Chapter F11

Sparse Linear Algebra

Contents

1 Scope of the Chapter **2**

2 Background to the Problems **2**
2.1 Iterative Methods . . . 2
2.2 Unsymmetric Systems of Simultaneous Linear Equations . . . 3
2.3 Symmetric Systems of Simultaneous Linear Equations . . . 3
2.4 Sparse Matrices and Their Storage . . . 4
2.4.1 Coordinate Storage (CS) Format . . . 4
2.4.2 Sparse Matrices with Symmetric Sparsity Pattern . . . 5
2.5 Data Distribution . . . 5
2.5.1 Sparse Matrices in Matrix-Free Solvers . . . 5
2.5.2 Sparse Matrices Represented in Coordinate Storage Format . . . 5
2.5.3 Dense Vectors in Matrix-Free Solvers . . . 5
2.5.4 Conformally Distributed Dense Vectors . . . 6
2.6 Local Matrices . . . 6
2.7 Preconditioners . . . 7
2.8 Parallel Sparse Matrix Operations . . . 8
2.8.1 Reverse-Communication Solvers . . . 8
2.8.2 Matrix-Vector Multiplication . . . 8
2.8.3 Block-Jacobi Preconditioning . . . 9
2.9 References . . . 9

3 Recommendations on Choice and Use of Available Routines **10**
3.1 Types of Routines Available . . . 10
3.2 Auxiliary Information for Parallel Sparse Matrix Computations . . . 11
3.2.1 Contents of IAINFO . . . 11
3.2.2 Sequence of Accesses to IAINFO . . . 12
3.2.3 Required Size of IAINFO . . . 12
3.3 Utilities for Sparse Matrix Operation . . . 13
3.4 Unsymmetric Systems of Simultaneous Linear Equations . . . 13
3.5 Symmetric Systems of Simultaneous Linear Equations . . . 13

1 Scope of the Chapter

This chapter provides routines for the solution of large sparse systems of simultaneous linear equations. These consist of **iterative** methods for real unsymmetric and symmetric linear systems.

2 Background to the Problems

This section gives only a brief introduction to the solution of sparse linear systems. For more detailed discussions see, for example, Axelsson [2], Barrett *et al.* [3], Bruaset [4], or Saad [10].

2.1 Iterative Methods

The routines in this chapter are designed to solve a system of simultaneous linear equations $Ax = b$ of order n, where the matrix A is large and sparse, either unsymmetric or symmetric (positive-definite or indefinite).

All routines in this chapter employ **iterative methods**, where the solution is approached through a sequence of iterations, until some user-specified termination criterion is met or until some predefined maximum number of iterations has been carried out. The number of iterations required for convergence cannot generally be known in advance, as it depends on the matrix of the coefficients — its sparsity pattern, conditioning and spectrum of eigenvalues — and on the **preconditioner** employed (see below), if any, and on the accuracy required.

The methods employed in this chapter start from the residual $r_0 = b - Ax_0$, where x_0 is an initial estimate for the solution (often $x_0 = 0$), and generate an orthogonal basis for the Krylov subspace span$\{A^k r_0\}$, for $k = 0, 1, \ldots$, by means of recurrence relations (Golub and Van Loan [6]). Here and in the following, the subscript k denotes the iteration count. A sequence of upper Hessenberg matrices $\{H_k\}$, for unsymmetric A, or of symmetric tridiagonal matrices $\{T_k\}$, for symmetric A, is also generated. The resulting upper Hessenberg or symmetric tridiagonal systems of equations are usually more easily solved than the original problem. A sequence of solution iterates $\{x_k\}$ is thus generated such that the sequence of the norms of the residuals $\{\|r_k\|\}$ converges to the required tolerance. Note that, in general, the convergence is not monotonic.

In exact arithmetic, after n iterations, this process is equivalent to an orthogonal reduction of A to upper Hessenberg form, $H_n = Q^T AQ$ or to symmetric tridiagonal form, $T_n = Q^T AQ$, for unsymmetric or symmetric A, respectively; the solution x_n would thus achieve exact convergence. In finite-precision arithmetic, cancellation and round-off errors accumulate causing loss of orthogonality. These methods must therefore be viewed as genuinely iterative methods, able to converge to a solution **within a prescribed tolerance**.

Faster convergence can be achieved using a **preconditioner** (Golub and Van Loan [6], Barrett *et al.* [3]). A preconditioner maps the original system of equations onto a transformed system, say

$$\bar{A}\bar{x} = \bar{b}, \tag{1}$$

with, hopefully, better characteristics with respect to its speed of convergence: for example, the condition number of the matrix of the coefficients can be improved or eigenvalues in its spectrum can be made to coalesce. An orthogonal basis for the Krylov subspace span$\{\bar{A}^k \bar{r}_0\}$, for $k = 0, 1, \ldots$, is generated and the solution proceeds as outlined above. The algorithms used produce the solution and residual iterates of the **original** system $Ax = b$. An unsuitable preconditioner may result in a very slow rate or lack of convergence. However, preconditioning involves a trade-off between the reduction in the number of iterations required for convergence and the additional computational costs per iteration. Also, setting up a preconditioner may involve non-negligible overheads. The application of preconditioners to unsymmetric and symmetric systems of equations is further considered in Sections 2.2 and 2.3.

Termination criteria can involve the solution of either the original or the preconditioned system of equations. In both cases, bounds derived from **backward** error analysis are employed to ensure that the computed solution is the exact solution of a problem as close to the original as the termination tolerance requires. Termination criteria could involve bounds derived from **forward** error analysis. However, any such criteria would require information about the condition number $\kappa(A)$ or $\kappa(\bar{A})$, which is not easily obtainable.

All methods, for both unsymmetric and symmetric problems, can use the termination criterion

$$\|r_k\|_p \leq \tau \left(\|b\|_p + \|A\|_p \, \|x_k\|_p\right) \tag{2}$$

where $p = 1, 2$ or ∞, and τ denotes a user-specified tolerance, such that $\epsilon \leq \tau < 1$, where ϵ is the ***machine precision***. Facilities are provided for the estimation of the norm $\|A\|_p$ in (2) using Higham's method (Higham [8]), when this is not known in advance. Note that when A is not symmetric, Higham's method in addition to $v = Au$, also requires matrix–vector products of the form $v = A^T u$.

The termination criteria based on the preconditioned system of equations can use only the l_2-norm, i.e., $p = 2$. They vary with the method of solution and will be described for each method in Sections 2.2 and 2.3. These criteria monitor the solution of the problem actually being solved, while the criterion (2) focusses on the solution of the original problem. With a judicious choice of preconditioner, either of the two criteria can be used, though the computational costs involved may differ.

Optionally, a vector w of user-specified weights can be used in the computation of the vector norms in (2). The **weighted** l_p-norm of a vector v is then defined by $\|[w_1 v_1, w_2 v_2 \ldots, w_n v_n]^T\|_p$. Note that the use of weights increases the computational costs.

2.2 Unsymmetric Systems of Simultaneous Linear Equations

At this release, only the Restarted Generalised Minimal Residual method (RGMRES) is provided for systems with an unsymmetric matrix A (Saad and Schultz [11], Barrett *et al.* [3], Dias da Cunha and Hopkins [5]). An orthogonal basis for the Krylov subspace span$\{\bar{A}^k \bar{r}_0\}$, for $k = 1, 2, \ldots,$ is explicitly generated: this is referred to as Arnoldi's method (Arnoldi [1]). The solution is then expanded onto the orthogonal basis so as to minimize the residual norm $\|\bar{b} - \bar{A}\bar{x}\|_2$.

The lack of symmetry of A implies that the orthogonal basis is generated by applying long recurrence relations, whose length increases linearly with the iteration count. For all but the most trivial problems, computational and storage costs can quickly become prohibitive as the iteration count increases. RGMRES limits these costs by employing a restart strategy: every m iterations, the Arnoldi process is restarted from $\bar{r}_l = \bar{b} - \bar{A}\bar{x}_l$, where the subscript l denotes the last available iterate. Each group of m iterations is referred to as a 'super-iteration'. The value of m is chosen in advance and is fixed throughout the computation. Unfortunately, an optimum value of m cannot easily be predicted.

Any preconditioner M such that $\bar{A} = M^{-1}A \approx I_n$, where I_n is the identity matrix of order n, can be used in (1). With this choice, $\bar{b} = M^{-1}b$ and $\bar{x} = x$ are used in (1). Note that neither the preconditioning matrix M nor its inverse M^{-1} have to be provided explicitly. Only the computation of matrix–vector products of the form $v = Au$ and the solution of the preconditioning equations $Mv = u$ are required.

A termination criterion, based on the preconditioned system of equations, is available:

$$\|\bar{r}_k\|_2 \leq \tau \left(\|\bar{b}\|_2 + \sigma_1(\bar{A})\|x_k\|_2\right) \tag{3}$$

where $\sigma_1(\bar{A}) = \|\bar{A}\|_2$ is the largest singular value of the (preconditioned) iteration matrix $\bar{A}$. When $\sigma_1(\bar{A})$ is not supplied, the inexpensive estimate $\sigma_1(\bar{A}) \approx \max_k \|U_k\|_1$ is used instead, where $\{U_k\}$ is a sequence of upper triangular matrices orthogonally similar to the sequence of upper Hessenberg matrices $\{H_k\}$ generated by the Arnoldi method. Note that only order of magnitude estimates are required by the termination criterion.

Termination criterion (3) can be applied during any iteration, while criterion (2) can be applied anywhere only if user-specified weights are not used, otherwise it can only be applied at the end of a super-iteration.

2.3 Symmetric Systems of Simultaneous Linear Equations

At this release, two methods of solution are provided when the matrix A is symmetric.

Conjugate Gradient Method. For this method (Hestenes and Stiefel [7], Golub and Van Loan [6], Barrett *et al.* [3], Dias da Cunha and Hopkins [5]), the matrix A should ideally be positive-definite. The application of the Conjugate Gradient method to indefinite matrices may lead to failure or to lack of convergence.

Lanczos Method (SYMMLQ). This method, based upon the algorithm SYMMLQ (Paige and Saunders [9], Barrett *et al.* [3]), is suitable for both positive-definite and indefinite matrices. It is more robust than the Conjugate Gradient method but less efficient when A is positive-definite.

In both methods, three-term recurrence relations are used to generate an orthogonal basis for the Krylov subspace span$\{\bar{A}^k\bar{r}_0\}$, for $k = 1, 2, \ldots$. The orthogonal basis is not formed explicitly. Their basic difference lies in the method of solution of the resulting symmetric tridiagonal systems of equations: the Conjugate Gradient method carries out an LDL^T factorization whereas the Lanczos method (SYMMLQ) uses an LQ factorization, where L is lower triangular, D is diagonal, and Q is an orthogonal matrix.

Any preconditioners must be symmetric and positive-definite, i.e., representable by $M = EE^T$, where M is non-singular, and such that $\bar{A} = E^{-1}AE^{-T} \approx I_n$ in (1), where I_n is the identity matrix of order n. Also, we can define $\bar{b} = E^{-1}b$ and $\bar{x} = E^T x$. Note that neither the preconditioning matrix M nor its inverse M^{-1} have to be provided explicitly. Only the computation of matrix–vector products of the form $v = Au$ and the solution of the preconditioning equations $Mv = u$ are required.

A termination criterion based on the preconditioned system of equations is available for each method:

$\|M^{-1}r_k\|_2 \le \tau\,\sigma_1(\bar{A})\|x_k\|_2$, for the Conjugate Gradient method;

$\|\bar{r}_k\|_2 \le \tau\,\max(1.0,\,\|b\|_2/\|r_0\|_2)\,(\|\bar{r}_0\|_2 + \sigma_1(\bar{A})\,\|\bar{x}_k - \bar{x}_{k-1}\|_2)$, for the Lanczos method (SYMMLQ);

where $\sigma_1(\bar{A}) = \|\bar{A}\|_2$ is the largest singular value of the (preconditioned) iteration matrix $\bar{A}$. When $\sigma_1(\bar{A})$ is not supplied, facilities are provided for its estimation by $\sigma_1(\bar{A}) \approx \max_k \sigma_1(T_k)$. The interlacing property $\sigma_1(T_{k-1}) \le \sigma_1(T_k)$ and Gerschgorin's theorem provide lower and upper bounds for $\sigma_1(T_k)$. These bounds can be used to compute $\sigma_1(T_k)$ using the bisection method. Note that only order of magnitude estimates are required by the termination criterion.

2.4 Sparse Matrices and Their Storage

A matrix A can be termed **sparse** if special techniques can be utilised to take advantage of the zero elements and their locations.

If A is sparse, and the chosen algorithm requires the matrix coefficients to be stored, a significant saving in storage can be made by storing only non-zero elements. A number of different storage formats may be used to represent sparse matrices economically. These differ in the amount of storage required, in the degree of indirect addressing necessary to perform fundamental operations such as matrix–vector products, and in their suitability for vector and/or parallel architectures. For a survey of the different storage formats see Barrett *et al.* [3].

Some of the routines in this chapter have been designed to be independent of the matrix storage format. This allows the user to choose his or her own preferred format, or to avoid storing the matrix altogether. Other routines may be easier to use, but are based on fixed storage formats. Currently, one such fixed format, a distributed variant of the coordinate storage (CS) format, is supported.

2.4.1 Coordinate Storage (CS) Format

This storage format represents a sparse unsymmetric matrix A, with NNZ non-zero elements, in terms of a real array A and two integer arrays IROW and ICOL. These arrays are all of dimension 1 and of length at least NNZ. The array A contains the numerical values of the non-zero elements, while IROW and ICOL store the corresponding row and column indices respectively.

For example, the matrix

$$A = \begin{pmatrix} 1 & 2 & -1 & -1 & -3 \\ 0 & -1 & 0 & 0 & -4 \\ 3 & 0 & 0 & 0 & 2 \\ 2 & 0 & 4 & 1 & 1 \\ -2 & 0 & 0 & 0 & 1 \end{pmatrix}$$

has 15 non-zero elements, NNZ = 15, and can be represented in the arrays A, IROW and ICOL as

$$\mathrm{A} = (1, 2, -1, -1, -3, -1, -4, 3, 2, 2, 4, 1, 1, -2, 1)$$

$$\mathrm{IROW} = (1, 1, 1, 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5)$$

$$\mathrm{ICOL} = (1, 2, 3, 4, 5, 2, 5, 1, 5, 1, 3, 4, 5, 1, 5)$$

Note:

(i) The matrix elements do not have to appear in the arrays A, IROW and ICOL in any particular order. For instance, the above matrix A can be respresented as

$$\mathrm{A} = (1,-2,1,1,4,2,2,3,-4,-1,-3,-1,-1,2,1)$$

$$\mathrm{IROW} = (5,5,4,4,4,4,3,3,2,2,1,1,1,1,1)$$

$$\mathrm{ICOL} = (5,1,5,4,3,1,5,1,5,2,5,4,3,2,1)$$

However, for efficiency reasons a specific ordering is required by most routines and a set-up routine is provided to order the elements appropriately.

(ii) With this storage format it is possible to enter duplicate elements, i.e., elements with identical row and column indices. These may be dealt with in different ways. For example, an error may be raised, all but the first (or the last) duplicate entry may be ignored, or the duplicate entries may be summed.

2.4.2 Sparse Matrices with Symmetric Sparsity Pattern

In this release there is no special storage format particularly tailored to symmetric matrices. However, some routines can take algorithmic advantage of matrices A which are unsymmetric, but have a symmetric **sparsity pattern**. This means that the non-zero structure of A is symmetric:

$$a_{ij} \neq 0 \quad \text{if and only if} \quad a_{ji} \neq 0 \quad \text{for} \quad i,j = 1,\ldots,n.$$

2.5 Data Distribution

2.5.1 Sparse Matrices in Matrix-Free Solvers

The basic solver routines are **matrix-free**, i.e., the matrix A of a given system of linear equations does **not** have to be provided explicitly by the user program. This implies that the matrix and the preconditioner, if used, can be distributed over the processors as best suits the problem being solved.

2.5.2 Sparse Matrices Represented in Coordinate Storage Format

In this release, the routines in this chapter use a **cyclic row block distribution** for the sparse matrices represented explicitly in coordinate storage format. In the cyclic row block distribution, blocks of M_b contiguous rows of a matrix A are distributed on the logical grid of processors cyclically row by row (i.e., in the row major ordering of the grid) starting from the $\{0,0\}$ logical processor. The non-zero elements in the row blocks assigned to a logical processor are represented in coordinate storage format and are stored in the **local** arrays A, IROW and ICOL.

Figure 1 and Figure 2 illustrate the cyclic row block distribution of a matrix consisting of twelve row blocks on a two by three logical grid of processors. The row blocks of the matrix are numbered from 1 to 12.

2.5.3 Dense Vectors in Matrix-Free Solvers

In the matrix-free solver routines, the vectors may be distributed in two different ways, as specified by the input parameters of the respective set-up routines.

First, vectors may be distributed across all processors in the grid, with different processors holding different parts of the vector. In this case, all vectors are distributed in the same way. This distribution must be chosen if the solver routines are to be used in connection with any F11 routine other than the corresponding set-up or diagnostic routines.

Second, on initial entry to the solver routines, **input vectors** may be distributed along the first column or row of the logical grid of processors. Each processor in the first column or row then broadcasts the elements stored locally to all the other processors in the same row or column, respectively. Thereafter, **all** vectors in the solve routine, input, output and internal vectors alike, are distributed identically along all the columns or the rows of the logical grid of processors. Hence, if vectors are distributed by column

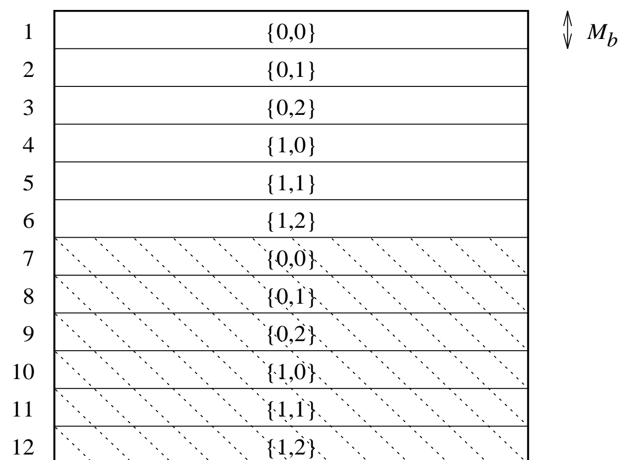

Figure 1
Cyclic row block distribution over a 2 by 3 logical grid of processors

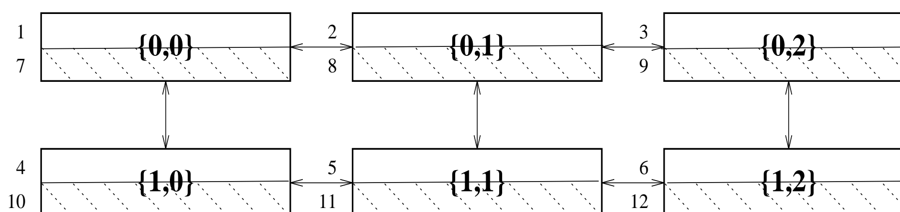

Figure 2
Data distribution from the processors' point of view

or row, all processors in the same row or column, respectively, of the grid will hold copies of the same vector elements.

In either of the distribution schemes, details of the distribution pattern are not required by any of the routines in the suites, only the number of vector elements stored in a processor.

2.5.4 Conformally Distributed Dense Vectors

Whenever a sparse matrix A is represented explicitly in coordinate storage format, any vector x used in conjunction with A must be distributed **conformally** to A. This means that the elements of the vector x are **aligned** with the **rows** of A, i.e., the element $x(i)$ and the row $A(i, 1:n)$, for $i = 1, \ldots, n$, are assigned to same logical processor. This kind of distribution can also be obtained by interpreting x as a column vector and distributing x across the logical processor grid identically to each of the columns $A(1:n, j)$, $j = 1, \ldots, n$.

Figure 3 illustrates the conformal distribution of a vector; the matrix and distribution parameters are chosen as in Figure 1.

2.6 Local Matrices

In order to allow an efficient implementation of sparse matrix operations, a user-specified distributed sparse matrix A, represented in coordinate storage format, has to undergo a preprocessing phase. This not only includes

a) dealing with entries with duplicate row and column indices: entries with duplicate row and column indices may be removed, or their values may be summed; and

b) dealing with entries with zero value: entries with zero values may optionally be removed;

but also

c) transforming the user-supplied index values in IROW and ICOL; and

d) reordering the non-zero elements and identifying specific parts of the matrix A.

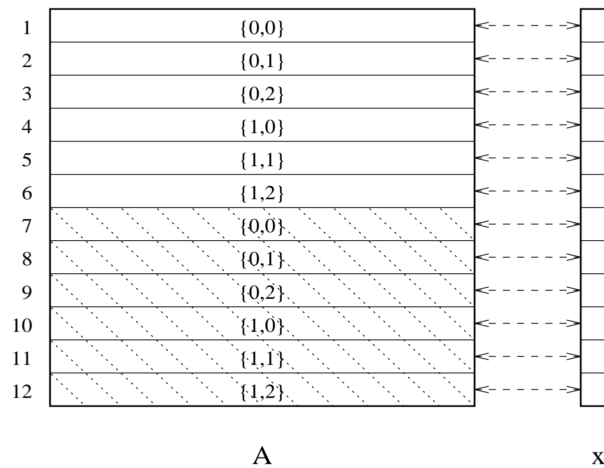

Figure 3
Conformal distribution of vector over a 2 by 3 logical grid of processors

Each real sparse unsymmetric matrix A, represented in distributed coordinate storage format, has to be appropriately preprocessed before any other library routine can be applied to A. The following terms and notational conventions are used in this context:

Rows Assigned to a Logical Processor
Let $\mathcal{I}$ denote the index set of the rows assigned to a logical processor. Then, m_l is used to denote the number of elements in $\mathcal{I}$, i.e., the number of rows assigned to a logical processor.

Internal and Interface Coefficients and Indices
A local non-zero element a_{ij} is said to be an **internal coefficient** if $j \in \mathcal{I}$. Otherwise, a_{ij} is said to be an **interface coefficient**, and the indices i and j are referred to as **internal** and **external interface indices**. The numbers of internal and external interface indices on a logical processor are denoted n^i_{int} and n^e_{int}, respectively.

Global and Local Indexing
It is assumed that the same numbering of unknowns is used on all logical processors before a distributed matrix represented in coordinate storage format is preprocessed. The corresponding row and column indices are therefore referred to as **global** indices. In the course of the preprocessing actions, the global indices are transformed in a way that depends on and is tailored to the data distribution and the sparsity pattern of A. The transformation used on each processor is based on a **local** permutation of the index set $\{1, 2, \ldots, n\}$, i.e., different logical processors employ different permutations. The resulting row and column indices are referred to as **local indices**. Note that an unknown is usually assigned different local index values on different logical processors.

Local Diagonal Blocks
The symbol n_{LB} is used to denote the number of row blocks assigned to a logical processor. Let $\mathcal{I}_k$, for $k = 1, 2, \ldots, n_{\mathrm{LB}}$, be the set of row indices associated with each block. Then, the **local diagonal blocks** A_k, for $k = 1, 2, \ldots, n_{\mathrm{LB}}$, are defined by $A_k = (a_{ij})_{i,j \in \mathcal{I}_k}$.

These relationships are illustrated in Figure 4 using a 16 by 16 sparse matrix distributed on a 2 by 1 processor grid using a block size $M_b = 8$. The non-zero elements are marked by crosses and circles. Specifically, crosses signify internal and circles signify external coefficients. The rows assigned to the first processor are identified by solid, the rows assigned to the second processor by dashed horizontal lines. Solid sections of vertical lines indicate parts of matrix contained in local diagonal blocks; dashed sections indicate off-diagonal block parts.

2.7 Preconditioners

In this release **block Jacobi preconditioners**, also known as **non-overlapping additive Schwarz preconditioners**, are provided. These are obtained by (approximately) inverting the local diagonal blocks A_k, for $k = 1, 2, \ldots, n_{\mathrm{LB}}$, on each processor. The approximate inverse of each local diagonal block A_k is calculated using an **incomplete** *LU* **factorization**. This means that A_k is decomposed in the form

$$A_k = M_k + R_k$$

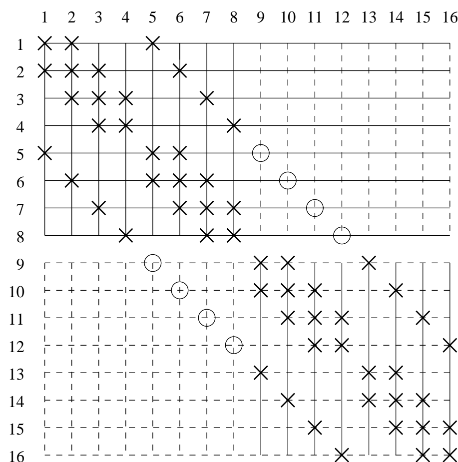

Figure 4
Local Diagonal Blocks and Interface Coefficients

where

$$M_k = P_k L_k D_k U_k Q_k$$

and L_k is lower triangular with unit diagonal elements, D_k is diagonal, U_k is upper triangular with unit diagonal elements, P_k and Q_k are permutation matrices, and R_k is a remainder matrix. The preconditioning matrix M is distributed identically to the coefficient matrix A on the logical grid of processors. Therefore, the local diagonal blocks of the block diagonal matrix M are given by M_k, $k = 1, 2, \ldots, n_{\mathrm{LB}}$.

The quality of block Jacobi preconditioners depends largely on (i) the coupling between unknowns associated with different local diagonal blocks and (ii) the accuracies of the incomplete LU factorizations. An indicator of the coupling between unknowns associated with different local diagonal blocks is the number and size of non-zero elements in off-diagonal parts of A relative to the number and size of non-zero elements in diagonal blocks. If this coupling is strong, then block Jacobi preconditioners are generally of poor quality. If this coupling is weak, then block Jacobi preconditioners can be reasonably effective, provided the incomplete LU factorizations yield approximate inverses of sufficient accuracy. An indicator of the accuracy of an incomplete LU factorization is the size of the non-zero elements in the remainder matrix R_k relative to the the size of the non-zero elements in A_k.

2.8 Parallel Sparse Matrix Operations

2.8.1 Reverse-Communication Solvers

The main computational kernels of the reverse-communication routines in each suite are Level-1 BLAS operations. These are performed on each logical processor independently. The only exception is the global summation operation required to calculate dot products. In the global summation operation, communications take place across the whole processor grid, or only along the columns of the logical grid or along its rows, depending on the way vectors have been distributed.

2.8.2 Matrix-Vector Multiplication

The matrix-vector product $Au = v$ is computed by each logical processor calculating $v_{\mathrm{loc}} = A_{\mathrm{loc}} u$, where $v_{\mathrm{loc}} := (v_j)_{j \in \mathcal{I}}$ is the local part of v and $A_{\mathrm{loc}} = (a_{ij})_{i \in \mathcal{I}, j \in \{1,\ldots,n\}}$ is the local part of the matrix A. If the matrix A_{loc} is partitioned into $A_{\mathrm{loc}}^{(l)} = (a_{ij})_{i,j \in \mathcal{I}}$ and $A_{\mathrm{loc}}^{(e)} = (a_{ij})_{i \in \mathcal{I}, j \notin \mathcal{I}}$ and the vector u is partitioned into $u_{\mathrm{loc}} = (u_j)_{j \in \mathcal{I}}$ and $u_{\mathrm{ext}} = (u_j)_{j \notin \mathcal{I}}$, then the relationship

$$v_{\mathrm{loc}} = A_{\mathrm{loc}} u = A_{\mathrm{loc}}^{(l)} u_{\mathrm{loc}} + A_{\mathrm{loc}}^{(e)} u_{\mathrm{ext}}$$

is obtained. Thus, the matrix-vector product $Au = v$ can be calculated by performing the following sequence of algorithmic steps on each processor:

Step 1) send elements of u_{loc} required by other logical processors;

Step 2) calculate $v_{\mathrm{loc}} = A_{\mathrm{loc}}^{(l)} u_{\mathrm{loc}}$;

Step 3) receive required elements of u_{ext};

Step 4) calculate $v_{\text{loc}} = v_{\text{loc}} + A_{\text{loc}}^{(e)} u_{\text{ext}}$

Step 2) can be performed using the data stored locally, whereas Step 4) involves elements of the vector u stored on other processors. Note that only elements u_j, where j is an external interface index (see Section 2.6), are required to perform Step 4). Hence, while the number of floating-point operations in $A_{\text{loc}}u$ depends only on the number of non-zero elements stored locally, the number of elements of the vector u to be sent to or received from other processors additionally depends on the inter-relationship between the distribution and the sparsity pattern of A. If the numbers of non-zero elements in the blocks $A_{\text{loc}}^{(l)}$ are large compared to the numbers of non-zero elements in the blocks $A_{\text{loc}}^{(e)}$, there is a high **computation-to-communication ratio** and the efficiency of the parallel algorithm can be expected to be satisfactory. On the other hand, if the **computation-to-communication ratio** is low, the overhead associated with communication operations usually results in a poor program performance.

Note that the above algorithmic organization allows for an optimal overlap of computation and communication when calculating matrix-vector products. A similar technique can also be used for the transposed matrix-vector product $A^T u = v$.

2.8.3 Block-Jacobi Preconditioning

The main advantage of block Jacobi preconditioning in a parallel environment is that solving the preconditining equation $Mv = u$ does not require any inter-processor communication. This is due to the fact that M is block diagonal and that each diagonal block of M is stored on a single logical processor. Let $\mathcal{I}_k$, for $k = 1, 2, \ldots, n_{\text{LB}}$, be the index set associated with the local diagonal block M_k of M, and let u_k and v_k denote the corresponding parts of u and v, respectively. Then, the local computations required to solve the preconditining equation $Mv = u$ are the subsystem solves $M_k v_k = u_k$, $k = 1, 2, \ldots, n_{\text{LB}}$, which can be performed on each processor independently.

2.9 References

[1] Arnoldi W (1951) The principle of minimized iterations in the solution of the matrix eigenvalue problem *Quart. Appl. Math.* **9** 17–29

[2] Axelsson O (1996) *Iterative Solution Methods* Cambridge University Press, Cambridge

[3] Barrett R, Berry M, Chan T F, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C and van der Vorst H (1994) *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* SIAM, Philadelphia

[4] Bruaset A M (1995) A survey of preconditioned iterative methods *Pitman Research Notes in Mathematics* **328** Longman Scientific & Technical, Harlow, Essex

[5] Dias da Cunha R and Hopkins T (1994) PIM 1.1 — the parallel iterative method package for systems of linear equations user's guide — Fortran 77 version *Technical Report* Computing Laboratory, University of Kent at Canterbury, Kent CT2 7NZ, UK

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

[7] Hestenes M and Stiefel E (1952) Methods of conjugate gradients for solving linear systems *J. Res. Nat. Bur. Stand.* **49** 409–436

[8] 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

[9] Paige C C and Saunders M A (1975) Solution of sparse indefinite systems of linear equations *SIAM J. Numer. Anal.* **12** 617–629

[10] Saad Y (1996) *Iterative Methods for Sparse Linear Systems* PWS Publishing Company, Boston, MA

[11] Saad Y and Schultz M (1986) GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **7** 856–869

3 Recommendations on Choice and Use of Available Routines

3.1 Types of Routines Available

The routines available in this chapter are divided essentially into three types: basic routines, utility routines and black boxes.

Basic routines are grouped in suites of three, and implement the underlying iterative method. Each suite comprises a set-up routine, a solver, and a routine to return additional information. The solver routine is independent of the matrix storage format (indeed the matrix need not be stored at all) and the type of preconditioner. It uses **reverse communication**, i.e., it returns repeatedly to the calling program with the parameter IREVCM set to specified values which require the calling program to carry out a specific task (either to compute a matrix–vector product or to solve the preconditioning equation), to signal the completion of the computation or to allow the calling program to monitor the solution.

Reverse communication has the following advantages.

(i) Maximum flexibility in the representation and storage of sparse matrices. All matrix operations are carried out outside the solver routine, thereby avoiding the need for a complicated interface with enough flexibility to cope with all types of storage schemes and sparsity patterns. This applies also to preconditioners.

(ii) Enhanced user interaction: the progress of the solution can be closely monitored by the user and tidy or immediate termination can be requested. This is useful, for example, when alternative termination criteria are to be employed or in case of failure of the external routines used to perform matrix operations.

At present there are two suites of basic routines, for real symmetric, and for real unsymmetric systems, respectively.

The following section of code illustrates a simple loop to call one of the iterative solvers (F11GBFP) using reverse communication. The loop is preceded by a call to the set-up routine (F11GAFP) and, optionally, followed by a call to the third routine in the suite.

```
*
*     Initialize the suite
*
      CALL F11GAFP( ... , IFAIL)
      IF (IFAIL .... ) ....
*
*     Solve the equations
*
      LOOP = .TRUE.
      IREVCM = 0
   10 CONTINUE
         CALL F11GBFP( ... , IREVCM, ... , IFAIL)
*
         IF (IREVCM.EQ.1) THEN
*
*           Matrix-vector product
*
            .........
*
         ELSE IF (IREVCM.EQ.2) THEN
*
*           Solve the preconditioning equations
*
            .........
*
         ELSE IF (IREVCM.EQ.3) THEN
*
*           Monitoring step (optional)
```

```
*
            .........
*
         ELSE IF (IREVCM.EQ.4) THEN
*
*           Completion
*
            LOOP = .FALSE.
         END IF
      IF (LOOP) GO TO 10
      IF (IFAIL ... ) ...
*
*     Get additional information
*
      CALL F11GCFP( ... )
```

Utility routines perform such tasks as initializing the preconditioning matrix M, solving linear systems involving M, or computing matrix-vector products, for particular preconditioners and matrix storage formats. Used in combination basic routines and utility routines therefore provide iterative methods with a considerable degree of flexibility; allowing the user to select from different termination criteria, monitor the approximate solution, and compute various diagnostic parameters. The tasks of computing the matrix-vector products and dealing with the preconditioner are removed from the user, but at the expense of sacrificing some flexibility in the choice of preconditioner and matrix storage format.

Black box routines call basic and utility routines in order to provide easy-to-use routines for particular preconditioners and sparse matrix storage formats. They are much less flexible than the basic routines, but do not use reverse communication, and may be suitable in many simple cases.

The structure of this chapter has been designed to cater for as many types of application as possible. If a black box exists which is suitable for a given application you are recommended to use it. If you then decide you need some additional flexibility it is easy to achieve this by using basic and utility routines which reproduce the algorithm used in the black box, but allow more access to algorithmic control parameters and monitoring. If you wish to use a preconditioner or storage format for which no utility routines are provided, you must call basic routines, and provide your own utility routines.

3.2 Auxiliary Information for Parallel Sparse Matrix Computations

Most operations involving a sparse matrix A stored in distributed coordinate storage representation (see Section 2.4.1) require additional auxiliary information about A in order to be performed efficiently in parallel by multiple processors. This auxiliary information is stored in and retrieved from the associated array IAINFO by a number of routines in Chapters F01, F11 and X04. .

3.2.1 Contents of IAINFO

While most of the data contained in IAINFO is meant to be used **only** internally by library routines, the following array elements are crucial to determining the required size of local arrays and therefore may need to be accessed by user programs:

IAINFO(1)	–	On successful exit, whenever a routine adds information to IAINFO, the element r = IAINFO(1) contains the maximum number of elements of IAINFO used by that routine. Thus, r provides the minimum size of IAINFO required to complete the routine successfully.
IAINFO(2)	–	On successful exit, whenever a routine adds information to IAINFO, the element s = IAINFO(2) contains the number of elements of IAINFO which can subsequently be used by other library routines. Thus the first s elements of IAINFO must not be changed by the user program between any subsequent calls to library routines involving the matrix A.
IAINFO(3)	–	m_l, the number of rows of A stored on the local processor (see Section 2.6).
IAINFO(5)	–	$m_l^{\max}$, the maximum number of rows of A stored on any processor of the Library Grid.
IAINFO(6)	–	n_{int}^i, the number of internal interface indices for the local processor (see Section 2.6).
IAINFO(7)	–	n_{int}^e, the number of external interface indices for the local processor (see Section 2.6).
IAINFO(8)	–	n_{LB}, the number of row blocks stored on the local processor (see Section 2.6).

3.2.2 Sequence of Accesses to IAINFO

Fundamental items of information are stored in IAINFO by an appropriate **general set-up routine**. In this release, only one such general set-up routine, F11ZAFP, is available. Since F11ZAFP initialises the array IAINFO, it has to be called once for every real sparse matrix A, represented in distributed coordinate storage format, before any other library routine can be applied to A.

Additional auxiliary information needed for some sparse matrix operations is stored in IAINFO by corresponding **operation-specific set-up routines**. In this release, operation-specific set-up routines are available for matrix-vector multiplication, F11XAFP, and block Jacobi preconditioning, F11DAFP. These set-up routines have to be called once before the corresponding operations can be performed using F11XBFP and F11DBFP, respectively.

The direction of arrows in Figure 5 illustrates the order in which the array IAINFO is supposed to be accessed (and possibly modified for subsequent routine calls) by different routines in Chapters F01, F11 and X04. Routines which modify IAINFO are contained in solid boxes, while routines which merely retrieve data from IAINFO are signified by dashed boxes. Note that routines not connected by arrows can be called in any order.

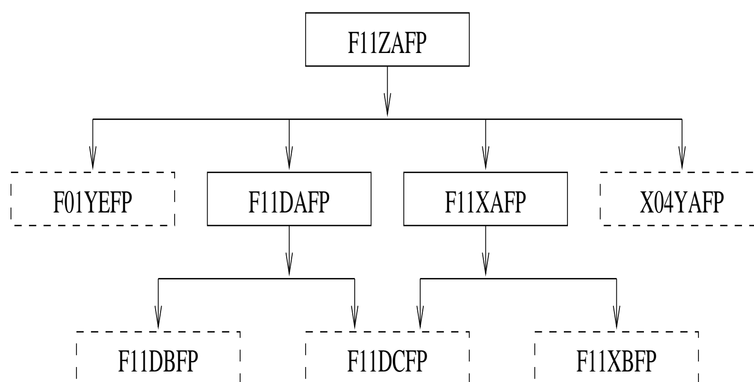

Figure 5
Sequence of Accesses to IAINFO

3.2.3 Required Size of IAINFO

The size of IAINFO required to store all auxiliary information needed to perform a set of sparse matrix operations depends on a number of problem characteristics, most importantly, the order of the matrix A, the partitioning of A into row blocks, and the sparsity pattern of A. The dependency of the required size of IAINFO on the sparsity pattern of A, in particular, makes it difficult to determine in advance the storage requirement for IAINFO in a set-up routine. Therefore, suggested values for the size of IAINFO

in set-up routines can prove insufficient for particular sparse matrices. In this case it is recommended to increase the storage space according to the value of IAINFO(1) returned from the respective set-up routine.

3.3 Utilities for Sparse Matrix Operation

The utility routines provided for sparse matrices use the coordinate storage (CS) format described in Section 2.4.1. The general set-up routine F11ZAFP must be called for each real sparse matrix A, represented in distributed coordinate storage format, before any other library routine can be applied to A (see Section 2.6).

F11DAFP computes a preconditioning matrix based on incomplete LU factorizations of local diagonal blocks (see Section 2.7), and F11DBFP solves linear systems involving the preconditioner generated by F11DAFP (see Section 2.8.3). The amount of fill-in occurring in the incomplete factorizations can be controlled by specifying either the level of fill, or the drop tolerance. Partial or complete pivoting may optionally be employed, and the factorizations can be modified to preserve row-sums.

F11XBFP computes matrix-vector products for unsymmetric matrices. Prior to F11XBFP the set-up routine F11XAFP must be called.

3.4 Unsymmetric Systems of Simultaneous Linear Equations

The suite of basic routines F11BAFP, F11BBFP and F11BCFP implements RGMRES for the iterative solution of the sparse unsymmetric linear system $Ax = b$. These routines allow a choice of termination criteria and the norms used in them, allow monitoring of the approximate solution, and can return estimates of the norm of A and the largest singular value of the preconditioned matrix $\bar{A}$.

The black box routine F11DCFP makes calls to F11BAFP, F11BBFP, F11BCFP, F11DBFP and F11XBFP, to solve a sparse unsymmetric linear system, represented in CS format, using RGMRES with block Jacobi preconditioning.

3.5 Symmetric Systems of Simultaneous Linear Equations

The suite of basic routines F11GAFP, F11GBFP and F11GCFP implement either the conjugate gradient (CG) method, or a Lanczos method based on SYMMLQ, for the iterative solution of the sparse symmetric linear system $Ax = b$. If A is known to be positive-definite the CG method should be chosen; the Lanczos method is more robust but less efficient for positive-definite matrices. These routines allow a choice of termination criteria and the norms used in them, allow monitoring of the approximate solution, and can return estimates of the norm of A and the largest singular value of the preconditioned matrix $\bar{A}$.

In this release no utility or black box routines specifically tailored to symmetric matrices are available. It is recommended to use the utility routines available for general sparse matrices in connection with symmetric matrices as well. The black box routine F11DCFP, however, should not be used for symmetric matrices because RGMRES usually is significantly less efficient than CG or SYMMLQ for symmetric matrices.