Parallel Implementation of a Block Algorithm for Matrix 1-Norm Estimation*

Sheung Hun Cheng[1] and Nicholas J. Higham[2]

[1] Centre for Novel Computing, Department of Computer Science, University of Manchester
Manchester, M13 9PL, England
scheng@cs.man.ac.uk
http://www.cs.man.ac.uk/~scheng/

[2] Department of Mathematics, University of Manchester
Manchester, M13 9PL, England
higham@ma.man.ac.uk
http://www.ma.man.ac.uk/~higham/

Abstract. We describe a parallel Fortran 77 implementation, in ScaLAPACK style, of a block matrix 1-norm estimator of Higham and Tisseur. This estimator differs from that underlying the existing ScaLAPACK code, `PxLACON`, in that it iterates with a matrix with t columns, where $t \geq 1$ is a parameter, rather than with a vector, and so the basic computational kernel is level 3 BLAS operations. Our experiments on an SGI Origin2000 show that with $t = 2$ or 4 the new code offers better estimates than `PDLACON` with a similar execution time. Moreover, with $t > 4$, estimates exact over 90% of the time are achieved with execution time growing much slower than t.

1 Introduction

Error bounds for computed solutions to linear systems, least squares and eigenvalue problems all involve condition numbers, which measure the sensitivity of the solution to perturbations in the data. Thus, condition numbers are an important tool for assessing the quality of the computed solutions. Typically, these condition numbers are as expensive to compute as the solution itself [6]. The LAPACK [1] and ScaLAPACK [2] condition numbers and error bounds are based on estimated condition numbers, using the method of Hager [3], which was subsequently improved by Higham [4]. Hager's method estimates $\|B\|_1$ given only the ability to compute matrix-vector products Bx and $B^T y$. If we take $B = A^{-1}$ and compute the required products by solving linear systems with A, we obtain an estimate of the 1-norm condition number $\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$.

In LAPACK and ScaLAPACK Higham's version of Hager's method is implemented in routines `xLACON` and `PxLACON`, respectively. Both routines have a

* This work was supported by Engineering and Physical Sciences Research Council grant GR/L94314. The work of the second author was also supported by a Royal Society Leverhulme Trust Senior Research Fellowship.

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 568–577, 2001.

reverse communication interface. There are two advantages to having such an interface. First it provides flexibility, as the dependence on B and its associated matrix-vector operations is isolated from the computational routines `xLACON` and `PxLACON`, with the matrix-vector products provided by a "black box" [4]. By changing these black boxes, `xLACON` and `PxLACON` can be applied to different matrix functions for both dense and sparse matrices. Second, as the bulk of the computational effort is in matrix-vector operations, efficient implementation of these operations ensures good overall performance of `xLACON` and `PxLACON`, and thus a focus is provided for performance tuning.

The price to pay for using an estimate instead of the exact condition number is that it can sometimes be a poor estimate. Experiments in [4] show that the underestimation is rarely by more than a factor of 10 (the estimate is, in fact, a lower bound), which is acceptable in practice as it is the magnitude of the condition number that is of interest. However, counterexamples for which the condition numbers can be arbitrarily poor estimates exist [4], [5]. Moreover, when the accuracy of the estimates becomes important for certain applications [7], the method does not provide an obvious way to improve the estimate.

Higham and Tisseur [7] present a block generalization of the estimator of [3,4] that iterates with an $n \times t$ matrix, where $t \geq 1$ is a parameter, enabling the exploitation of matrix-matrix operations (level 3 BLAS) and thus promising greater efficiency and parallelism. The block algorithm also offers the potential of better estimates and a faster convergence rate, through providing more information on which to base decisions. Moreover, part of the starting matrix is randomly formed, which introduces a stochastic flavour and reduces the importance of counterexamples.

We have implemented this block algorithm using Fortran 77 in the ScaLAPACK programming style and report performance on a 16 processor SGI Origin2000. The rest of the paper is organized as follows. We describe the block 1-norm estimator in Section 2. In Section 3 we present and explain details of our parallel implementation of the estimator. The performance of the implementation is evaluated in Section 4. Finally, we summarize our findings in Section 5.

2 Block 1-Norm Estimator

In this section we give pseudo-code for the block 1-norm estimator, which is basically a block power method for the matrix 1-norm. See [7] for a derivation and explanation of the algorithm. We use MATLAB array and indexing notation [8].

Algorithm 1 (block 1-norm estimator) *Given $A \in \mathbb{R}^{n\times n}$ and positive integers t and* itmax ≥ 2*, this algorithm computes a scalar* est *and vectors v and w such that* est $\leq \|A\|_1$*, $w = Av$ and $\|w\|_1 = \text{est}\|v\|_1$.*

Choose starting matrix $X \in \mathbb{R}^{n\times t}$ with columns of unit 1-norm.
ind_hist = [ ] % Integer vector recording indices of used unit vectors e_j.
$\text{est}_{\text{old}} = 0$, ind = zeros$(n, 1)$, S = zeros(n, t)
for $k = 1, 2, \ldots$

(1) $Y = AX$
est $= \max\{\, \|Y(:,j)\|_1 : j = 1{:}t \,\}$
if est $>$ est$_{\text{old}}$ or $k = 2$
ind_best $=$ ind$_j$ where est $= \|Y(:,j)\|_1$, $w = Y(:,\text{ind\_best})$
end
if $k \geq 2$ and est $\leq$ est$_{\text{old}}$, est $=$ est$_{\text{old}}$, goto (5), end
est$_{\text{old}}$ $=$ est, $S_{\text{old}} = S$
(2) if $k >$ itmax, goto (5), end
$S = \text{sign}(Y)$ % sign$(x) = 1$ if $x \geq 0$ else -1
If every column of S is parallel to a column of S_{old}, goto (5), end
if $t > 1$
(3) Ensure that no column of S is parallel to another column of S
or to a column of S_{old} by replacing columns of S by rand$\{-1, 1\}$.
end
(4) $Z = A^T S$
$h_i = \|Z(i,:)\|_\infty$, ind$_i = i$, $i = 1{:}n$
if $k \geq 2$ and $\max(h_i) = h_{\text{ind\_best}}$, goto (5), end
Sort h so that $h_1 \geq \cdots \geq h_n$ and re-order ind correspondingly.
if $t > 1$
If ind$(1{:}t)$ is contained in ind_hist, goto (5), end
Replace ind$(1{:}t)$ by the first t indices in ind$(1{:}n)$ that are
not in ind_hist.
end
$X(:,j) = e_{\text{ind}_j}$, $j = 1{:}t$
ind_hist $=$ [ind_hist ind$(1{:}t)$]
end
(5) $v = e_{\text{ind\_best}}$

Statements (1) and (4) are the most expensive parts of the computation and are where a reverse communication interface is employed. It is easily seen that if statements (1) and (4) are replaced by "Solve $AY = X$ for Y" and "Solve Z for $A^T Z = S$ for Z", respectively, then Algorithm 1 estimates $\|A^{-1}\|_1$.

MATLAB 6 contains an implementation of Algorithm 1 in function `normest1`, which is used by the condition number estimation function `condest`.

3 Parallel Implementation

We have implemented Algorithm 1 in double precision using Fortran 77 in the ScaLAPACK programming style. For dense matrices, ScaLAPACK assumes the data to be distributed according to the *two-dimensional block-cyclic* data layout scheme; see Figure 1 for an example. Our code uses the highest level of BLAS and PBLAS whenever possible.

In `PxLACON` the vectors resulting from the matrix-vector operations are always stored in the first process column. In order to ensure all processes follow the same execution path, the resulting vectors are copied to every process column. We

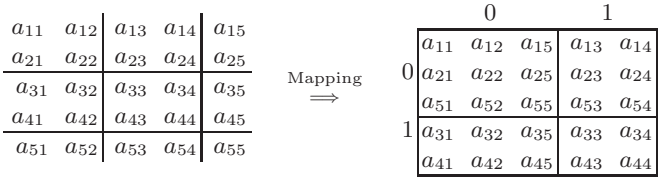

Fig. 1. A 5×5 matrix decomposed into 2×2 blocks mapped onto a 2×2 process grid using the two-dimensional block-cyclic data layout scheme

have adopted a similar approach in our implementation in that we assume all t search vectors are stored in the first process column. Consequently, the maximum value of t is equal to the column block size for partitioning the matrix. This restriction on t eliminates a large amount of communication between process columns, which in our experience is very costly. This restriction is not severe, as good norm estimates are obtained even with t relatively small compared with the column block size. Moreover, instead of copying the search vectors across process columns, which becomes increasingly expensive as t increases, the first process column performs all the computational work and then broadcasts two scalar variables (`est` in Algorithm 1 and an integer variable used in the reverse communication mechanism) across the process columns to ensure all processes follow the same execution path. This provides another large saving in communication cost. Furthermore, we arrange that S (the current sign matrix, whose elements are ± 1) and S_{old} (the previous sign matrix) share the same distribution scheme.

We set the maximum number of iterations itmax to 5, which is rarely reached. When this limit is reached we have, in fact, performed $5\frac{1}{2}$ iterations, as the test (2) in Algorithm 1 comes after the matrix product $Y = AX$. This allows us to make use of the new search direction generated at the end of the fifth iteration.

Most of Algorithm 1 is straightforwardly translated into Fortran code apart from statement (3), which deserves detailed explanation. Statement (3) is a novel feature of Algorithm 1 in which parallel columns within the current sign matrix S and between S and S_{old} are replaced by rand$\{-1, 1\}$, where rand$\{-1, 1\}$ denotes a random vector with entries -1 or 1. The replacement of parallel columns avoids redundant computation and may lead to a better estimate [7]. The detection of parallel columns is done by forming inner products between columns and looking for elements of magnitude n. Obviously, we should only check for parallel columns when $t > 1$. Using the notation of Algorithm 1, statement (3) is implemented as follows:

$$
\begin{array}{ll}
 & \text{iter} = 0 \\
 & \text{for } i = 1{:}\,t \\
 & \qquad \text{while iter} < n/t \\
(A) & \qquad\qquad y = S_{\text{old}}^T S(:, i) \\
 & \qquad\qquad \text{iter} = \text{iter} + 1
\end{array}
$$

```
                    if ‖y‖∞ < n
(B)                     y = S(:, 1:i − 1)^T S(:, i)
                        iter = iter + 1
                        if ‖y‖∞ < n, goto (#), end
                    end
                    S(:, i) = rand{−1, 1}
                end
(#)  end
```

In the inner loop the number of matrix-vector products is limited to n/t. As the computational cost of Algorithm 1 is $O(n^2t)$ flops and (A) and (B) both cost $O(nt)$ flops, this choice of limit ensures that the cost of replacing parallel columns does not dominate the overall cost.

The replacement of parallel columns involves the use of a random number generator and ScaLAPACK is intended to work in a heterogeneous environment. Our restriction that the first process column performs all the computations of random numbers ensures that the code works properly in a heterogeneous environment. Furthermore, by keeping a single copy of the search vectors, rather than multiple copies on each process column as in ScaLAPACK, we eliminate a potential performance-degrading communication overhead.

The new code is called `PDLACON1`. Its storage requirements are not significantly greater than those of `PxLACON`, unless t is large. Let m denote the number of process rows in the process grid. To be precise, `PDLACON1` and `PDLACON` require $(2nt + n)/m + 4n + t$ and $3n/m$ storage space per process respectively.

Note that the replacement of parallel columns is not essential for Algorithm 1 to produce good estimates. The advantage of this feature in the distributed context is less clear than in the serial and shared-memory cases. We include this feature in our implementation so that it is consistent with our MATLAB and LAPACK-style versions.

4 Numerical Experiments

In this section, our aim is to examine the performance of `PDLACON1`. We have addressed the issues of accuracy and reliability of `PDLACON1` by reproducing parts of the experimental results in Higham and Tisseur [7], thereby validating our code. Our main focus in this section is to measure the efficiency of our implementation. We investigate how the relation between accuracy and execution time varies with n and t.

We tested `PDLACON1` on a SGI Origin2000. The compiler and library details are given in Table 1.

We use the same compiler flags as was used to compile the ScaLAPACK installation. This provides a basis for measuring and comparing the performance of our implementation with that of the ScaLAPACK routine `PDLACON`. All timing results were obtained on a shared machine with exclusive access to a set of consecutive CPUs and their memory using the IRIX command `cpuset`; see

Table 1. Characteristics of the SGI Origin2000, libraries and compiler options for the experiments

Compiler	Compiler Flags	SGI BLAS	Precision
MIPSpro Compilers: Version 7.3.1.2m (`f77`)	`-O2 -64 -mips4 -r10000`	`-lblas` (optimized)	double
ScaLAPACK version 1.6	MPI BLACS version1.1 + patch	PBLAS v2.0	SGI MPI `-lmpi`

Table 2. Parameters set in the configuration file of `cpuset`

Parameters
`EXCLUSIVE`, `MEMORY_LOCAL`, `MEMORY_EXCLUSIVE`, `MEMORY_KERNEL_AVOID`, `MEMORY_MANDATORY`, `POLICY_KILL`

Table 2 for the configuration of `cpuset`. Both row and column block sizes for partitioning the matrix were set to 64.

We estimate $\|A^{-1}\|_1$ for random matrices $A \in \mathbb{R}^{n\times n}$ with $1200 \le n \le 2700$. For each n, a total of 500 random matrices A are generated, variously from the uniform $(0,1)$, uniform $(-1,1)$ or normal $(0,1)$ distributions. The LU factorization with partial pivoting of A is supplied to the 1-norm estimators. The cost of this part of computation does not contribute to the overall timing result. This arrangement is reasonable as the LU factorization is usually readily available in practice, as when solving a linear system, for example. The inverse of A is computed explicitly to obtain the "exact" $\|A^{-1}\|_1$. For a given matrix A we first generated a starting matrix X_1 with 64 columns, where 64 is the largest value of t to be used, and then ran Algorithm 1 for $t = 1, 2, \ldots, 64$ using starting matrix $X_1(:, 1\!:\!t)$. In this way we could see the effect of increasing t with fixed n. Each set of tests was repeated on 2, 4, 6, 8, 12 processors with different process grids. For example, we ran tests using 4 processors on 1×4, 2×2 and 4×1 process grids.

For each test matrix we recorded a variety of statistics in which the subscripts min, max and an overbar denote the minimum, maximum, and average of a particular measure respectively:

- α: the underestimation ratio $\alpha = \text{est}/\|A^{-1}\|_1 \le 1$, over each A for fixed t.
- %E: the percentage of estimates that are exact. An estimate is regarded as exact if the relative error $|\text{est} - \|A^{-1}\|_1|/\|A^{-1}\|_1$ is no larger than 10^{-14} (the unit roundoff is of order 10^{-16}).
- %I: for a given t, the percentage of estimates that are at least as large as the estimates for all smaller t.
- %A: For a given t, the percentage of the estimates that are at least as large as the estimates from `PDLACON`.

Table 3. Experimental results of 500 random matrices with dimensions $n = 1200$ and $n = 2700$ on a 2×2 process grid

$n = 1200$														
t	$\alpha_{\min}$	$\overline{\alpha}$	%E	%I	%A	%T	$N_{\max}$	$\overline{N}$	$C_{\max}$	$\overline{C}$	$D_{\max}$	$\overline{D}$	$K_{\max}$	$\overline{K}$
1[a]	0.29	0.98	84.8	–	–	–	–	–	1.00	0.71	23.26	20.25	9	5.4
1	0.29	0.98	84.8	–	100.0	0.0	0.94	0.83	7.85	5.15	29.40	28.25	8	4.4
2	0.59	0.99	90.8	98.6	98.6	1.6	1.37	0.88	5.38	5.07	28.65	27.40	6	4.1
4	0.67	1.00	96.0	98.8	99.4	83.2	1.62	1.02	5.41	4.83	25.94	24.91	6	4.0
8	0.97	1.00	98.4	99.8	100.0	97.6	1.80	1.37	5.45	4.50	22.45	21.19	4	4.0
16	0.94	1.00	98.6	99.6	100.0	100.0	2.89	2.20	5.54	4.08	21.02	19.42	4	4.0
32	1.00	1.00	99.0	100.0	100.0	100.0	4.81	3.67	6.01	4.38	18.25	16.36	4	4.0
64	1.00	1.00	99.0	100.0	100.0	100.0	8.97	6.81	6.97	4.85	16.62	14.39	4	4.0
$n = 2700$														
t	$\alpha_{\min}$	$\overline{\alpha}$	%E	%I	%A	%T	$N_{\max}$	$\overline{N}$	$C_{\max}$	$\overline{C}$	$D_{\max}$	$\overline{D}$	$K_{\max}$	$\overline{K}$
1[a]	0.67	0.99	80.2	–	–	–	–	–	0.37	0.32	13.64	11.99	11	5.4
1	0.67	0.99	80.2	–	100.0	0.0	0.94	0.83	3.37	2.97	17.58	16.88	10	4.4
2	0.76	1.00	86.0	98.4	98.4	2.2	1.29	0.83	3.14	3.07	17.90	17.22	8	4.1
4	0.89	1.00	91.0	99.4	99.6	0.8	1.48	0.94	3.08	2.99	16.67	16.06	6	4.0
8	0.94	1.00	92.0	99.2	99.6	83.0	1.97	1.25	2.90	2.72	14.07	13.63	6	4.0
16	0.96	1.00	92.6	99.8	100.0	99.8	2.10	1.97	2.65	2.44	13.20	12.72	4	4.0
32	1.00	1.00	92.8	100.0	100.0	100.0	3.58	3.36	2.97	2.53	11.82	11.26	4	4.0
64	1.00	1.00	92.8	100.0	100.0	100.0	6.96	6.53	3.35	2.77	10.90	10.21	4	4.0

[a] Data for `PDLACON`

- %T: the percentage of the cases for which our implementation took longer to complete than `PDLACON`.
- N: The execution time for `PDLACON1` normalized against the time taken by `PDLACON`.
- C: the percentage of time spent in `PDLACON1` for a given A on the leading process column.
- D: the percentage of time spent in `PDLACON1` for a given A on non-leading process columns.
- K: the number of matrix-matrix operations for a given A.

We present a subset of experimental results that capture general characteristics of the performance of `PDLACON1`. In Table 3 we show detailed statistical results for a 2×2 process grid. In Figure 2 we compare the performance when running the experiments on the same number of processors but different process grids. We make the following comments.

- Increasing t usually improves the quality of the estimates. However, this is not always true as %I is not monotonic increasing. Nevertheless, estimates exact over 90% of the time can be computed with t relatively small compared with n. Fast convergence, which is not explained by the underlying theory, is recorded throughout the experiments. All these observations are consistent with those in [7].

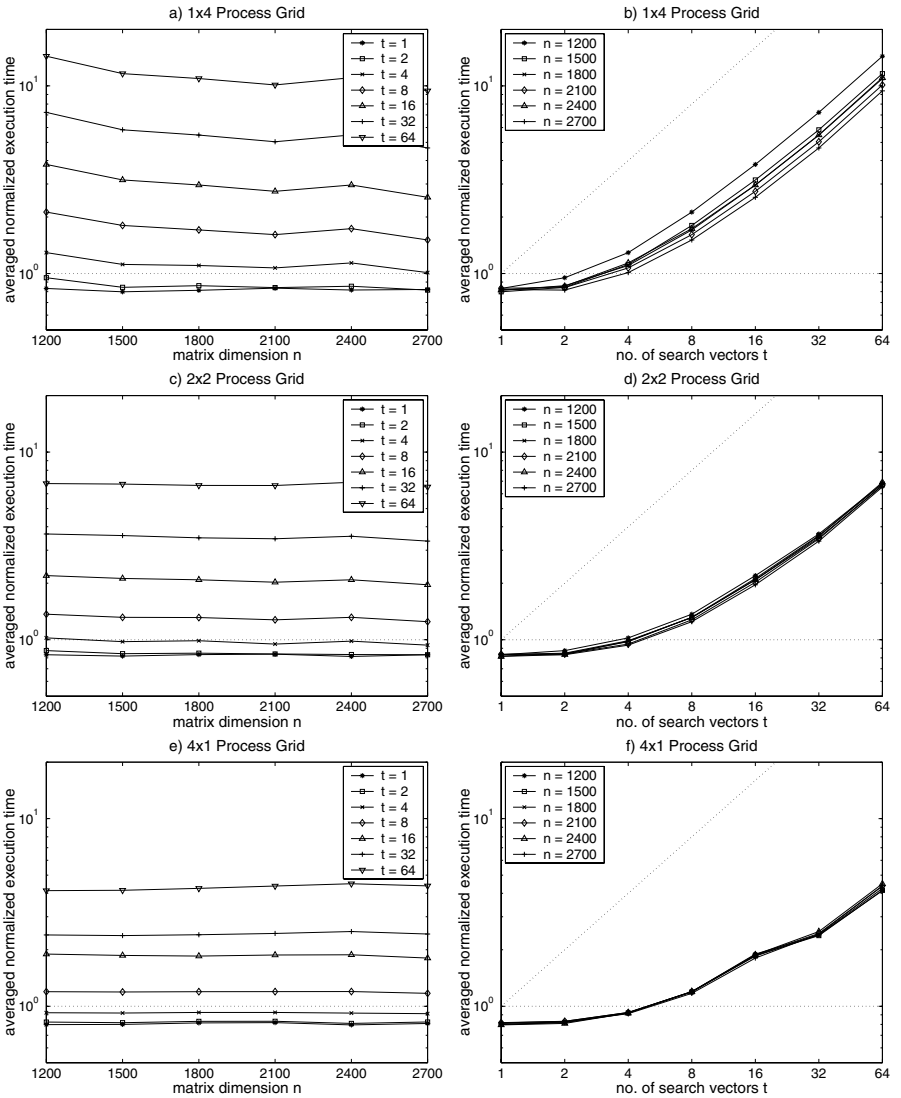

Fig. 2. Averaged execution time $\overline{\mathrm{N}}$ of `PDLACON1`, normalized with respect to `PDLACON`, when experiments were run on 4 processors with 1×4, 2×2 and 4×1 process grids

– As t increases, the time taken for each iteration increases. However, using multiple search vectors ($t > 1$) also accelerates the rate of convergence. The results show that it is possible to obtain better estimates using less time on average compared with `PDLACON`. The cut-off is at $t = 2$ or $t = 4$ in our experiments, in which the matrix-matrix products in Algorithm 1 are evaluated by solving with LU factors.

- As n or t increases, $\overline{\mathrm{C}}$ and $\overline{\mathrm{D}}$ decrease as the matrix-matrix operations start to dominate the overall cost. However, D is consistently larger (5%–23%) than C. The additional cost is largely due to the broadcasting of the two scalar variables from the leading process column to the non-leading process columns at each reverse communication for `PDLACON1`. `PDLACON` suffers in a similar way as the search vector is broadcast at each reverse communication.
- In Figure 2, it is easy to see that `PDLACON1` performs well compared with `PDLACON` and the execution time increases at a much slower rate than t, thanks to the use of level 3 BLAS and PBLAS.

For more processors all the above observations remain true.

5 Concluding Remarks

We have described a parallel Fortran 77 implementation in ScaLAPACK style of the block matrix 1-norm estimator of Higham and Tisseur [7]. Our experiments show that with $t = 2$ or 4 the new code offers better estimates than the existing ScaLAPACK code `PDLACON` with similar execution time. For larger t, estimates exact over 90% of the time are achieved, with execution time growing much slower than t thanks to the parallelism. The new code uses a different programming strategy than `PDLACON` in order to eliminate the cost of broadcasting search vectors over process columns, which is essential to achieve an efficient implementation of the block matrix 1-norm estimator.

Acknowledgements

We thank Françoise Tisseur for supplying an LAPACK-style Fortran implementation of quick sort, and Françoise and Michael Bane, Rupert Ford and Antoine Petitet for their constructive comments on this work.

References

1. E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third edition, 1999. 568
2. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, first edition, 1997. 568
3. W. W. Hager. Conditions estimates. *SIAM J. Sci. Stat. Comput.*, 5:311–316, 1984. 568, 569
4. Nicholas J. Higham. FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation (Algorithm 674). *ACM Trans. Math. Software*, 14(4):381–396, December 1988. 568, 569

5. Nicholas J. Higham. Experience with a matrix norm estimator. *SIAM J. Sci. Stat. Comput.*, 11:804–809, 1990. 569
6. Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996. 568
7. Nicholas J. Higham and Françoise Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. *SIAM J. Matrix Anal. Appl.*, 21(4):1185–1201, 2000. 569, 571, 572, 574, 576
8. *Using MATLAB*. The MathWorks, Inc., Natick, MA, USA. 2000. Online version. 569