Chapter F07

Linear Equations (ScaLAPACK)

Contents

1 Scope of the Chapter

This chapter provides routines for the solution of real and complex systems of simultaneous linear equations, and associated computations. It provides routines for:

matrix factorizations

solution of linear equations

estimating triangular matrix condition numbers

The routines in this chapter (F07), which have been mostly derived from the ScaLAPACK project (see Choi *et al.* [3], Anderson *et al.* [1]), can handle both **real** and **complex dense** matrices. They have been designed to be efficient on a wide range of parallel computers. General sparse matrices are handled by the routines in Chapter F11.

2 Background to the Problems

This section is only a brief introduction to the numerical solution of systems of linear equations. Consult a standard textbook, for example Golub and Van Loan [5], for a more thorough discussion.

2.1 Notation

We use the standard notation for a system of simultaneous linear equations:

$$Ax = b \tag{1}$$

where A is the **coefficient matrix**, b is the **right-hand side**, and x is the **solution**. A is assumed to be a square matrix of order n. If there are several right-hand sides, we write

$$AX = B \tag{2}$$

where the columns of B are the individual right-hand sides, and the columns of X are the corresponding solutions.

We also use the following notation, both here and in the routine documents:

$\hat{x}$ a **computed** solution to $Ax = b$, which usually differs from the exact solution x because of round-off error

$r = b - A\hat{x}$ the **residual** corresponding to the computed solution $\hat{x}$

$\|x\|_\infty = \max_i |x_i|$ the infinity-norm of the vector x

$\|A\|_\infty = \max_i \sum_j |a_{ij}|$ the infinity-norm of the matrix A

$\|x\|_1 = \sum_i |x_i|$

$\|A\|_1 = \max_j \sum_i |a_{ij}|$

2.2 Matrix Factorizations

If A is upper or lower triangular, $Ax = b$ can be solved by a straightforward process of backward or forward substitution. Otherwise, the solution is obtained after first factorizing A, as follows.

General matrices (*LU* factorization with partial pivoting):

$$A = PLU$$

where P is a permutation matrix, L is lower triangular with diagonal elements equal to 1, and U is upper triangular; the permutation matrix P (which represents row interchanges) is needed to ensure numerical stability.

Symmetric or Hermitian positive-definite matrices (Cholesky factorization):

$$A = U^T U \text{ or } A = LL^T \quad (A = U^H U \text{ or } A = LL^H \text{ in the complex Hermitian case})$$

where U is upper triangular and L is lower triangular.

2.3 Solution of Systems of Equations

Given one of the above matrix factorizations, it is straightforward to compute a solution to $Ax = b$ by solving two subproblems, as shown below, first for y and then for x. Each subproblem consists essentially of solving a triangular system of equations by forward or backward substitution; the permutation matrix P introduces only a little extra complication.

General matrices (*LU* factorization with partial pivoting):

$$\begin{aligned} Ly &= P^T b \\ Ux &= y \end{aligned}$$

Symmetric or Hermitian positive-definite matrices (Cholesky factorization):

$$\begin{aligned} U^T y &= b \\ Ux &= y \end{aligned} \quad \text{or} \quad \begin{aligned} Ly &= b \\ L^T x &= y \end{aligned}$$

($U^T y = b$ and $L^T x = y$ become $U^H y = b$ and $L^H x = y$ in the complex Hermitian case).

2.4 Sensitivity and Error Analysis

2.4.1 Normwise error bounds

Frequently in practical problems, the data A and b are not known exactly, and it is then important to understand how uncertainties or perturbations in the data can affect the solution. If x is the exact solution to $Ax = b$, and $x + \delta x$ is the exact solution to a perturbed problem $(A + \delta A)(x + \delta x) = (b + \delta b)$, then

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right) + \cdots (\text{2nd-order terms})$$

where $\kappa(A)$ is the **condition number** of A defined by:

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|. \tag{3}$$

In other words, relative errors in A or b may be amplified in x by a factor $\kappa(A)$. Section 2.4.2 discusses how to compute or estimate $\kappa(A)$.

Similar considerations apply when we study the effects of **rounding errors** introduced by computation in finite precision. The effects of rounding errors can be shown to be equivalent to perturbations in the original data, such that $\|\delta A\|/\|A\|$ and $\|\delta b\|/\|b\|$ are usually at most $p(n)\epsilon$, where ϵ is the ***machine precision*** and $p(n)$ is an increasing function of n which is seldom larger than $10n$ (although in theory it can be as large as 2^{n-1}). In other words, the computed solution $\hat{x}$ is the exact solution of a linear system $(A + \delta A)\hat{x} = b + \delta b$ which is close to the original system in a normwise sense.

2.4.2 Estimating condition numbers

The previous section has emphasized the usefulness of the quantity $\kappa(A)$ in understanding the sensitivity of the solution of $Ax = b$. To compute the value of $\kappa(A)$ from equation (3) is more expensive than solving $Ax = b$ in the first place. Hence it is standard practice to **estimate** $\kappa(A)$, in either the 1-norm or the ∞-norm, by a method which only requires $O(n^2)$ additional operations, assuming that a suitable factorization of A is available. The method used in this chapter is Higham's modification of Hager's method (Higham [6]). It yields an estimate which is never larger than the true value, but which seldom falls short by more than a factor of 3 (although artificial examples can be constructed where it is much smaller). This is acceptable since it is the order of magnitude of $\kappa(A)$ which is important rather than its precise value. Because $\kappa(A)$ is infinite if A is singular, the routines in this chapter actually return the **reciprocal** of $\kappa(A)$.

2.5 Block Algorithms

The routines in this chapter use what is termed a **block-partitioned algorithm**. This means that at each major step of the algorithm a **block** of rows or columns is updated, and most of the computation is performed by matrix–matrix operations on these blocks. Blocks are distributed among the participating processors in a **cyclic 2-d block** fashion (see Section 2.6). The matrix–matrix operations are performed

by calls to the Level 3 PBLAS (Parallel BLAS) (see Choi *et al.* [3]), which rely on the communication primitives provided by the Basic Linear Algebra Communication Subprograms or BLACS (Dongarra *et al.* [4]). See Golub and Van Loan [5] or Anderson *et al.* [1] for more information about block-partitioned algorithms. The performance of a block-partitioned algorithm varies to some extent with the block size — that is, the number of rows or columns per block. In general, this is a machine-dependent parameter.

2.6 Data Distribution

The routines in this chapter use **cyclic 2-d block distribution** for all matrices and vectors, in order to try to minimise data movement. This distribution is such that row blocks of the matrix are distributed in wrapped fashion to the associated row of the logical grid of processors and, likewise, column blocks are distributed in wrapped fashion to the associated column of the logical grid of processors. Here the terms row block and column block refer to one or more contiguous rows or columns of a matrix which are treated as a single entity from the algorithmic point of view. For those familiar with High Performance Fortran (HPF) terminology this is equivalent to !HPF$ DISTRIBUTE CYCLIC (MB), CYCLIC (NB) where MB and NB are the row and column blocking factors of the matrix distribution. See [8].

	1	2	3	4	5	6	7	8	9	10	11	12
1	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
2	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
3	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
4	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
5	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
6	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
7	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
8	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
9	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
10	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
11	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
12	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}

Figure 1
Block distribution over a 2 by 3 logical grid of processors.

Figure 1 shows how blocks of a matrix are distributed over a 2 by 3 logical grid of processors: the matrix has 12 column and 12 row blocks; the column and row indices in Figure 1 indicate row and column blocks, respectively. Each box contains the index of the processor storing that particular block. The shading is provided only as a visual aid to highlight the processor template and has no other meaning.
Figure 2 shows the same distribution from the processors' point of view. Each of the larger boxes in Figure 2 is labelled by the processor number which stores the blocks that box contains. The indices of the rows and columns in Figure 2 denote the indices of the row and column blocks.

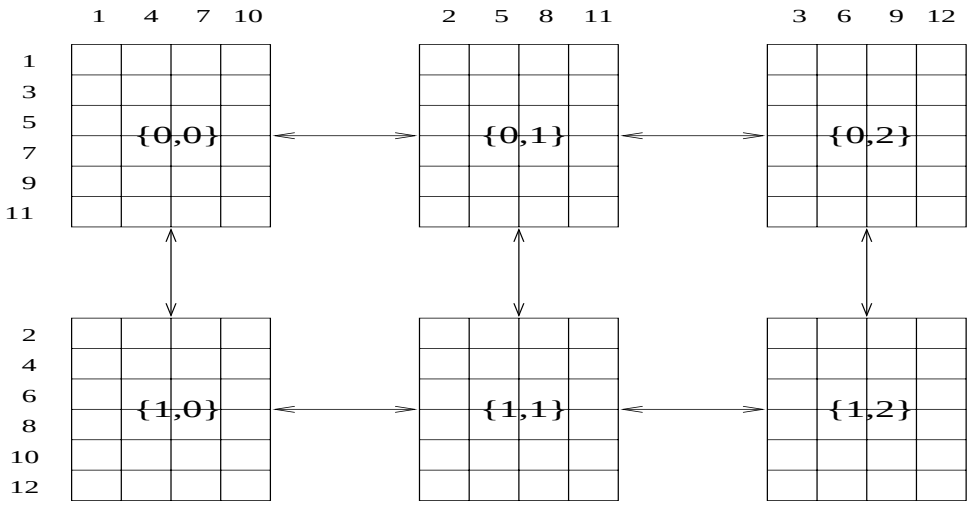

Figure 2
Data distribution from the processors' point of view.

2.7 References

[1] Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J J, Du Croz J J, Greenbaum A, Hammarling S, McKenney A, Ostrouchov S and Sorensen D (1995) *LAPACK Users' Guide* (2nd Edition) SIAM, Philadelphia

[2] Choi J, Demmel J, Dhillon I, Dongarra J J, Ostrouchov S, Petitet A, Stanley K, Walker D W and Whaley R C (1995) ScaLAPACK: a portable linear algebra library for distributed memory computers–design issues and performance *LAPACK Working Note No. 95 (Technical Report CS-95-283)* Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA. Published as: Dongarra J J, Masden K and Waśniewski J (Ed), (Proceedings of the Second International Workshop, PARA '95, Lyngby, Denmark) *Applied Parallel Computing* Springer-Verlag, Berlin, Germany 95–106.
URL: http://www.netlib.org/lapack/lawns/lawn95.ps

[3] Choi J, Dongarra J J, Ostrouchov S, Petitet A, Walker D W and Whaley R C (1995) A proposal for a set of parallel basic linear algebra subprograms *LAPACK Working Note No. 100 (Technical Report CS-95-292)* Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA. Published as: Dongarra J J, Masden K and Waśniewski J (Ed), (Proceedings of the Second International Workshop, PARA '95, Lyngby, Denmark) *Applied Parallel Computing* Springer-Verlag, Berlin, Germany 107–114.
URL: http://www.netlib.org/lapack/lawns/lawn100.ps

[4] Dongarra J J and Whaley R C (1995) A users' guide to the BLACS v1.0. *LAPACK Working Note 94 (Technical Report CS-95-281)* Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA.
URL: http://www.netlib.org/lapack/lawns/lawn94.ps

[5] Golub G H and Van Loan C F (1989) *Matrix Computations* Johns Hopkins University Press (2nd Edition), Baltimore

[6] Higham N J (1988) FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation *ACM Trans. Math. Software* **14** 381–396

[7] (1996) *ScaLAPACK Users Guide* (Draft), To be published. Department of Computer Science, University of Tennessee, 104 Ayres Hall, Knoxville, TN 37996-1301, USA.
URL: http://www.netlib.org/scalapack/scalapack_ug.ps

[8] Koelbel C H, Loveman D B, Schreiber R S, Steele Jr. G L, and Zosel M E (1994) *The High Performance Fortran Handbook* The MIT Press, Cambridge, MA, USA

3 Recommendations on Choice and Use of Available Routines

Note: Refer to the Users' Note for your implementation to check that a routine is available.

3.1 Available Routines

The table in Section 3.4 shows the routines which are provided for performing different computations on different types of matrices. Each entry in the table gives the NAG name and, when applicable, the ScaLAPACK double precision name.

Routines are provided for the following types of matrix:

general

symmetric and Hermitian positive-definite

triangular

For each of the above types of matrix (except where indicated), routines are provided to perform the following computations:

(a) (except for triangular matrices) factorize the matrix (see Section 2.2);

(b) (except for triangular matrices) solve a system of linear equations, using the factorization (see Section 2.3);

(c) (only for triangular matrices) estimate the condition number of the matrix.

Thus, to solve a particular problem, it is usually necessary to call two or more routines in succession. This is illustrated in the example programs in the routine documents.

3.2 NAG Names and ScaLAPACK Names

As well as the NAG routine name (beginning F07-), the table in Section 3.4 shows the ScaLAPACK routine names in double precision. The routines may be called either by their NAG names or by their ScaLAPACK names. The double precision form of the ScaLAPACK name can be used (beginning with PD- or PZ-). References to F07 routines in the Manual normally include the ScaLAPACK double precision names, for example, F07ADFP (PDGETRF).

The ScaLAPACK routine names follow a simple scheme. Each name has the structure **PXYYZZZ**, where the components have the following meanings:

X the second letter indicates the data type (real or complex) and precision

- **D** real, double precision (in Fortran, `DOUBLE PRECISION`)
- **Z** complex, double precision (in Fortran, `COMPLEX*16`)
  (for real and complex, single precision, S and C respectively)

YY the third and fourth letters indicate the type of the matrix A

- **GE** general
- **PO** symmetric or Hermitian positive-definite
- **TR** triangular

ZZZ the last three letters indicate the computation performed

- **TRF** triangular factorization
- **TRS** solution of linear equations, using the factorization
- **CON** estimate condition number

Thus PDGETRF performs a triangular factorization of a real general matrix in double precision.

3.3 Parameters Conventions

3.3.1 Option parameters

Most routines in this chapter have one or more option parameters, of type CHARACTER. The descriptions in Section 4 of the routine documents refer only to upper-case values (for example 'U' or 'L'); however in every case, the corresponding lower-case characters may be supplied (with the same meaning). Any other value is illegal.

A longer character string can be passed as the actual parameter, making the calling program more readable, but only the first character is significant. (This is a feature of Fortran 77.) For example:

```
CALL F07AEFP ('transpose', . . . )
```

3.3.2 Problem dimensions (null matrices)

It is permissible for the problem dimensions (N or NRHS) to be passed as zero, in which case the computation is skipped. Negative dimensions are regarded as an error.

3.3.3 Matrix Data

In all routines in this chapter the local elements of a matrix are stored in a 1-d array. For example, the local elements of the $m_A \times n_A$ matrix A can be stored in the real array A($*$). However, it is more convenient to consider A as a 2-d array of dimension (LDA,$*$), where LDA must be greater than or equal to the number of rows of A stored in the specific row of the processor grid and the array A must have a number of columns greater than or equal to the number of columns of A stored in the specific column of the processor grid. Further information about the distribution of the matrix over the participating processors are encapsulated in an integer array called an **array descriptor**. Such a descriptor is associated with each distributed matrix. The entries of the descriptor uniquely determine the mapping of the matrix entries onto local processors' memories. Moreover, with the exception of the local leading dimension and the associated BLACS context, the descriptor array elements are global characterising the distributed matrix. As an example, in an LU factorization and the associated solver routines, a descriptor array of dimension (9), say IDESCA, would store the following information:

IDESCA(1) the descriptor type: for cyclic 2-d block distribution this must be set to 1;

IDESCA(2) the BLACS context (ICNTXT) for the processor grid;

IDESCA(3) m_A, the number of rows of A;

IDESCA(4) n_A, the number of columns of A;

IDESCA(5) M_b, the blocking factor used to distribute the rows of A, i.e., the number of rows stored in a block;

IDESCA(6) N_b, the blocking factor used to distribute the columns of A, i.e., the number of columns stored in a block;

IDESCA(7) the processor row index over which the first row of A is distributed;

IDESCA(8) the processor column index over which the first column of A is distributed;

IDESCA(9) the leading dimension (LDA) of the local array A storing the local blocks of A.

The descriptor array **does not change**, in general, throughout the life cycle of the matrix with which it is associated.

It is possible to reference an m by n **submatrix** of A, for example $A(i_A : m + i_A - 1, j_A : n + j_A - 1)$, by specifying the four parameters IA $= i_A$, JA $= j_A$, M $= m$ and N $= n$, in the interface of the routine called.
Figure 3 illustrates graphically the meaning and use of the parameters M, N, IA and JA and of some of the entries of the array descriptor IDESCA. The case depicted shows a matrix distributed over a 2×3 logical grid of processors. The first row of the matrix is stored in the second row of the grid (IDESCA(7) = 1), and the first column is stored in the third column of the grid (IDESCA(8) = 2). The index within each block refers to the processor (numbered from 0 to 5) which stores that block.

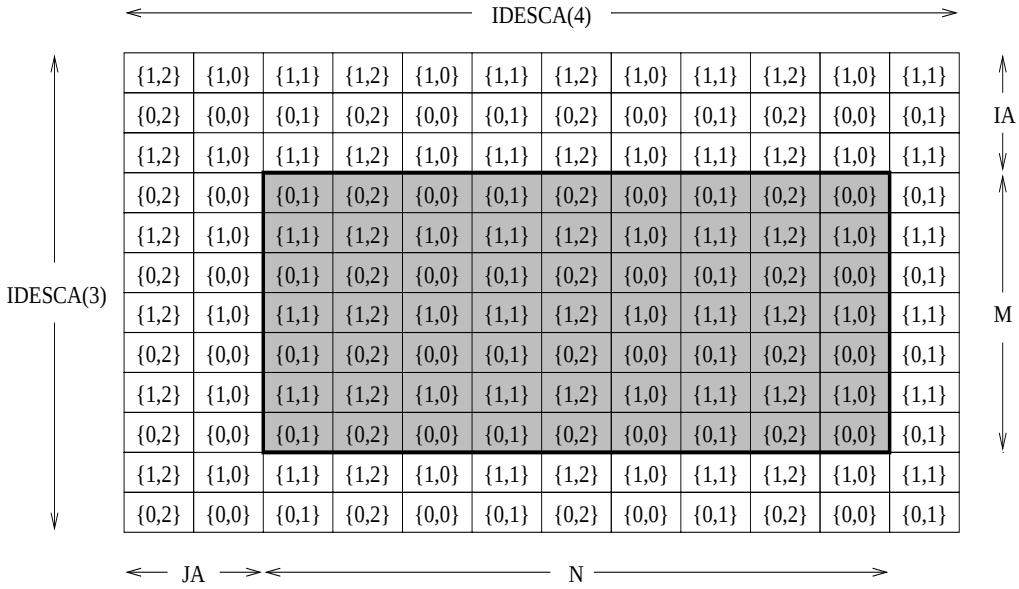

Figure 3
Referencing a submatrix

If the **whole** matrix is referenced, then, obviously, IA = 1, JA = 1, M = IDESCA(3) = m_A and N = IDESCA(4) = n_A.

In general, submatrices must start on the boundary between blocks, i.e., mod(IA−1,M_b) = 0 and mod(JA−1,N_b) = 0. However, submatrices of the matrices of the right-hand sides of the routines which solve system of simultaneous equations can start in **any column** of the overall matrices.

3.3.4 Error-handling and the diagnostic parameter INFO

Routines in this chapter do not use the usual NAG Parallel Library error-handling mechanism, involving the parameter IFAIL. Instead they have a diagnostic parameter INFO. (Thus they preserve compatibility with the ScaLAPACK specification.)

Whereas IFAIL is an **Input/Output** parameter and must be set before calling a routine, INFO is purely an **Output** parameter and need not be set before entry.

INFO indicates the success or failure of the computation, as follows:

INFO = 0 indicates successful termination;

INFO = $-(i \times 100 + j)$ indicates an error in the jth component of the ith argument (for example, a component of an array descriptor);

INFO = $-i$ indicates an error in the ith argument;

INFO > 0 indicates an error detected during execution.

It is **essential** to test INFO on exit from the routine (this corresponds to a **soft failure** in terms of the usual error-handling terminology used for the rest of the Library), both for argument errors and errors detected during execution.

3.4 Table of Available Routines

Type of matrix	**factorize**	**solve**	**condition number**
general real	F07ADFP PDGETRF	F07AEFP PDGETRS	
general complex	F07ARFP PZGETRF	F07ASFP PZGETRS	
symmetric positive-definite	F07FDFP PDPOTRF	F07FEFP PDPOTRS	
complex Hermitian positive-definite	F07FRFP PZPOTRF	F07FSFP PZPOTRS	
real triangular			F07TGFP PDTRCON

Each entry gives

the NAG routine name

the real or complex double precision ScaLAPACK routine name, when applicable