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

Sheung Hun Cheng† Nicholas J. Higham‡

August 13, 2001

Abstract

We describe double precision and complex*16 Fortran 77 implementations, in LAPACK style, of a block matrix 1-norm estimator of Higham and Tisseur. This estimator differs from that underlying the existing LAPACK code, `xLACON`, 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 with random matrices on a Sun SPARCStation Ultra-5 show that with $t = 2$ or 4 the new code offers better estimates than `xLACON` with a similar execution time. Moreover, with $t > 2$, estimates exact over 95% and 75% of the time are achieved for the real and complex version respectively, with execution time growing much slower than t. We recommend this new code be included as an auxiliary routine in LAPACK to complement the existing LAPACK routine `xLACON`, upon which the various drivers should still be based for compatibility reasons.

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 [8]. The LAPACK [1] and ScaLAPACK [2] condition numbers and error bounds are based on estimated condition numbers, using the method of Hager [5], which was subsequently improved by Higham [6]. 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 $\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$.

*Numerical Analysis Report No. 393, Manchester Centre for Computational Mathematics, Manchester, England, August 2001, and LAPACK Working Note 152. This work was supported by Engineering and Physical Sciences Research Council grant GR/L94314.

†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/`).

‡Department of Mathematics, University of Manchester, Manchester, M13 9PL, England (`higham@ma.man.ac.uk`, `http://www.ma.man.ac.uk/~higham/`).

In LAPACK and ScaLAPACK Higham's version of Hager's method is implemented in routines `xLACON` and `PxLACON`, respectively. Both routines have a 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" [6]. 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 [6] 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 [6], [7]. Moreover, when the accuracy of the estimates becomes important for certain applications [9], the method does not provide an obvious way to improve the estimate.

Higham and Tisseur [9] present a block generalization of the estimator of [5], [6] 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 LAPACK programming style and report performance on a Sun SPRACStation Ultra-5. The rest of this note is organized as follows. We describe the block 1-norm estimator in Section 2. In Section 3 we present and explain details of our 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 real version of the block 1-norm estimator, which is basically a block power method for the matrix 1-norm. See [9] for a derivation and explanation of the algorithm. We use MATLAB array and indexing notation [10]. We denote by rand$\{-1, 1\}$ a vector with elements from the uniform distribution on the set $\{-1, 1\}$.

Algorithm 2.1 (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 first column the vector of 1s
> and remaining columns rand$\{-1, 1\}$, with a check for and replacement
> of parallel columns.

ind_hist = [ ] % Integer vector recording indices of used unit vectors e_j.
$\text{est}_{\text{old}} = 0$, ind = zeros$(n, 1)$, $S = \text{zeros}(n, t)$
for $k = 1, 2, \ldots$

(1) $Y = AX$
est = $\max\{\, \|Y(:, j)\|_1 : j = 1{:}\, t \,\}$
if est $> \text{est}_{\text{old}}$ or $k = 2$
ind_best = ind_j where est = $\|Y(:, j)\|_1$, $w = Y(:, \text{ind\_best})$
end
if $k \geq 2$ and est $\leq \text{est}_{\text{old}}$, est = est_{old}, goto (5), end
$\text{est}_{\text{old}} = \text{est}$, $S_{\text{old}} = S$
(2) if $k >$ itmax, goto (5), end
$S = \text{sign}(Y)$ % $\text{sign}(x) = 1$ 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) $Z = A^T S$
$h_i = \|Z(i, :)\|_\infty$, $\text{ind}_i = i$, $i = 1{:}\, n$
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 2.1 estimates $\|A^{-1}\|_1$.

MATLAB 6 contains an implementation of Algorithm 2.1 in function `normest1`, which is used by the condition number estimation function `condest`. Moreover, a ScaLAPACK programming style implementation is also available. The implementation details and its performance are reported in [3].

An example, discovered by Dhillon [4], for which the current LAPACK estimator badly underestimates the norm is the inverse of the tridiagonal matrix T with zero diagonal and ones on the sub- and superdiagonals. Algorithm 2.1 does much better for this matrix. Here we use MATLAB (R12.1) to show the result of applying `DLACON` and Algorithm 2.1 with $t = 2$ to estimate the condition number of T. The following M-file invokes Algorithm 2.1 twice in succession (hence with different starting vectors).

```
rand('seed',1)
disp('       Underestimation ratios')
```

```
disp(' n     Alg. 2.1 Alg. 2.1 DLACON')
for n=10:10:100
    A = full(gallery('tridiag',n,1,0,1));
    fprintf('%3.0f    %4.3f    %4.3f    %4.3f\n', n, condest(A)/cond(A,1), ...
             condest(A)/cond(A,1), 1/(cond(A,1)*rcond(A)))
end
```

The output is

```
       Underestimation ratios
 n     Alg. 2.1 Alg. 2.1 DLACON
 10    1.000    1.000    0.200
 20    1.000    1.000    0.100
 30    1.000    1.000    0.067
 40    1.000    1.000    0.050
 50    1.000    1.000    0.040
 60    1.000    0.833    0.033
 70    1.000    1.000    0.029
 80    1.000    1.000    0.025
 90    0.956    0.956    0.022
100    0.880    1.000    0.020
```

3 Implementation Details

We have implemented Algorithm 2.1 in double precision and complex*16 using Fortran 77 in the LAPACK programming style. Our code uses the highest level of BLAS whenever possible.

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 2.1 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 2.1 is straightforwardly translated into Fortran code apart from statement (3), which deserves detailed explanation. Statement (3) is a novel feature of Algorithm 2.1 in which parallel columns within the current sign matrix S and between S and S_{old} are replaced by rand$\{-1, 1\}$. The replacement of parallel columns avoids redundant computation and may lead to a better estimate [9]. 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 2.1, statement (3) is implemented as follows:

```
        iter = 0
        for i = 1:t
            while iter < n/t
(A)             y = S_old^T S(:,i)
                iter = iter + 1
                if ||y||_∞ < n
(B)                 y = S(:,1:i-1)^T S(:,i)
```

Table 1: Storage requirements of old (`xLACON`) and new (`xLACN1`) codes.

Data type	`DLACON`	`DLACN1`	`ZLACON`	`ZLACN1`
double/complex	$2n$	$2tn + 2n + t$	$2n$	$2tn + 2n$
integer	n	$2n$	$-$	$2n$

```
                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 2.1 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.

For the complex*16 implementation, the sign matrix function is defined elementwise by $\mathrm{sign}(a_{ij}) = a_{ij}/|a_{ij}|$ and $\mathrm{sign}(0) = 1$. The benefit of checking for parallel columns is dramatically reduced in the complex case, as explained in [9]. We omit this novel feature in our complex*16 implementation.

The new real and complex versions of our implementations are called `DLACN1` and `ZLACN1` respectively. Their storage requirements grow roughly like $2tn$; see Table 1 for a comparison of storage with `xLACON`.

Two additional routines are used by `xLACN1`. `DLAPST` is a modified version of the LAPACK quicksort routine `DLASRT`; it returns the permutation that achieves the sort as well as the sorted array. The routine `DLARPC` carries out the tests for and replacement of repeated columns in statement (3) of Algorithm 2.1.

4 Numerical Experiments

In this section, our aim is to examine the performance of `DLACN1`and `ZLACN1`. We have addressed the issues of accuracy and reliability of our implementations by reproducing parts of the experimental results in Higham and Tisseur [9], thereby validating our code. We note that there is no test routine for `xLACON` in the LAPACK test suite, but all the drivers that call `xLACON` (such as `xGECON`) do have test routines. In order to test `DLACN1` and `ZLACN1` we modified all LAPACK routines that call `xLACON` to make use of `xLACN1` for $1 \le t \le 4$ and then ran the LAPACK tests. There were no failures, so in this sense both `DLACN1` and `ZLACN1` have passed the LAPACK test suite in the computing environment described below.

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 `DLACN1` and `ZLACN1` on a Sun SPARCStation Ultra-5. The compiler and library details are given in Table 2.

Table 2: Characteristics of the Sun SPARCStation Ultra-5, libraries and compiler options for the experiments.

Compiler	Sun WorkShop Compilers: Version 5.0 (`f77`)
Compiler Flags	`-u -f -dalign -native -xO5 -xarch=v8plusa`
LAPACK	version 3.0
BLAS	ATLAS optimized
ATLAS	version 3.2.1

We use the same compiler flags as were used to compile the LAPACK installation. This provides a basis for measuring and comparing the performance of our implementation with their counterparts in the LAPACK routine, namely `DLACON` and `ZLACON`.

We estimate $\|A^{-1}\|_1$ for $n \times n$ random matrices A with $n = 800$ and 1600. 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 for testing `DLACN1`. For testing `ZLACN1`, random matrices are generated with the following distributions;

- both the real and imaginary parts are each uniform$(0, 1)$,
- both the real and imaginary parts are each uniform $(-1, 1)$,
- both the real and imaginary parts are each normal $(0, 1)$.

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 128 columns, where 128 is the largest value of t to be used, and then ran Algorithm 2.1 for $t = 1, 2, \ldots, 128$ using starting matrix $X_1(:, 1\colon t)$. In this way we could see the effect of increasing t with fixed n.

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 $\alpha = \text{est}/\|A^{-1}\|_1 \leq 1$, 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 nu, where u is the unit roundoff ($u = 2^{-53} \approx 10^{-16}$).
- %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 `xLACON`, to within a relative tolerance nu.
- %T: the percentage of the cases for which our implementation took longer to complete than `xLACON`.

Table 3: Experimental results for `DLACN1` using 500 real random matrices with dimensions $n = 800$ and $n = 1600$.

$n = 800$												
t	$\alpha_{\min}$	$\overline{\alpha}$	%E	%I	%A	%T	$N_{\max}$	$\overline{N}$	$C_{\max}$	$\overline{C}$	$K_{\max}$	$\overline{K}$
1^a	0.20	0.98	84.2	–	–	–	–	–	1.49	0.51	9	5.3
1	0.20	0.98	84.2	–	100.0	27.2	1.29	1.00	1.97	0.82	8	4.3
2	0.74	1.00	93.8	98.6	98.6	88.4	1.86	1.18	2.00	1.08	6	4.1
4	0.92	1.00	97.6	99.0	99.8	97.8	2.16	1.36	2.40	1.55	6	4.0
8	0.98	1.00	99.4	98.0	99.8	98.4	1.59	1.48	4.05	2.76	4	4.0
16	1.00	1.00	100.0	98.6	100.0	100.0	2.50	2.32	5.90	4.00	4	4.0
32	1.00	1.00	100.0	98.6	100.0	100.0	4.57	4.18	10.75	7.47	4	4.0
64	1.00	1.00	100.0	98.6	100.0	100.0	10.04	8.59	24.78	16.58	4	4.0
128	1.00	1.00	100.0	98.2	100.0	100.0	23.79	19.64	43.14	31.42	4	4.0
$n = 1600$												
t	$\alpha_{\min}$	$\overline{\alpha}$	%E	%I	%A	%T	$N_{\max}$	$\overline{N}$	$C_{\max}$	$\overline{C}$	$K_{\max}$	$\overline{K}$
1^a	0.46	0.98	82.8	–	–	–	–	–	1.85	0.23	9	5.5
1	0.46	0.98	82.8	–	100.0	92.0	1.39	1.09	0.88	0.32	8	4.5
2	0.63	0.99	89.8	98.2	98.2	91.0	2.00	1.26	1.74	0.44	8	4.1
4	0.76	1.00	96.8	98.4	99.6	96.6	2.15	1.33	2.58	0.72	6	4.0
8	0.94	1.00	98.8	93.6	100.0	99.8	1.67	1.53	2.23	1.48	4	4.0
16	0.97	1.00	99.6	94.0	100.0	100.0	2.60	2.38	3.91	2.78	4	4.0
32	1.00	1.00	100.0	94.4	100.0	100.0	4.64	4.21	7.54	5.37	4	4.0
64	1.00	1.00	100.0	94.4	100.0	100.0	9.16	8.19	15.20	10.49	4	4.0
128	1.00	1.00	100.0	91.8	100.0	100.0	18.97	16.55	26.25	19.17	4	4.0

[a]Data for `DLACON`

- N: The execution time for `xLACN1` normalized against the time taken by `xLACON`.
- C: the percentage of time spent in `xLACN1` for a given A.
- K: the number of matrix-matrix operations for a given A.

In Tables 3 and 4 we show detailed statistical results and 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% and 70% of the time can be computed for the real and complex cases respectively, 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 [9].
- As t increases, the time taken for each iteration increases. However, using multiple search vectors ($t > 1$) accelerates the rate of convergence and also allows the use of level 3 BLAS, so execution time grows much more slowly than t. In this computing environment $t = 2$ or $t = 4$ produces distinctly better estimates than those from `xLACON` with only a modest increase in execution time.

Table 4: Experimental results for `ZLACN1` using 500 complex random matrices with dimensions $n = 800$ and $n = 1600$.

$n = 800$												
t	$\alpha_{\min}$	$\overline{\alpha}$	%E	%I	%A	%T	$N_{\max}$	$\overline{N}$	$C_{\max}$	$\overline{C}$	$K_{\max}$	$\overline{K}$
1^a	0.25	0.85	28.4	–	–	–	–	–	1.67	0.85	11	6.4
1	0.25	0.94	49.2	–	100.0	100.0	1.61	1.19	1.86	0.81	11	5.4
2	0.68	0.97	62.4	90.2	98.8	94.2	2.97	1.41	2.15	1.58	11	5.1
4	0.56	0.98	78.2	87.4	99.8	91.0	2.59	1.39	3.85	3.06	9	4.9
8	0.78	0.99	89.6	90.2	100.0	100.0	4.03	1.90	5.06	4.35	9	4.7
16	0.78	1.00	96.4	93.6	100.0	100.0	5.33	2.98	6.23	5.51	9	4.6
32	0.78	1.00	98.6	94.2	100.0	100.0	7.53	5.60	6.48	5.93	7	4.5
64	0.94	1.00	99.8	95.0	100.0	100.0	15.15	11.23	6.63	6.15	7	4.5
128	1.00	1.00	100.0	95.2	100.0	100.0	26.06	19.12	7.75	7.27	5	4.5
$n = 1600$												
t	$\alpha_{\min}$	$\overline{\alpha}$	%E	%I	%A	%T	$N_{\max}$	$\overline{N}$	$C_{\max}$	$\overline{C}$	$K_{\max}$	$\overline{K}$
1^a	0.26	0.86	26.2	–	–	–	–	–	1.55	0.39	11	6.4
1	0.51	0.93	44.6	–	100.0	100.0	2.04	1.47	1.13	0.31	11	5.4
2	0.66	0.97	57.2	80.4	99.0	99.8	3.68	1.68	1.52	0.63	11	5.1
4	0.67	0.98	75.8	80.8	99.6	99.2	3.14	1.69	2.34	1.22	9	5.0
8	0.81	0.99	88.2	84.4	100.0	100.0	5.10	2.21	2.74	1.80	9	4.8
16	0.86	1.00	95.4	87.4	100.0	100.0	5.88	3.24	3.01	2.45	7	4.7
32	0.88	1.00	98.8	89.2	100.0	100.0	8.15	5.47	3.38	2.89	7	4.6
64	0.92	1.00	99.8	90.0	100.0	100.0	13.60	10.26	3.58	3.22	7	4.6
128	0.94	1.00	99.8	90.0	100.0	100.0	24.14	18.13	4.01	3.67	5	4.6

[a]Data for `ZLACON`

- As n increases, $\overline{C}$ decreases as the matrix-matrix operations start to dominate the overall cost.

5 Concluding Remarks

We have described a Fortran 77 implementation in LAPACK style of the block matrix 1-norm estimator of Higham and Tisseur [9]. Our experiments show that with $t = 2$ or 4 the new code offers better estimates than the existing LAPACK code `xLACON` with similar execution time. For our random test matrices, with $t = 4$ estimates exact over 95% and 75% of the time are achieved in the real and complex cases, respectively, with execution time growing much slower than t thanks to the use of level 3 BLAS and the accelerated convergence.

We propose that ***xLACN1*** *be included as an auxiliary routine in LAPACK.* We do not suggest replacing `xLACON` with `xLACN1` (with $t = 2$, say)[1] because this would bring a change in calling sequence and an increase in workspace requirements and hence would

[1]We note that the `xGyCON` drivers call special versions of level 2 BLAS triangular solvers that scale to prevent overflow; if these drivers are modified to call `xLACN1` then special level 3 BLAS triangular solvers with scaling would need to be written.

require users to modify existing codes that call LAPACK drivers or call the estimator directly[2]. Moreover, since `xLACON` is deterministic while `xLACN1` with $t > 1$ can produce different estimates on different invocations (because of the random starting vectors) such a change could confuse users. But since `xLACN1` is the state of the art norm estimator and is of value to knowledgeable users who need better and tuneable norm estimates we believe it is worth including in LAPACK as an auxiliary routine.

Acknowledgements

We thank Françoise Tisseur for the quicksort subroutine `DLASRT`.

[2]Replacing `xLACON` by `xLACN1` with $t = 1$ is not an option, since the latter is less reliable, due to the former's rather ad hoc but very effective "extra vector".

A Appendix

In this appendix we list the double precision codes DLACN1 (main routine) and DLARPC (tests for and replaces repeated columns).

All the codes are available from http://www.cs.man.ac.uk/~scheng/PCMF/

```
      SUBROUTINE DLACN1( N, T, V, X, LDX, XOLD, LDXOLD, WRK,
     $                   H, IND, INDH, EST, KASE, ISEED, INFO )
*
*     .. Scalar Arguments ..
      INTEGER            INFO, KASE, LDXOLD, LDX, N, T
      DOUBLE PRECISION   EST
*     ..
*     .. Array Arguments ..
      INTEGER            IND( * ), INDH( * ), ISEED( 4 )
      DOUBLE PRECISION   H( * ),  V( * ), X( LDX, * ), WRK( * ),
     $                   XOLD( LDXOLD, * )
*     ..
*
*  Purpose
*  =======
*
*  DLACN1 estimates the 1-norm of a square, real matrix A.
*  Reverse communication is used for evaluating matrix-matrix products.
*
*  Arguments
*  =========
*
*  N      (input) INTEGER
*         The order of the matrix.  N >= 1.
*
*  T      (input) INTEGER
*         The number of columns used at each step.
*
*  V      (output) DOUBLE PRECISION array, dimension (N).
*         On the final return, V = A*W,  where  EST = norm(V)/norm(W)
*         (W is not returned).
*
*  X      (input/output) DOUBLE PRECISION array, dimension (N,T)
*         On an intermediate return, X should be overwritten by
*               A * X,   if KASE=1,
*               A' * X,  if KASE=2,
*         and DLACN1 must be re-called with all the other parameters
*         unchanged.
*
*  LDX    (input) INTEGER
*         The leading dimension of X.  LDX >= max(1,N).
*
*  XOLD   (workspace) DOUBLE PRECISION array, dimension (N,T)
*
```

```
*  LDXOLD (input) INTEGER
*         The leading dimension of XOLD.  LDXOLD >= max(1,N).
*
*  WRK    (workspace) DOUBLE PRECISION array, dimension (T)
*
*  H      (workspace) DOUBLE PRECISION array, dimension (N)
*
*  IND    (workspace) INTEGER array, dimension (N)
*
*  INDH   (workspace) INTEGER array, dimension (N)
*
*  EST    (output) DOUBLE PRECISION
*         An estimate (a lower bound) for norm(A).
*
*  KASE   (input/output) INTEGER
*         On the initial call to DLACN1, KASE should be 0.
*         On an intermediate return, KASE will be 1 or 2, indicating
*         whether X should be overwritten by A * X  or A' * X.
*         On the final return from DLACN1, KASE will again be 0.
*
*  ISEED  (input/output) INTEGER array, dimension (4)
*          On entry, the seed of the random number generator; the array
*          elements must be between 0 and 4095, and ISEED(4) must be
*          odd.
*          On exit, the seed is updated.
*
*  INFO   (output) INTEGER
*         INFO describes how the iteration terminated:
*            INFO = 1: iteration limit reached.
*            INFO = 2: estimate not increased.
*            INFO = 3: repeated sign matrix.
*            INFO = 4: power method convergence test.
*            INFO = 5: repeated unit vectors.
*
*  =====================================================================
*
*     .. Parameters ..
      INTEGER            ITMAX
      PARAMETER          ( ITMAX = 5 )
      DOUBLE PRECISION   ZERO, ONE, TWO
      PARAMETER          ( ZERO = 0.0D+0, ONE = 1.0D+0, TWO = 2.0D+0 )
*     ..
*     .. Local Scalars ..
      INTEGER            I, IBEST, ITEMP, ITER, J, JUMP
      DOUBLE PRECISION   ESTOLD, TEMP
*     ..
*     .. External Functions ..
      INTEGER            IDAMAX
      DOUBLE PRECISION   DASUM, DDOT
      EXTERNAL           DASUM, DDOT, IDAMAX
```

```
*     ..
*     .. External Subroutines ..
      EXTERNAL           DCOPY, DLACPY, DLAPST, DLARNV, DLARPC, DLASCL
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          ABS, DBLE, NINT, SIGN
*     ..
*     .. Save statement ..
      SAVE
*     ..
*     .. Executable Statements ..
*
*
      IF( KASE .EQ. 0 ) THEN
*
         ESTOLD = ZERO
         ITER = 1
         ITEMP = 1
         INFO = 0
*
         DO 10 I = 1, N
            X( I, 1 ) = ONE
            IND( I ) = I
            INDH( I ) = 0
   10    CONTINUE
*
         DO 30 J = 2, T
            CALL DLARNV( 2, ISEED, N, X( 1, J ) )
            DO 20 I = 1, N
               X( I, J ) = SIGN( ONE, X( I, J ) )
   20       CONTINUE
   30    CONTINUE
*
         IF ( T .GT. 1 )
     $      CALL DLARPC( N, T, X, LDX, XOLD, LDXOLD, WRK, KASE, ISEED)
*
         CALL DLASCL( 'G', 0, 0, DBLE(N), ONE, N, T, X, LDX, INFO )
*
         KASE = 1
         JUMP = 1
         RETURN
      END IF
*
      GO TO ( 40, 100 ) JUMP
*
*     ................ ENTRY   (JUMP = 1)
*     FIRST HALF OF THE ITERATION: X HAS BEEN OVERWRITTEN BY A*X.
*
   40 CONTINUE
*
```

```
      IF ( ITER .EQ. 1  .AND.  N .EQ. 1 ) THEN
         V( 1 ) = X( 1, 1 )
         EST = ABS( V( 1 ) )
*        ... QUIT
         GO TO 210
      END IF

      EST = ZERO
      DO 50 J = 1, T
         TEMP = DASUM( N, X( 1, J ), 1 )
         IF ( TEMP .GT. EST ) THEN
             EST = TEMP
             ITEMP = J
         END IF
   50 CONTINUE
*
      IF ( EST .GT. ESTOLD  .OR.  ITER .EQ. 2 ) THEN
         IBEST = IND( ITEMP )
      END IF
*
      IF ( EST .LE. ESTOLD  .AND.  ITER .GE. 2 ) THEN
         EST = ESTOLD
         INFO = 2
         GO TO 210
      END IF
*
      ESTOLD = EST
      CALL DCOPY( N, X( 1, ITEMP ), 1, V, 1 )
*
      IF ( ITER .GT. ITMAX ) THEN
         INFO = 1
         GO TO 210
      END IF
*
      DO 70 J = 1, T
         DO 60 I = 1, N
            X( I, J ) = SIGN( ONE, X( I, J ) )
   60    CONTINUE
   70 CONTINUE
*
      IF ( ITER .GT. 1 ) THEN
*
*        IF ALL COLUMNS of X PARALLEL TO XOLD, EXIT.
*
         DO 80 J = 1, T
            CALL DGEMV( 'Transpose', N, T, ONE, XOLD, LDXOLD,
     $           X( 1, J ), 1, ZERO, WRK, 1)
            IF ( NINT(ABS(WRK(IDAMAX(T, WRK, 1)))) .LT. N ) GO TO 90
   80    CONTINUE
         INFO = 3
```

```
         GO TO 210
*
   90    CONTINUE
*
         IF ( T. GT. 1 )
     $      CALL DLARPC( N, T, X, LDX, XOLD, LDXOLD, WRK, KASE, ISEED)
*
      ELSE
*
         IF ( T. GT. 1 )
     $      CALL DLARPC( N, T, X, LDX, XOLD, LDXOLD, WRK, 0, ISEED)
*
      END IF
*
      CALL DLACPY( 'Whole', N, T, X, LDX, XOLD, LDXOLD )
*
      KASE = 2
      JUMP = 2
      RETURN
*
*     ................ ENTRY   (JUMP = 2)
*     SECOND HALF OF THE ITERATION:
*                      X HAS BEEN OVERWRITTEN BY TRANSPOSE(A)*X.
*
  100 CONTINUE
*
      DO 110 I = 1, N
         H( I ) = ABS( X ( I, IDAMAX( T, X( I, 1 ), N ) ) )
         IND( I ) = I
  110 CONTINUE
*
      IF ( ITER .GE. 2 .AND. H( IDAMAX(N, H, 1) ) .EQ. H(IBEST) ) THEN
         INFO = 4
         GO TO 210
      END IF
*
*     Sort so that h(i) >= h(j) for i < j
*
      CALL DLAPST( 'D', N, H, IND, ITEMP )
*
      IF ( ITER .EQ. 1 ) THEN
         ITEMP = T
         GO TO 170
      END IF
*
*     IF IND(1:T) IS CONTAINED IN INDH, TERMINATE.
*
      IF ( T .GT. 1 ) THEN
         DO 130 J = 1, T
            DO 120 I = 1, (ITER-1)*T
```

```
               IF (I .GT. N .OR. IND( J ) .EQ. INDH( I )) GO TO 130
  120       CONTINUE
            GO TO 140
  130    CONTINUE
         INFO = 5
         GO TO 210
  140    CONTINUE
*
*        REPLACE IND(1:T) BY THE FIRST T INDICES IN IND THAT
*        ARE NOT IN INDH.
*
         ITEMP = 1
         DO 160 J = 1, N
            DO 150 I = 1, (ITER-1)*T
               IF ( I .GT. N .OR. IND( J ) .EQ. INDH( I ) ) GO TO 160
  150       CONTINUE
            IND( ITEMP ) = IND( J )
            IF ( ITEMP .EQ. T ) GO TO 170
            ITEMP = ITEMP + 1
  160    CONTINUE
      END IF
*
      ITEMP = ITEMP - 1
*
  170 CONTINUE
*
      IF ( (ITER-1)*T .GE. N ) THEN
         DO 180 J = 1, ITEMP
            INDH( (ITER-1)*T+J ) = IND( J )
  180    CONTINUE
      END IF
*
      DO 200 J = 1, T
         DO 190 I = 1, N
            X( I, J ) = ZERO
  190    CONTINUE
         X( IND( J ), J ) = ONE
  200 CONTINUE
*
      ITER = ITER + 1
*
      KASE = 1
      JUMP = 1
      RETURN
*
  210 CONTINUE
      KASE = 0
      RETURN
*
*     End of DLACN1
```

```
*
      END
```

```
      SUBROUTINE DLARPC( N, T, X, LDX, XOLD, LDXOLD, WRK, KASE, ISEED)
*
*     .. Scalar Arguments ..
      INTEGER            N, T, LDX, LDXOLD, KASE, ISEED(4)
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   WRK( * ), X( LDX, * ), XOLD( LDXOLD, * )
*     ..
*
*  Purpose
*  =======
*
*  DLARPC looks for and replaces columns of X which are parallel to
*         columns of XOLD and itself.
*
*  Arguments
*  =========
*
*  N      (input) INTEGER
*         The number of rows.  N >= 1.
*
*  T      (input) INTEGER
*         The number of columns used at each step.
*
*  X      (input/output) DOUBLE PRECISION array, dimension (N,T)
*         On return, X will have full rank.
*
*  LDX    (input) INTEGER
*         The leading dimension of X.  LDX >= max(1,N).
*
*  XOLD   (input/output) DOUBLE PRECISION array, dimension (N,T)
*         On return, XOLD will have full rank.
*
*  LDXOLD (input) INTEGER
*         The leading dimension of XOLD.  LDXOLD >= max(1,N).
*
*  WRK    (workspace) DOUBLE PRECISION array, dimension (T)
*
*  KASE   (input) INTEGER
*          Check parallel columns within X only when KASE = 0,
*          check both X and XOLD otherwise.
*
*  ISEED  (input/output) INTEGER array, dimension (4)
*          On entry, the seed of the random number generator; the array
*          elements must be between 0 and 4095, and ISEED(4) must be
```

```
*          odd.
*          On exit, the seed is updated.
*
*     .. Parameters ..
      DOUBLE PRECISION   ZERO, ONE
      PARAMETER          ( ZERO = 0.0D+0, ONE = 1.0D+0 )
*     ..
*     .. Local Scalars ..
      INTEGER            I, J, JSTART, PCOL
*     ..
*     .. External Functions ..
      INTEGER            IDAMAX
      EXTERNAL           IDAMAX
*     ..
*     .. External Subroutines ..
      EXTERNAL           DGEMV, DLARNV
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          ABS, NINT, SIGN
*     ..
*     .. Executable Statements ..
*
      IF ( KASE .EQ. 0 ) THEN
         JSTART = 2
      ELSE
         JSTART = 1
      END IF
*
      DO 50 J = JSTART, T
*
         PCOL = 0
*
         IF ( KASE .EQ. 0 ) GO TO 30
   10    CALL DGEMV( 'Transpose', N, T, ONE, XOLD, LDXOLD,
     $                   X( 1, J ), 1, ZERO, WRK, 1)
         IF ( NINT ( ABS ( WRK ( IDAMAX ( T, WRK, 1 ) ) ) )
     $        .EQ. N ) THEN
            PCOL = PCOL + 1
            CALL DLARNV( 2, ISEED, N, X( 1, J ) )
            DO 20 I = 1, N
               X( I, J ) = SIGN( ONE, X( I, J ) )
   20       CONTINUE
            IF ( PCOL .GE. N/T ) GO TO 60
            GO TO 10
         END IF
*
         IF ( J .EQ. 1 ) GO TO 50
   30    CALL DGEMV( 'Transpose', N, J-1, ONE, X, LDX,
     $                   X( 1, J ), 1, ZERO, WRK, 1)
         IF ( NINT ( ABS ( WRK ( IDAMAX ( J-1, WRK, 1 ) ) ) )
```

```
     $        .EQ. N ) THEN
              PCOL = PCOL + 1
              CALL DLARNV( 2, ISEED, N, X( 1, J ) )
              DO 40 I = 1, N
                 X( I, J ) = SIGN( ONE, X( I, J ) )
   40         CONTINUE
              IF ( PCOL .GE. N/T ) GO TO 60
              IF ( KASE .EQ. 0 ) THEN
                 GO TO 30
              ELSE
                 GO TO 10
              END IF
           END IF
*
   50 CONTINUE
   60 CONTINUE
      RETURN
*
*     End of DLARPC
*
      END
```

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.

[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.

[3] Sheung Hun Cheng and Nicholas J. Higham. Parallel implementation of a block algorithm for matrix 1-norm estimation. Numerical Analysis Report No. 374, Manchester Centre for Computational Mathematics, Manchester, England, February 2001. To appear in Proceedings of EuroPar 2001, Manchester.

[4] Inderjit Dhillon. Reliable computation of the condition number of a tridiagonal matrix in $O(n)$ time. *SIAM J. Matrix Anal. Appl.*, 19(3):776–796, 1998.

[5] W. W. Hager. Conditions estimates. *SIAM J. Sci. Stat. Comput.*, 5:311–316, 1984.

[6] 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.

[7] Nicholas J. Higham. Experience with a matrix norm estimator. *SIAM J. Sci. Stat. Comput.*, 11:804–809, 1990.

[8] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[9] 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.

[10] The MathWork, Inc., Natick, MA, USA. *Using MATLAB*, 2000. Online version.