9 | 3.7e-17 | 3.7e-11 |
| 591 | 54929 | 322483 | 5.8e13 | 61.02 | 1.85 | 3.7e-17 | 5.6e-8 | 3.7e-17 | 2.1e-9 |
| 567 | 59310 | 717620 | 1.4e14 | 219.66 | 2.07 | 3.5e-17 | 1.7e-15 | 3.5e-17 | 1.2e-16 |
| 592 | 64089 | 376395 | 7.9e15 | 75.23 | 1.83 | 3.7e-17 | 6.5e-7 | 3.7e-17 | 9.5e-8 |
| 593 | 64089 | 376395 | 5.8e13 | 73.71 | 1.87 | 3.7e-17 | 1.5e-4 | 3.7e-17 | 1.8e-6 |
| 373 | 80209 | 307604 | 5.7e11 | 9.05 | 2.09 | 2.7e-17 | 3.6e-13 | 2.7e-17 | 3.1e-13 |
| 1374 | 87190 | 606489 | 2.0e15 | 17.88 | 2.11 | 3.7e-17 | 1.5e-16 | 3.7e-17 | 2.2e-16 |
| 2657 | 87936 | 593276 | 4.1e10 | 18.49 | 2.03 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 1343 | 94294 | 476766 | 5.6e12 | 17.26 | 2.48 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 1344 | 94294 | 479246 | 5.6e12 | 25.08 | 1.90 | 3.7e-17 | 1.1e-11 | 3.7e-17 | 7.7e-12 |
| 1345 | 94294 | 479151 | 5.6e12 | 20.01 | 2.65 | 3.7e-17 | 7.4e-14 | 3.7e-17 | 4.3e-14 |
| 919 | 16428 | 63406 | 1.2e14 | 2.50 | 1.37 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 2566 | 20468 | 206076 | 9.4e10 | 2.71 | 1.40 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 2567 | 40948 | 412148 | 9.4e10 | 5.98 | 1.63 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 288 | 14734 | 95053 | 9.6e3 | 4.17 | 1.92 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 289 | 25228 | 175027 | 5.3e3 | 7.92 | 1.61 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 290 | 84617 | 463625 | 7.5e4 | 22.22 | 2.10 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 2822 | 17922 | 561677 | 2.6e6 | 4.70 | 1.34 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 2823 | 32510 | 1030878 | 6.3e6 | 15.76 | 2.30 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 2824 | 56021 | 1797934 | 1.4e7 | 29.06 | 1.61 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 2825 | 100037 | 3226066 | 3.4e7 | 63.50 | 1.74 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 2826 | 178437 | 5778545 | 8.2e7 | 149.03 | 1.80 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 1415 | 99340 | 940621 | 1.5e11 | 25.01 | 2.10 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 1417 | 321821 | 1931828 | 5.1e22 | 110.07 | 4.57 | 3.7e-17 | 3.0e-16 | 3.7e-17 | 3.0e-16 |
| 1419 | 682862 | 2638997 | 9.5e19 | 768.65 | 2.57 | 4.5e-17 | 2.1e-9 | 4.5e-17 | 2.2e-9 |
| 326 | 2534 | 463360 | 5.2e5 | 20.08 | 1.62 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 1407 | 10605 | 522387 | 1.0e15 | 52.32 | 1.91 | NaN | NaN | 1.1e-16 | 1.8e-11 |
| 2555 | 37365 | 330633 | 2.7e5 | 60.08 | 1.87 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 2556 | 90249 | 803173 | 3.2e5 | 214.20 | 1.98 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |

*This manuscript is for review purposes only.*

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

| | "new" | "new0" |
|---|---|---|
| inclusions | failed in 2 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 | $7.8 \cdot 10^{-12}$ | $5.1 \cdot 10^{-12}$ |

810 is a bit tricky. Here *cnd* denotes the estimated condition number of the augmented
811 matrix in (6.1).
812 There were no complex examples in [5] satisfying (9.3). The conditions in (9.3)
813 lead to 26 test cases because most of the examples where either well-conditioned or
814 extremely ill-conditioned, often with condition number $\infty$. The results are displayed
815 in Table 16. As can be seen Algorithm `verifySparselss` failed for the two cases 1950
816 and 2055 of [5], Algorithm `verifySparselss0` failed only for the last case 2055.
817 There is not too much difference in computing for "new" and "new0". In the
818 median the computing times are almost the same, in the worst case "new" is 2.2
819 times faster than "new0", and "new0" is 1.3 times faster than "new".
820 A reason is that, in contrast to the square case, there is not much difference in
821 the fill-in of the factor $L_1$ because the majority of diagonal elements of the augmented
822 matrix (6.1) are already nonzero.
823 There is quite a spread in computing time between `lu` and our new algorithms.
824 In the median `lu` is 1.2 times faster than "new", but in the worst case "new" is 274
825 times faster than `lu`, and `lu` is 93 times faster than "new".
826 Both Algorithms "new" and "new0" compute always inclusions with maximal
827 accuracy for all entries of the solution. In contrast, the approximations by Matlab's
828 `lu` are significantly less accurate. The median and maximum relative errors of the
829 approximation by `lu` and "verifySparselss0" are displayed in Figure 8. As can be seen
830 in the median some 12 figures of the approximation by `lu` are correct, but in one case
831 only 4 digits of at least one entry of the approximation. In contrast, "verifySparselss0"
832 (and also "verifySparselss") compute almost always maximally accurate inclusions for
833 all entries.
834 Out of the ill-conditioned test cases satisfying (9.3) there were 37 matrices with
835 zero columns. That implies that the matrix is rank-deficient. When deleting those
836 columns there was a dichotomy. Either the matrices became well-conditioned, i.e.,
837 condition number less than $5 \cdot 10^7$, or, the matrices were still extremely ill-conditioned,
838 i.e., condition number larger than $3 \cdot 10^{20}$. In the former case it was no problem
839 to compute verified inclusions, the latter cases are out of the scope of verification
840 methods. Therefore, we refrain from giving additional computational results for those.
841 We finally show some test results for systems of nonlinear equations. The first
842 source of test examples stems from the MINPACK project [23]. The source code for
843 23 examples can be found at
844 `https://people.sc.fsu.edu/~jburkardt/m_src/test_nonlin/test_nonlin.html}}`
845 In 4 examples the dimension can be freely specified. In the first example $p01$ the
846 floating-point Newton iteration did not converge. For the other three example $p09$,
847 $p13$ and $p14$ we list computational results for different dimensions.

*This manuscript is for review purposes only.*

TABLE 16

*Timing and accuracy for sparse linear systems in [5] satisfying the conditions in* (9.3).

| | *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 | 0.307 | 2.005 | 0.999 | 2.0e-13 | 5.1e-9 | 4.0e-17 | 1.1e-16 | 4.0e-17 | 1.1e-16 |
| 615 | 5831 | 2171 | 33081 | 2.5e7 | 0.142 | 1.203 | 0.807 | 2.9e-15 | 7.4e-12 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 628 | 1706 | 1309 | 6937 | 3.7e8 | 0.030 | 0.281 | 0.194 | 4.6e-15 | 2.2e-11 | 3.9e-17 | 1.1e-16 | 3.9e-17 | 1.1e-16 |
| 697 | 23541 | 16675 | 72721 | 4.0e7 | 70.980 | 3.886 | 3.690 | 2.8e-14 | 1.1e-9 | 3.9e-17 | 1.1e-16 | 3.9e-17 | 1.1e-16 |
| 981 | 29493 | 11822 | 117954 | 1.7e10 | 21.915 | 13.537 | 14.729 | 1.9e-9 | 2.6e-5 | 4.4e-17 | 1.2e-14 | 4.0e-17 | 1.3e-15 |
| 1708 | 38602 | 24617 | 156466 | 1.3e9 | 158.681 | 498.115 | 577.639 | 2.9e-14 | 4.8e-9 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1713 | 16369 | 10099 | 44825 | 1.7e10 | 8.028 | 2.339 | 1.567 | 4.8e-15 | 5.6e-11 | 3.9e-17 | 1.1e-16 | 3.9e-17 | 1.1e-16 |
| 1731 | 16819 | 4400 | 150372 | 4.4e7 | 0.301 | 2.129 | 2.285 | 1.9e-14 | 3.6e-10 | 3.9e-17 | 1.1e-16 | 3.9e-17 | 1.1e-16 |
| 1737 | 8734 | 2301 | 68225 | 4.3e7 | 0.533 | 0.912 | 0.763 | 3.4e-15 | 2.5e-12 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1750 | 7967 | 2095 | 19826 | 6.4e8 | 0.754 | 0.520 | 0.526 | 1.3e-15 | 1.0e-12 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1756 | 1900 | 1650 | 8897 | 3.2e7 | 0.046 | 0.497 | 0.347 | 2.4e-14 | 1.8e-7 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1775 | 63076 | 3173 | 491336 | 4.4e7 | 19.070 | 3.003 | 2.647 | 1.3e-13 | 3.5e-10 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 1779 | 46937 | 6590 | 164538 | 4.3e11 | 48.686 | 3.701 | 4.133 | 3.0e-15 | 9.7e-12 | 3.6e-17 | 1.1e-16 | 3.6e-17 | 1.1e-16 |
| 1816 | 6654 | 3170 | 15397 | 5.7e7 | 0.013 | 0.563 | 0.491 | 5.1e-15 | 2.1e-11 | 3.7e-17 | 1.1e-16 | 3.7e-17 | 1.1e-16 |
| 1818 | 8617 | 4282 | 20635 | 1.1e8 | 0.032 | 0.588 | 0.565 | 6.6e-15 | 2.0e-10 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1827 | 13847 | 9743 | 35885 | 6.1e7 | 2.745 | 1.587 | 1.025 | 1.2e-15 | 6.3e-12 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1829 | 46679 | 32847 | 120141 | 6.5e8 | 332.492 | 5.376 | 2.416 | 1.9e-15 | 1.1e-11 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1834 | 27691 | 14364 | 58439 | 5.7e8 | 155.820 | 1.679 | 1.272 | 1.8e-15 | 1.2e-11 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1870 | 26722 | 11028 | 102432 | 1.6e7 | 34.109 | 1.559 | 1.499 | 1.7e-11 | 2.3e-7 | 3.9e-17 | 1.1e-16 | 3.9e-17 | 1.1e-16 |
| 1871 | 14318 | 11028 | 57376 | 6.3e7 | 32.080 | 2.552 | 1.267 | 8.2e-12 | 7.5e-8 | 3.9e-17 | 1.1e-16 | 3.9e-17 | 1.1e-16 |
| 1872 | 28634 | 11028 | 115262 | 1.6e7 | 33.919 | 1.851 | 1.595 | 1.6e-11 | 7.1e-8 | 4.0e-17 | 1.1e-16 | 4.0e-17 | 1.1e-16 |
| 1947 | 1302 | 1121 | 11185 | 5.0e7 | 0.023 | 0.730 | 0.923 | 1.2e-13 | 5.7e-11 | 3.9e-17 | 1.1e-16 | 3.9e-17 | 1.1e-16 |
| 1948 | 3160 | 2644 | 29862 | 3.1e8 | 0.113 | 6.832 | 8.211 | 1.5e-12 | 1.2e-8 | 3.8e-17 | 1.1e-16 | 3.8e-17 | 1.1e-16 |
| 1949 | 7742 | 6334 | 80057 | 2.3e9 | 1.103 | 302.484 | 211.135 | 1.3e-11 | 8.3e-9 | 3.9e-17 | 1.1e-16 | 3.9e-17 | 1.1e-16 |
| 1950 | 19321 | 15437 | 216173 | 1.5e10 | 14.993 | 5131.570 | 2067.807 | 1.5e-11 | 4.7e-7 | NaN | NaN | 3.9e-17 | 1.1e-16 |
| 2055 | 3003 | 1716 | 12012 | 9.0e15 | 0.167 | 34.050 | 91.822 | NaN | NaN | NaN | NaN | NaN | NaN |

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

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

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

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

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

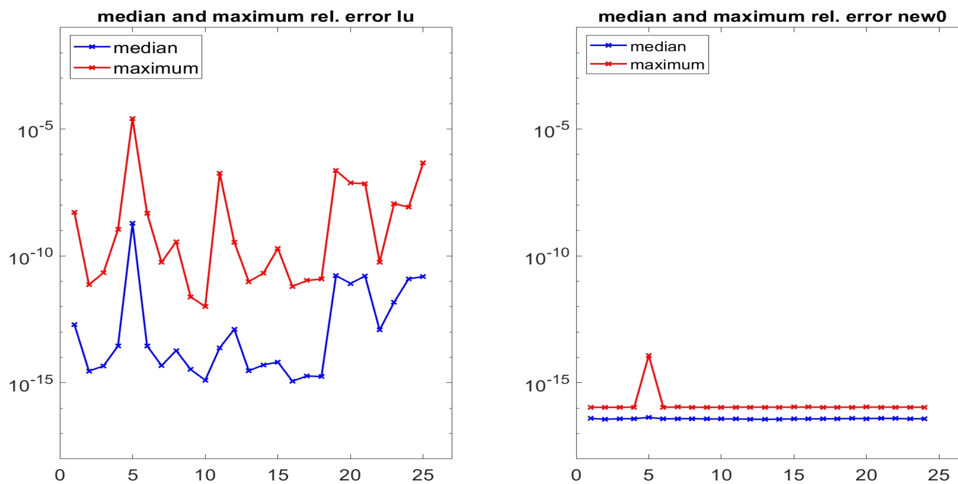


FIG. 8. *Median of relative errors of* `verifySparselss` *and* `verifySparselss0`*.*

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 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. All of our 5 examples are solved successfully by **verifynlss**, but larger dimensions are prohibitive for our laptop.

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` | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $t_{new}$ | $t_{new0}$ | $t_{\texttt{fsolve}}$ | *iter* | *median* | *max* | *iter* | *median* | *max* | *median* | *max* |
| *p*09 | 1,000 | 0.1 | 0.1 | 0.4 | 5/2 | 3.0e-11 | 1.6e-10 | 5/2 | 5.9e-11 | 1.6e-10 | 0.20 | 0.33 |
| | 10,000 | 0.2 | 0.2 | 5.5 | 5/2 | 3.0e-9 | 9.6e-9 | 5/2 | 5.9e-9 | 1.2e-8 | 0.34 | 0.50 |
| | 100,000 | 1.2 | 1.2 | | 6/2 | 3.1e-7 | 1.3e-6 | 6/2 | 6.3e-7 | 1.3e-6 | out of memory | |
| | 1,000,000 | 14.9 | 14.8 | | 10/2 | 3.0e-5 | 1.5e-4 | 10/2 | 5.9e-5 | 1.5e-4 | out of memory | |
| | 10,000,000 | 215.9 | 231.0 | | 13/2 | 3.0e-3 | 9.6e-3 | 13/2 | 6.0e-3 | 1.2e-2 | out of memory | |
| *p*13 | 1,000 | 0.1 | 0.1 | 0.4 | 6/1 | 6.3e-16 | 2.0e-15 | 6/3 | 9.5e-16 | 2.8e-15 | 3.2e-15 | 7.5e-14 |
| | 10,000 | 0.2 | 0.2 | 24.6 | 6/1 | 6.3e-16 | 2.0e-15 | 6/2 | 9.5e-16 | 2.8e-15 | 5.9e-13 | 1.9e-11 |
| | 100,000 | 1.1 | 2.1 | | 6/1 | 6.3e-16 | 2.0e-15 | 6/2 | 9.5e-16 | 2.8e-15 | out of memory | |
| | 1,000,000 | 11.5 | 21.5 | | 6/1 | 6.3e-16 | 2.0e-15 | 6/2 | 9.5e-16 | 2.8e-15 | out of memory | |
| | 10,000,000 | 134.2 | | | 6/1 | 6.3e-16 | 2.0e-15 | | failed | | out of memory | |
| *p*14 | 1,000 | 0.6 | 0.6 | 1.9 | 7/1 | 7.2e-16 | 1.5e-15 | 7/2 | 1.5e-15 | 3.4e-15 | 1.5e-15 | 7.0e-9 |
| | 10,000 | 0.5 | 0.2 | 149.2 | 7/1 | 7.2e-16 | 1.5e-15 | 7/2 | 1.5e-15 | 5.4e-16 | 5.4e-16 | 1.1e-14 |
| | 100,000 | 7.0 | 34.2 | | 7/1 | 1.5e-15 | 2.9e-15 | 7/2 | 2.9e-15 | 8.0e-15 | out of memory | |
| | 1,000,000 | 88.8 | 2264.6 | | 7/1 | 1.5e-15 | 2.9e-15 | 7/2 | 2.9e-15 | 8.0e-15 | out of memory | |
| | 10,000,000 | 7088.9 | | | 7/1 | 1.5e-15 | 2.9e-15 | | failed | | out of memory | |
| MC | 1,000 | 0.2 | 0.2 | 0.4 | 4/2 | 1.1e-14 | 8.1e-10 | 4/2 | 1.6e-15 | 6.4e-10 | 0.0014 | 0.0017 |
| | 10,000 | 0.3 | 0.4 | 27.9 | 4/2 | 1.3e-13 | 8.4e-8 | 4/2 | 7.9e-15 | 6.6e-8 | 0.0011 | 0.0013 |
| | 100,000 | | 16.1 | | | failed | | 4/2 | 1.2e-12 | 9.2e-6 | out of memory | |
| | 1,000,000 | | 2079.7 | | | failed | | 5/2 | 1.1e-11 | 6.1e-4 | out of memory | |
| | 10,000,000 | | | | | failed | | | failed | | out of memory | |
| AB | 100 | 0.7 | 0.5 | 0.4 | 10/2 | 3.8e-16 | 1.6e-13 | 10/2 | 5.1e-16 | 4.4e-14 | 1.1e-12 | 1.2e-11 |
| | 1,000 | 0.2 | 0.2 | 0.9 | 12/2 | 3.9e-16 | 2.6e-11 | 12/2 | 5.5e-16 | 8.4e-13 | 2.1e-9 | 2.2e-7 |
| | 5,000 | 0.4 | 0.4 | 22.1 | 14/3 | 3.9e-16 | 7.8e-10 | 14/2 | 7.8e-16 | 1.3e-11 | 2.0e-7 | 1.3e-5 |
| | 9,000 | 1.0 | 0.7 | 67.0 | 15/5 | 3.9e-16 | 2.7e-9 | 15/2 | 1.0e-15 | 3.9e-11 | 6.7e-6 | 0.0037 |
| | 10,000 | | 0.8 | | | failed | | 15/2 | 8.7e-16 | 7.2e-11 | out of memory | |
| | 30,000 | | 5.5 | | | failed | | 15/3 | 1.3e-15 | 1.4e-10 | out of memory | |
| | 50,000 | | 14.4 | | | failed | | 15/3 | 1.6e-14 | 1.5e-9 | out of memory | |
| | 60,000 | | | | | failed | | | failed | | out of memory | |

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 problem "MC" the approximation is in the median accurate to some 3 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 $p09$, $p13$ 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 $p14$.

Contrary, for problems MC and 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. The median relative error of the inclusion does not change much, but the maximal error increases. However, that is basically due to the first entries of the solution with small magnitude.

**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.

*This manuscript is for review purposes only.*

922 REFERENCES


923 [1] J.P. Abbott and R.P. Brent. Fast Local Convergence with Single and Multistep Methods for
924 Nonlinear Equations. *Austr. Math. Soc. 19 (Series B)*, pages 173–199, 1975.
925 [2] P. Ahrens, J. Demmel, and H.D. Nguyen. Algorithms for efficient reproducible floating-point
926 summation. *ACM TOMS*, 46:1–49, 2020.
927 [3] I.J. Anderson. A distillation algorithm for floating-point summation. *SIAM J. Sci. Comput.*,
928 20:1797–1806, 1999.
929 [4] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Nauman. *Automatic Differentiation of*
930 *Algorithms – From Simulation to Optimisation*. Springer-Verlag, Berlin, 2002.
931 [5] T.A. Davis, Y. Hu: The University of Florida Sparse Matrix Collection. ACM Transactions on
932 Mathematical Software 38, 1, Article 1, 2011.
933 [6] J.B. Demmel. On floating point errors in Cholesky. LAPACK Working Note 14 CS-89-87,
934 Department of Computer Science, University of Tennessee, Knoxville, TN, USA, 1989.
935 [7] J. Demmel, Y. Hida. Accurate and efficient floating point summation. *SIAM J. Sci. Comput.*
936 *(SISC)*, 25:1214–1248, 2003.
937 [8] I.S. Duff, J. Koster. On algorithms for permuting large entries to the diagonal of a sparse
938 matrix. *SIAM Journal on Matrix Analysis and Applications (SIMAX)*, 22 (4):973–996,
939 2001.
940 [9] Iain S. Duff. Ma57—a code for the solution of sparse symmetric definite and indefinite systems.
941 *ACM Trans. Math. Softw.*, 30(2):118–144, 2004.
942 [10] A. Griewank. A Mathematical View of Automatic Differentiation. In *Acta Numerica*, volume 12,
943 pages 321–398. Cambridge University Press, 2003.
944 [11] N. J. Higham: *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadel-
945 phia, 2nd edition, 2002.
946 [12] P. Holoborodko. *Multiprecision Computing Toolbox for MATLAB 4.6.4.13348*. Advanpix LLC.,
947 Yokohama, Japan, 2019.
948 [13] IEEE Standard for Floating-point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-*
949 *2008)*, pages 1–84, 2019.
950 [14] D. E. Knuth: *The Art of Computer Programming: Seminumerical Algorithms*, volume 2.
951 Addison Wesley, Reading, Massachusetts, 1969.
952 [15] S.P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T.A. Davis, M. Henderson, Y. Hu, R.
953 Sandstrom: The SuiteSparse Matrix Collection Website Interface. Journal of Open Source
954 Software 4, 35, 1244-1248, 2019.
955 [16] C.-P. Jeannerod, S.M. Rump. Improved error bounds for inner products in floating-point
956 arithmetic. *SIAM J. Matrix Anal. Appl. (SIMAX)*, 34(2):338–344, 2013.
957 [17] M. Lange and S.M. Rump. Error estimates for the summation of real numbers with application
958 to floating-point summation. *BIT*, 57:927–941, 2017.
959 [18] M. Lange, S.M. Rump. Sharp estimates for perturbation errors in summations. *Math.Comp.*,
960 88:349–368, 2019.
961 [19] M. Lange and S.M. Rump. Floating-point matrix products with improved accuracy part I:
962 theoretical background. to appear.
963 [20] M. Lange and S.M. Rump. Floating-point matrix products with improved accuracy part II:
964 Schemes for matrix products. to appear.
965 [21] MATLAB. User's Guide, Version 2023b, the MathWorks Inc., 2023.
966 [22] J.J. Moré and M.Y. Cosnard. Numerical solution of non-linear equations. *ACM Trans. Math.*
967 *Software*, 5:64–85, 1979.
968 [23] J.J. Moré, D.C. Sorensen, K.E. Hillstrom, and B.S. Garbow. The MINPACK project. In W.J.
969 Cowell, editor, *Sources and Development of Mathematical Software*, pages 88–111. Prentice
970 Hall, 1984.
971 [24] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond,
972 R. Revol,, S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2nd
973 edition, 2018.
974 [25] A. Neumaier. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen.
975 *Zeitschrift für Angew. Math. Mech. (ZAMM)*, 54:39–51, 1974.
976 [26] A. Neumaier: *Interval methods for systems of equations*. Encyclopedia of Mathematics and its
977 Applications. Cambridge University Press, 1990.
978 [27] A. Neumaier. Grand challenges and scientific standards in interval analysis. *Reliable Computing*,
979 8(4):313–320, 2002.
980 [28] T. Ogita, S. M. Rump, S. Oishi: Accurate sum and dot product. *SIAM Journal on Scientific*
981 *Computing (SISC)*, 26(6):1955–1988, 2005.
982 [29] S. Oishi, K. Ichihara, M. Kashiwagi, T. Kimura, X. Liu, H. Masai, Y. Morikura, T. Ogita,

983 K. Ozaki, S. M. Rump, K. Sekine, A. Takayasu, N. Yamanaka: *Principle of Verified*
984 *Numerical Computations*. Corona Publisher, Tokyo, Japan, 2018. [in Japanese].
985 [30] K. Ozaki, T. Ogita, and S. Oishi. Tight and efficient enclosure of matrix multiplication by
986 using optimized BLAS. *Numerical Linear Algebra with Applications*, 18(2):237–248, 2011.
987 [31] K. Ozaki, T. Ogita, and S. Oishi. Improvement of error-free splitting for accurate matrix
988 multiplication. *Journal of Computational and Applied Mathematics*, 288:127–140, 2015.
989 [32] K. Ozaki, T. Ogita, and S. Oishi. Error-free transformation of matrix multiplication with a
990 posteriori validation. *Numerical Linear Algebra with Applications*, 23(5):931–946, 2016.
991 [33] K. Ozaki, T. Ogita, S.M. Rump, and S. Oishi. Fast algorithms for floating-point interval matrix
992 multiplication. *Journal of Computational and Applied Mathematics*, 236(7):1795–1814,
993 2012.
994 [34] K. Ozaki, T. Ogita, S. Oishi, and S.M. Rump. Error-free transformations of matrix multi-
995 plication by using fast routines of matrix multiplication and its applications. *Numerical*
996 *Algorithms*, 59(1):95–118, 2012.
997 [35] K. Ozaki, T. Ogita, S. Oishi, and S.M. Rump. Generalization of Error-Free Transformation
998 for Matrix Multiplication and its Application. *Nonlinear Theory and its Applications*,
999 4(1):2–11, 2013.
1000 [36] S.M. Rump. *Kleine Fehlerschranken bei Matrixproblemen*. PhD thesis, Universität Karlsruhe,
1001 1980.
1002 [37] S.M. Rump. Validated Solution of Large Linear Systems. In R. Albrecht, G. Alefeld, H.J.
1003 Stetter, editors, *Validation numerics: theory and applications*, volume 9 of *Computing*
1004 *Supplementum*, pages 191–212. Springer, 1993.
1005 [38] S.M. Rump. Verified Computation of the Solution of Large Sparse Linear Systems. *Zeitschrift*
1006 *für Angewandte Mathematik und Mechanik (ZAMM)*, 75:S439–S442, 1995.
1007 [39] S. M. Rump: INTLAB – INTerval LABoratory. In Tibor Csendes, editor, *Developments in*
1008 *Reliable Computing*, pages 77–104. Springer Netherlands, Dordrecht, 1999.
1009 [40] S.M. Rump. Verified Solution of Large Linear and Nonlinear Systems. In H. Bulgak, C.
1010 Zenger, editors, *Error Control and adaptivity in Scientific Computing*, pages 279–298.
1011 Kluwer Academic Publishers, 1999.
1012 [41] S. M. Rump: Verification methods: Rigorous results using floating-point arithmetic. *Acta*
1013 *Numerica*, 19:287–449, 2010.
1014 [42] S.M. Rump. Improved componentwise verified error bounds for least squares problems and
1015 underdetermined linear systems. 66:309–322, 2013.
1016 [43] S.M. Rump, T. Ogita. Super-fast validated solution of linear systems. *Journal of Computa-*
1017 *tional and Applied Mathematics (JCAM)*, 199(2):199–206, 2006. Special issue on Scientific
1018 Computing, Computer Arithmetic, and Validated Numerics (SCAN 2004).
1019 [44] S.M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful round-
1020 ing. *SIAM J. Sci. Comput. (SISC)*, 31(1):189–224, 2008.
1021 [45] R. Skeel. Iterative Refinement Implies Numerical Stability for Gaussian Elimination. *Math.*
1022 *Comp.*, 35(151):817–832, 1980.
1023 [46] Terao T., K. Ozaki. Method for verifying solutions of sparse linear systems with general
1024 coefficients. 2024. `https://arxiv.org/abs/2406.02033`.
1025 [47] G. Zielke, V. Drygalla. Genaue Lösung linearer Gleichungssysteme. *GAMM Mitt. Ges. Angew.*
1026 *Math. Mech.*, 26:7–108, 2003.