NAG Library Chapter Introduction

f11 – Large Scale Linear Systems

Contents

1 Scope of the Chapter

This chapter provides functions for the solution of large sparse systems of simultaneous linear equations. These include **iterative** methods for real nonsymmetric and symmetric, complex non-Hermitian and Hermitian linear systems and **direct** methods for general real linear systems. Further direct methods are currently available in Chapters f01 and f04.

2 Background to the Problems

This section is only a brief introduction to the solution of sparse linear systems. For a more detailed discussion see for example Duff *et al.* (1986) and Demmel *et al.* (1999) for direct methods, or Barrett *et al.* (1994) for iterative methods.

2.1 Sparse Matrices and Their Storage

A matrix A may be described as **sparse** if the number of zero elements is sufficiently large that it is worthwhile using algorithms which avoid computations involving zero elements.

If A is sparse, and the chosen algorithm requires the matrix coefficients to be stored, a significant saving in storage can often be made by storing only the nonzero elements. A number of different formats may be used to represent sparse matrices economically. These differ according to the amount of storage required, the amount of indirect addressing required for fundamental operations such as matrix-vector products, and their suitability for vector and/or parallel architectures. For a survey of some of these storage formats see Barrett *et al.* (1994).

Some of the functions in this chapter have been designed to be independent of the matrix storage format. This allows you to choose your own preferred format, or to avoid storing the matrix altogether. Other functions are the so-called **Black Boxes**, which are easier to use, but are based on fixed storage formats. Three fixed storage formats for sparse matrices are currently used. These are known as coordinate storage (CS) format, symmetric coordinate storage (SCS) format and compressed column storage (CCS) format.

2.1.1 Coordinate storage (CS) format

This storage format represents a sparse matrix A, with **nnz** nonzero elements, in terms of three one-dimensional arrays – a double or Complex array **a** and two Integer arrays **irow** and **icol**. These arrays are all of dimension at least **nnz**. **a** contains the nonzero elements themselves, while **irow** and **icol** store the corresponding row and column indices respectively.

For example, the matrix

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

might be represented in the arrays **a**, **irow** and **icol** as

$$\mathbf{a} = (1, 2, -1, -1, -3, -1, -4, 3, 2, 2, 4, 1, 1, -2, 1)$$

$$\mathbf{irow} = (1, 1, 1, 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5)$$

$$\mathbf{icol} = (1, 2, 3, 4, 5, 2, 5, 1, 5, 1, 3, 4, 5, 1, 5).$$

Notes

(i) The general format specifies no ordering of the array elements, but some functions may impose a specific ordering. For example, the nonzero elements may be required to be ordered by increasing row index and by increasing column index within each row, as in the example above. Utility functions are provided to order the elements appropriately (see Section 2.2).

(ii) With this storage format it is possible to enter duplicate elements. These may be interpreted in various ways (e.g., raising an error, ignoring all but the first entry, all but the last, or summing).

2.1.2 Symmetric coordinate storage (SCS) format

This storage format is suitable for symmetric and Hermitian matrices, and is identical to the CS format described in Section 2.1.1, except that only the lower triangular nonzero elements are stored. Thus, for example, the matrix

$$A = \begin{pmatrix} 4 & 1 & 0 & 0 & -1 & 2 \\ 1 & 5 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 & -1 \\ 0 & 2 & 1 & 3 & 1 & 0 \\ -1 & 0 & 0 & 1 & 4 & 0 \\ 2 & 0 & -1 & 0 & 0 & 3 \end{pmatrix}$$

might be represented in the arrays **a**, **irow** and **icol** as

$\mathbf{a} = (4, 1, 5, 2, 2, 1, 3, -1, 1, 4, 2, -1, 3).$

$\mathbf{irow} = (1, 2, 2, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6),$

$\mathbf{icol} = (1, 1, 2, 3, 2, 3, 4, 1, 4, 5, 1, 3, 6).$

2.1.3 Compressed column storage (CCS) format

This storage format also uses three one-dimensional arrays – a double or Complex array **a** and two Integer arrays **irowix** and **icolzp**. The array **a** and **irowix** are of dimension at least nnz, while **icolzp** is of dimension at least $\mathbf{n} + 1$. **a** contains the nonzero elements, going down the first column, then the second and so on. For example, the matrix in Section 2.1.1 above will be represented by

$\mathbf{a} = (1, 3, 2, -2, 2, -1, -1, 4, -1, 1, -3, -4, 2, 1, 1).$

irowix records the row index for each entry in **a**, so the same matrix will have

$\mathbf{irowix} = (1, 3, 4, 5, 1, 2, 1, 4, 1, 4, 1, 2, 3, 4, 5).$

icolzp records the index into **a** which starts each new column. The last entry of **icolzp** is equal to $nnz + 1$. An empty column (one filled with zeros, that is) is signalled by an index that is the same as the next non-empty column, or $nnz + 1$ if all subsequent columns are empty. The above example corresponds to

$\mathbf{icolzp} = (1, 5, 7, 9, 11, 16)$

The example in Section 2.1.2 above will be represented by

$\mathbf{a} = (4, 1, -1, 2, 1, 5, 2, 2, 1, -1, 2, 1, 3, 1, -1, 1, 4, 2, -1, 3)$

$\mathbf{irowix} = (1, 2, 5, 6, 1, 2, 4, 3, 4, 6, 2, 3, 4, 5, 1, 4, 5, 1, 3, 6)$

$\mathbf{icolzp} = (1, 5, 8, 11, 15, 18, 21)$

2.2 Direct Methods

Direct methods for the solution of the linear algebraic system

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

aim to determine the solution vector x in a fixed number of arithmetic operations, which is determined *a priori* by the number of unknowns. For example, an LU factorization of A followed by forward and backward substitution is a direct method for (1).

If the matrix A is sparse it is possible to design **direct** methods which exploit the sparsity pattern and are therefore much more computationally efficient than the algorithms in Chapter f07, which in general take no account of sparsity. However, if the matrix is very large and sparse, then **iterative** methods, with an appropriate preconditioner, (see Section 2.3) may be more efficient still.

This chapter provides a direct LU factorization method for sparse real systems. This method is based on special coding for supernodes, broadly defined as groups of consecutive columns with the same nonzero structure, which enables use of dense BLAS kernels. The algorithms contained here come from the SuperLU software suite (see Demmel *et al.* (1999)). An important requirement of sparse LU

factorization is keeping the factors as sparse as possible. It is well known that certain column orderings can produce much sparser factorizations than the normal left-to-right ordering. It is well worth the effort, then, to find such column orderings since they reduce both storage requirements of the factors, the time taken to compute them and the time taken to solve the linear system. The row reorderings, demanded by partial pivoting in order to keep the factorization stable, can further complicate the choice of the column ordering, but quite good and fast algorithms have been developed to make possible a fairly reliable computation of an appropriate column ordering for any sparsity pattern. We provide one such algorithm (known in the literature as COLAMD) through one function in the suite. Similar to the case for dense matrices, functions are provided to compute the LU factorization with partial row pivoting for numerical stability, solve (1) by performing the forward and backward substitutions for multiple right hand side vectors, refine the solution, minimize the backward error and estimate the forward error of the solutions, compute norms, estimate condition numbers and perform diagnostics of the factorization. It is also possible to explicitly construct, column by column, the dense inverse of the matrix by solving equation (1) for right hand sides corresponding to columns of the identity matrix. Blocks of dense columns can be handled at one time and then stored in some chosen sparse format, as system memory allows. For more details see Section 3.4.

It is also possible to use iterative method functions in this chapter to compute a direct factorization. Such methods are available for sparse real nonsymmetric, complex non-Hermitian, real symmetric positive definite and complex Hermitian positive definite systems. Further direct methods may be found in Chapters f01, f04 and f07.

2.3 Iterative Methods

In contrast to the direct methods discussed in Section 2.2, **iterative** methods for (1) approach the solution through a sequence of approximations until some user-specified termination criterion is met or until some predefined maximum number of iterations has been reached. The number of iterations required for convergence is not generally known in advance, as it depends on the accuracy required, and on the matrix A – its sparsity pattern, conditioning and eigenvalue spectrum.

Faster convergence can often be achieved using a **preconditioner** (see Golub and Van Loan (1996) and Barrett *et al.* (1994)). A preconditioner maps the original system of equations onto a different system

$$\bar{A}\bar{x} = \bar{b}, \tag{2}$$

which hopefully exhibits better convergence characteristics. For example, the condition number of the matrix $\bar{A}$ may be better than that of A, or it may have eigenvalues of greater multiplicity.

An unsuitable preconditioner or no preconditioning at all 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. Setting up a preconditioner may also involve non-negligible overheads. The application of preconditioners to real nonsymmetric, complex non-Hermitian, real symmetric and complex Hermitian and real symmetric systems of equations is further considered in Sections 2.4 and 2.5.

2.4 Iterative Methods for Real Nonsymmetric and Complex Non-Hermitian Linear Systems

Many of the most effective iterative methods for the solution of (1) lie in the class of non-stationary **Krylov subspace methods** (see Barrett *et al.* (1994)). For real nonsymmetric and complex non-Hermitian matrices this class includes:

the restarted generalized minimum residual (RGMRES) method (see Saad and Schultz (1986));

the conjugate gradient squared (CGS) method (see Sonneveld (1989));

the polynomial stabilized bi-conjugate gradient (Bi-CGSTAB(ℓ)) method (see Van der Vorst (1989) and Sleijpen and Fokkema (1993));

the transpose-free quasi-minimal residual method (TFQMR) (see Freund and Nachtigal (1991) and Freund (1993)).

Here we just give a brief overview of these algorithms as implemented in this chapter. For full details see the function documents for nag_sparse_nsym_basic_setup (f11bdc) and nag_sparse_nherm_basic_setup (f11brc).

RGMRES is based on the Arnoldi method, which explicitly generates an orthogonal basis for the Krylov subspace span$\{A^k r_0\}$, $k = 0, 1, 2, \ldots$, where r_0 is the initial residual. The solution is then expanded onto the orthogonal basis so as to minimize the residual norm. For real nonsymmetric and complex non-Hermitian matrices the generation of the basis requires a 'long' recurrence relation, resulting in prohibitive computational and storage costs. RGMRES limits these costs by restarting the Arnoldi process from the latest available residual every m iterations. The value of m is chosen in advance and is fixed throughout the computation. Unfortunately, an optimum value of m cannot easily be predicted.

CGS is a development of the bi-conjugate gradient method where the nonsymmetric Lanczos method is applied to reduce the coefficient matrix to tridiagonal form: two bi-orthogonal sequences of vectors are generated starting from the initial residual r_0 and from the *shadow residual* $\hat{r}_0$ corresponding to the arbitrary problem $A^{\mathrm{H}}\hat{x} = \hat{b}$, where $\hat{b}$ is chosen so that $r_0 = \hat{r}_0$. In the course of the iteration, the residual and shadow residual $r_i = P_i(A)r_0$ and $\hat{r}_i = P_i(A^{\mathrm{H}})\hat{r}_0$ are generated, where P_i is a polynomial of order i, and bi-orthogonality is exploited by computing the vector product $\rho_i = (\hat{r}_i, r_i) = (P_i(A^{\mathrm{H}})\hat{r}_0 P_i(A)r_0) = \left(\hat{r}_0, P_i^2(A)r_0\right)$. Applying the 'contraction' operator $P_i(A)$ twice, the iteration coefficients can still be recovered without advancing the solution of the shadow problem, which is of no interest. The CGS method often provides fast convergence; however, there is no reason why the contraction operator should also reduce the once reduced vector $P_i(A)r_0$: this can lead to a highly irregular convergence.

Bi-CGSTAB(ℓ) is similar to the CGS method. However, instead of generating the sequence $\left\{P_i^2(A)r_0\right\}$, it generates the sequence $\{Q_i(A)P_i(A)r_0\}$ where the $Q_i(A)$ are polynomials chosen to minimize the residual *after* the application of the contraction operator $P_i(A)$. Two main steps can be identified for each iteration: an OR (Orthogonal Residuals) step where a basis of order ℓ is generated by a Bi-CG iteration and an MR (Minimum Residuals) step where the residual is minimized over the basis generated, by a method similar to GMRES. For $\ell = 1$, the method corresponds to the Bi-CGSTAB method of Van der Vorst (1989). For $\ell > 1$, more information about complex eigenvalues of the iteration matrix can be taken into account, and this may lead to improved convergence and robustness. However, as ℓ increases, numerical instabilities may arise.

The transpose-free quasi-minimal residual method (TFQMR) (see Freund and Nachtigal (1991) and Freund (1993)) is conceptually derived from the CGS method. The residual is minimized over the space of the residual vectors generated by the CGS iterations under the simplifying assumption that residuals are almost orthogonal. In practice, this is not the case but theoretical analysis has proved the validity of the method. This has the effect of remedying the rather irregular convergence behaviour with wild oscillations in the residual norm that can degrade the numerical performance and robustness of the CGS method. In general, the TFQMR method can be expected to converge at least as fast as the CGS method, in terms of number of iterations, although each iteration involves a higher operation count. When the CGS method exhibits irregular convergence, the TFQMR method can produce much smoother, almost monotonic convergence curves. However, the close relationship between the CGS and TFQMR method implies that the *overall* speed of convergence is similar for both methods. In some cases, the TFQMR method may converge faster than the CGS method.

Faster convergence can usually be achieved by using a **preconditioner**. A *left* preconditioner M^{-1} can be used by the RGMRES, CGS and TFQMR methods, such that $\bar{A} = M^{-1}A \sim I_n$ in (2), where I_n is the identity matrix of order n; a *right* preconditioner M^{-1} can be used by the Bi-CGSTAB(ℓ) method, such that $\bar{A} = AM^{-1} \sim I_n$. These are formal definitions, used only in the design of the algorithms; in practice, only the means to compute the matrix-vector products $v = Au$ and $v = A^{\mathrm{H}}u$ (the latter only being required when an estimate of $\|A\|_1$ or $\|A\|_\infty$ is computed internally), and to solve the preconditioning equations $Mv = u$ are required, that is, explicit information about M, or its inverse is not required at any stage.

Preconditioning matrices M are typically based on incomplete factorizations (see Meijerink and Van der Vorst (1981)), or on the approximate inverses occurring in stationary iterative methods (see Young (1971)). A common example is the **incomplete *LU* factorization**

$$M = PLDUQ = A - R$$

where L is lower triangular with unit diagonal elements, D is diagonal, U is upper triangular with unit diagonals, P and Q are permutation matrices, and R is a remainder matrix. A **zero-fill** incomplete LU factorization is one for which the matrix

$$S = P(L + D + U)Q$$

has the same pattern of nonzero entries as A. This is obtained by discarding any **fill** elements (nonzero elements of S arising during the factorization in locations where A has zero elements). Allowing some of these fill elements to be kept rather than discarded generally increases the accuracy of the factorization at the expense of some loss of sparsity. For further details see Barrett *et al.* (1994).

2.5 Iterative Methods for Real Symmetric and Complex Hermitian Linear Systems

Three of the best known iterative methods applicable to real symmetric and complex Hermitian linear systems are the conjugate gradient (CG) method (see Hestenes and Stiefel (1952) and Golub and Van Loan (1996)) and Lanczos type methods based on SYMMLQ and MINRES (see Paige and Saunders (1975)). The description of these methods given below is for the real symmetric cases. The generalization to complex Hermitian matrices is straightforward.

For the CG method the matrix A should ideally be positive definite. The application of CG to indefinite matrices may lead to failure, or to lack of convergence. The SYMMLQ and MINRES methods are suitable for both positive definite and indefinite symmetric matrices. They are more robust than CG, but less efficient when A is positive definite.

The methods 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 three-term recurrence relations (see Golub and Van Loan (1996)). A sequence of symmetric tridiagonal matrices $\{T_k\}$ is also generated. Here and in the following, the index k denotes the iteration count. The resulting 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 a 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 symmetric tridiagonal form, $T_n = Q^{\mathrm{T}}AQ$; 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**.

The orthogonal basis is not formed explicitly in either method. The basic difference between the methods lies in the method of solution of the resulting symmetric tridiagonal systems of equations: the CG method is equivalent to carrying out an LDL^{T} (Cholesky) factorization whereas the Lanczos method (SYMMLQ) uses an LQ factorization. The MINRES method on the other hand minimizes the residual into 2-norm.

A preconditioner for these methods must be **symmetric and positive definite**, i.e., representable by $M = EE^{\mathrm{T}}$, where M is nonsingular, and such that $\bar{A} = E^{-1}AE^{-\mathrm{T}} \sim I_n$ in (2), where I_n is the identity matrix of order n. These are formal definitions, used only in the design of the algorithms; in practice, only the means to compute the matrix-vector products $v = Au$ and to solve the preconditioning equations $Mv = u$ are required.

Preconditioning matrices M are typically based on incomplete factorizations (see Meijerink and Van der Vorst (1977)), or on the approximate inverses occurring in stationary iterative methods (see Young (1971)). A common example is the **incomplete Cholesky factorization**

$$M = PLDL^{\mathrm{T}}P^{\mathrm{T}} = A - R$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements, D is diagonal and R is a remainder matrix. A **zero-fill** incomplete Cholesky factorization is one for which the matrix

$$S = P\left(L + D + L^{\mathrm{T}}\right)P^{\mathrm{T}}$$

has the same pattern of nonzero entries as A. This is obtained by discarding any **fill** elements (nonzero elements of S arising during the factorization in locations where A has zero elements). Allowing some of these fill elements to be kept rather than discarded generally increases the accuracy of the factorization at the expense of some loss of sparsity. For further details see Barrett *et al.* (1994).

3 Recommendations on Choice and Use of Available Functions

3.1 Types of Function Available

The direct method functions available in this chapter largely follow the LAPACK scheme in that four different functions separately handle the tasks of factorizing, solving, refining and condition number estimating. See Section 3.4.

The iterative method functions available in this chapter divide essentially into three types: basic functions, utility functions and Black Box functions.

Basic functions are grouped in suites of three, and implement the underlying iterative method. Each suite comprises a setup function, a solver, and a function to return additional information. The solver function 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** (see Section 2.3.2 in How to Use the NAG Library and its Documentation for further information), i.e., it returns repeatedly to the calling program with the argument **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 performed outside the solver function, thereby avoiding the need for a complicated interface with enough flexibility to cope with all types of storage schemes and sparsity patterns. This also applies to preconditioners.

(ii) Enhanced user interaction: you can closely monitor the solution 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 functions used to perform matrix operations.

At present there are suites of basic functions for real symmetric and nonsymmetric systems, and for complex Hermitian and non-Hermitian systems.

Utility functions 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 functions and utility functions therefore provide iterative methods with a considerable degree of flexibility, allowing you 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 you, but at the expense of sacrificing some flexibility in the choice of preconditioner and matrix storage format.

Black Box functions call basic and utility functions in order to provide easy-to-use functions for particular preconditioners and sparse matrix storage formats. They are much less flexible than the basic functions, 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 function 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 functions 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 functions are provided, you must call basic functions, and provide your own utility functions.

3.2 Iterative Methods for Real Nonsymmetric and Complex Non-Hermitian Linear Systems

The suite of basic functions nag_sparse_nsym_basic_setup (f11bdc), nag_sparse_nsym_basic_solver (f11bec) and nag_sparse_nsym_basic_diagnostic (f11bfc) implements either RGMRES, CGS, Bi-CGSTAB(ℓ), or TFQMR, for the iterative solution of the real sparse nonsymmetric linear system $Ax = b$. These functions 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 general, it is not possible to recommend one of these methods (RGMRES, CGS, Bi-CGSTAB(ℓ), or TFQMR) in preference to another. RGMRES is popular, but requires the most storage, and can easily stagnate when the size m of the orthogonal basis is too small, or the preconditioner is not good enough. CGS can be the fastest method, but the computed residuals can exhibit instability which may greatly affect the convergence and quality of the solution. Bi-CGSTAB(ℓ) seems robust and reliable, but it can be slower than the other methods. TFQMR can be viewed as a more robust variant of the CGS method: it shares the CGS method speed but avoids the CGS fluctuations in the residual, which may give, rise to instability. Some further discussion of the relative merits of these methods can be found in Barrett *et al.* (1994).

The utility functions provided for real nonsymmetric matrices use the coordinate storage (CS) format described in Section 2.1.1. nag_sparse_nsym_fac (f11dac) computes a preconditioning matrix based on incomplete LU factorization, and nag_sparse_nsym_precon_ilu_solve (f11dbc) solves linear systems involving the preconditioner generated by nag_sparse_nsym_fac (f11dac). The amount of fill-in occurring in the incomplete factorization can be controlled by specifying either the level of fill, or the drop tolerance. Partial or complete pivoting may optionally be employed, and the factorization can be modified to preserve row-sums.

nag_sparse_nsym_precon_bdilu (f11dfc) is a generalization of nag_sparse_nsym_fac (f11dac). It computes incomplete LU factorizations on a set of (possibly overlapping) block diagonal matrices, using a prescribed block structure, to provide a block Jacobi or additive Schwartz preconditioner. To solve the linear system defined by the preconditioner generated by nag_sparse_nsym_precon_bdilu (f11dfc), a sequence of calls to nag_sparse_nsym_precon_ilu_solve (f11dbc) (one for each block) would be required.

nag_sparse_nsym_precon_ssor_solve (f11ddc) is similar to nag_sparse_nsym_precon_ilu_solve (f11dbc), but solves linear systems involving the preconditioner corresponding to symmetric successive-over-relaxation (SSOR). The value of the relaxation parameter ω must currently be supplied by you. Automatic procedures for choosing ω will be included in the chapter at a future mark.

nag_sparse_nsym_jacobi (f11dkc) applies the iterated Jacobi method to a symmetric or nonsymmetric system of linear equations and can be used as a preconditioner. However, the domain of validity of the Jacobi method is rather restricted; you should read the function document for nag_sparse_nsym_jacobi (f11dkc) before using it.

nag_sparse_nsym_matvec (f11xac) computes matrix-vector products for real nonsymmetric matrices stored in ordered CS format. An additional utility function nag_sparse_nsym_sort (f11zac) orders the nonzero elements of a real sparse nonsymmetric matrix stored in general CS format. The same function can be used to convert a matrix from CS format to CCS format.

The Black Box function nag_sparse_nsym_fac_sol (f11dcc) makes calls to nag_sparse_nsym_basic_set up (f11bdc), nag_sparse_nsym_basic_solver (f11bec), nag_sparse_nsym_basic_diagnostic (f11bfc), nag_sparse_nsym_precon_ilu_solve (f11dbc) and nag_sparse_nsym_matvec (f11xac), to solve a real sparse nonsymmetric linear system, represented in CS format, using RGMRES, CGS, Bi-CGSTAB(ℓ), or TFQMR, with incomplete LU preconditioning. nag_sparse_nsym_sol (f11dec) is similar, but has options for no preconditioning, Jacobi preconditioning or SSOR preconditioning. nag_sparse_nsym_ precon_bdilu_solve (f11dgc) is also similar to nag_sparse_nsym_fac_sol (f11dcc), but uses block Jacobi or additive Schwartz preconditioning.

For complex non-Hermitian sparse matrices there is an equivalent suite of functions. nag_sparse_n herm_basic_setup (f11brc), nag_sparse_nherm_basic_solver (f11bsc) and nag_sparse_nherm_basic_diag nostic (f11btc) are the basic functions which implement the same methods used for real nonsymmetric systems, namely RGMRES, CGS, Bi-CGSTAB(ℓ) and TFQMR, for the solution of complex sparse non-

Hermitian linear systems. nag_sparse_nherm_fac (f11dnc) and nag_sparse_nherm_precon_ilu_solve (f11dpc) are the complex equivalents of nag_sparse_nsym_fac (f11dac) and nag_sparse_nsym_preco n_ilu_solve (f11dbc), respectively, providing facilities for implementing ILU preconditioning. nag_sparse_nherm_precon_ssor_solve (f11drc) and nag_sparse_nherm_precon_bdilu (f11dtc) implement complex versions of the SSOR and block Jacobi (or additive Schwartz) preconditioners, respectively. nag_sparse_nherm_jacobi (f11dxc) implements a complex version of the iterated Jacobi preconditioner. Utility functions nag_sparse_nherm_matvec (f11xnc) and nag_sparse_nherm_sort (f11znc) are provided for computing matrix-vector products and sorting the elements of complex sparse non-Hermitian matrices, respectively. Finally, the Black Box functions nag_sparse_nherm_fac_sol (f11dqc), nag_sparse_nherm_sol (f11dsc) and nag_sparse_nherm_precon_bdilu_solve (f11duc) are complex equivalents of nag_sparse_nsym_fac_sol (f11dcc), nag_sparse_nsym_sol (f11dec) and nag_sparse_n sym_precon_bdilu (f11dfc), respectively.

3.3 Iterative Methods for Real Symmetric and Complex Hermitian Linear Systems

The suite of basic functions nag_sparse_sym_basic_setup (f11gdc), nag_sparse_sym_basic_solver (f11gec) and nag_sparse_sym_basic_diagnostic (f11gfc) implement either the conjugate gradient (CG) method, or a Lanczos method based on SYMMLQ, for the iterative solution of the real 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 functions 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 utility functions provided for real symmetric matrices use the symmetric coordinate storage (SCS) format described in Section 2.1.2. nag_sparse_sym_chol_fac (f11jac) computes a preconditioning matrix based on incomplete Cholesky factorization, and nag_sparse_sym_precon_ichol_solve (f11jbc) solves linear systems involving the preconditioner generated by nag_sparse_sym_chol_fac (f11jac). The amount of fill-in occurring in the incomplete factorization can be controlled by specifying either the level of fill, or the drop tolerance. Diagonal Markowitz pivoting may optionally be employed, and the factorization can be modified to preserve row-sums. Additionally, the utility function nag_sparse_ sym_rcm (f11yec) can be used to discover a row and column permutation that reduces the bandwidth of A.

nag_sparse_sym_precon_ssor_solve (f11jdc) is similar to nag_sparse_sym_precon_ichol_solve (f11jbc), but solves linear systems involving the preconditioner corresponding to symmetric successive-over-relaxation (SSOR). The value of the relaxation parameter ω must currently be supplied by you. Automatic procedures for choosing ω will be included in the chapter at a future mark.

nag_sparse_nsym_jacobi (f11dkc) applies the iterated Jacobi method to a symmetric or nonsymmetric system of linear equations and can be used as a preconditioner. However, the domain of validity of the Jacobi method is rather restricted; you should read the function document for nag_sparse_nsym_jacobi (f11dkc) before using it.

nag_sparse_sym_matvec (f11xec) computes matrix-vector products for real symmetric matrices stored in ordered SCS format. An additional utility function nag_sparse_sym_sort (f11zbc) orders the nonzero elements of a real sparse symmetric matrix stored in general SCS format.

The Black Box function nag_sparse_sym_chol_sol (f11jcc) makes calls to nag_sparse_sym_basic_setup (f11gdc), nag_sparse_sym_basic_solver (f11gec), nag_sparse_sym_basic_diagnostic (f11gfc), nag_spar se_sym_precon_ichol_solve (f11jbc) and nag_sparse_sym_matvec (f11xec), to solve a real sparse symmetric linear system, represented in SCS format, using a conjugate gradient or Lanczos method, with incomplete Cholesky preconditioning. nag_sparse_sym_sol (f11jec) is similar, but has options for no preconditioning, Jacobi preconditioning or SSOR preconditioning.

For complex Hermitian sparse matrices there is an equivalent suite of functions. nag_sparse_herm_ba sic_setup (f11grc), nag_sparse_herm_basic_solver (f11gsc) and nag_sparse_herm_basic_diagnostic (f11gtc) are the basic functions which implement the same methods used for real symmetric systems, namely CG and SYMMLQ, for the solution of complex sparse Hermitian linear systems. nag_sparse_herm_chol_fac (f11jnc) and nag_sparse_herm_precon_ichol_solve (f11jpc) are the complex equivalents of nag_sparse_sym_chol_fac (f11jac) and nag_sparse_sym_precon_ichol_solve (f11jbc), respectively, providing facilities for implementing incomplete Cholesky preconditioning. nag_sparse_

herm_precon_ssor_solve (f11jrc) implements a complex version of the SSOR preconditioner. nag_sparse_nherm_jacobi (f11dxc) implements a complex version of the iterated Jacobi preconditioner. Utility functions nag_sparse_herm_matvec (f11xsc) and nag_sparse_herm_sort (f11zpc) are provided for computing matrix-vector products and sorting the elements of complex sparse Hermitian matrices, respectively. Finally, the Black Box functions nag_sparse_herm_chol_sol (f11jqc) and nag_sparse_herm_sol (f11jsc) provide easy-to-use implementations of the CG and SYMMLQ methods for complex Hermitian linear systems.

3.4 Direct Methods

The suite of functions nag_superlu_column_permutation (f11mdc), nag_superlu_lu_factorize (f11mec), nag_superlu_solve_lu (f11mfc), nag_superlu_condition_number_lu (f11mgc), nag_superlu_refine_lu (f11mhc), nag_superlu_matrix_product (f11mkc), nag_superlu_matrix_norm (f11mlc) and nag_superlu_diagnostic_lu (f11mmc) implement the COLAMD/SuperLU direct real sparse solver and associated utilities. You are expected to first call nag_superlu_column_permutation (f11mdc) to compute a suitable column permutation for the subsequent factorization by nag_superlu_lu_factorize (f11mec). nag_superlu_solve_lu (f11mfc) then solves the system of equations. A solution can be further refined by nag_superlu_refine_lu (f11mhc), which also minimizes the backward error and estimates a bound for the forward error in the solution. Diagnostics are provided by nag_superlu_condition_number_lu (f11mgc) which computes an estimate of the condition number of the matrix using the factorization output by nag_superlu_lu_factorize (f11mec), and nag_superlu_diagnostic_lu (f11mmc) which computes the reciprocal pivot growth (a numerical stability measure) of the factorization. The two utility functions, nag_superlu_matrix_product (f11mkc), which computes matrix-matrix products in the particular storage scheme demanded by the suite (CCS format), and nag_superlu_matrix_norm (f11mlc) which computes quantities relating to norms of a matrix in that particular storage scheme, complete the suite.

Another way of computing a direct solution is to choose specific arguments for the indirect solvers. For example, function nag_sparse_nsym_precon_ilu_solve (f11dbc) solves a linear system involving the incomplete LU preconditioning matrix

$$M = PLDUQ = A - R$$

generated by nag_sparse_nsym_fac (f11dac), where P and Q are permutation matrices, L is lower triangular with unit diagonal elements, U is upper triangular with unit diagonal elements, D is diagonal and R is a remainder matrix.

If A is nonsingular, a call to nag_sparse_nsym_fac (f11dac) with **lfill** < 0 and **dtol** $= 0.0$ results in a zero remainder matrix R and a **complete** factorization. A subsequent call to nag_sparse_nsym_precon_ilu_solve (f11dbc) will therefore result in a direct method for real sparse nonsymmetric systems.

If A is known to be symmetric positive definite, nag_sparse_sym_chol_fac (f11jac) and nag_sparse_sym_precon_ichol_solve (f11jbc) may similarly be used to give a direct solution. For further details see Section 9.4 in nag_sparse_sym_chol_fac (f11jac).

Complex non-Hermitian systems can be solved directly in the same way using nag_sparse_nherm_fac (f11dnc) and nag_sparse_nherm_precon_ilu_solve (f11dpc), while for complex Hermitian systems nag_sparse_herm_chol_fac (f11jnc) and nag_sparse_herm_precon_ichol_solve (f11jpc) may be used.

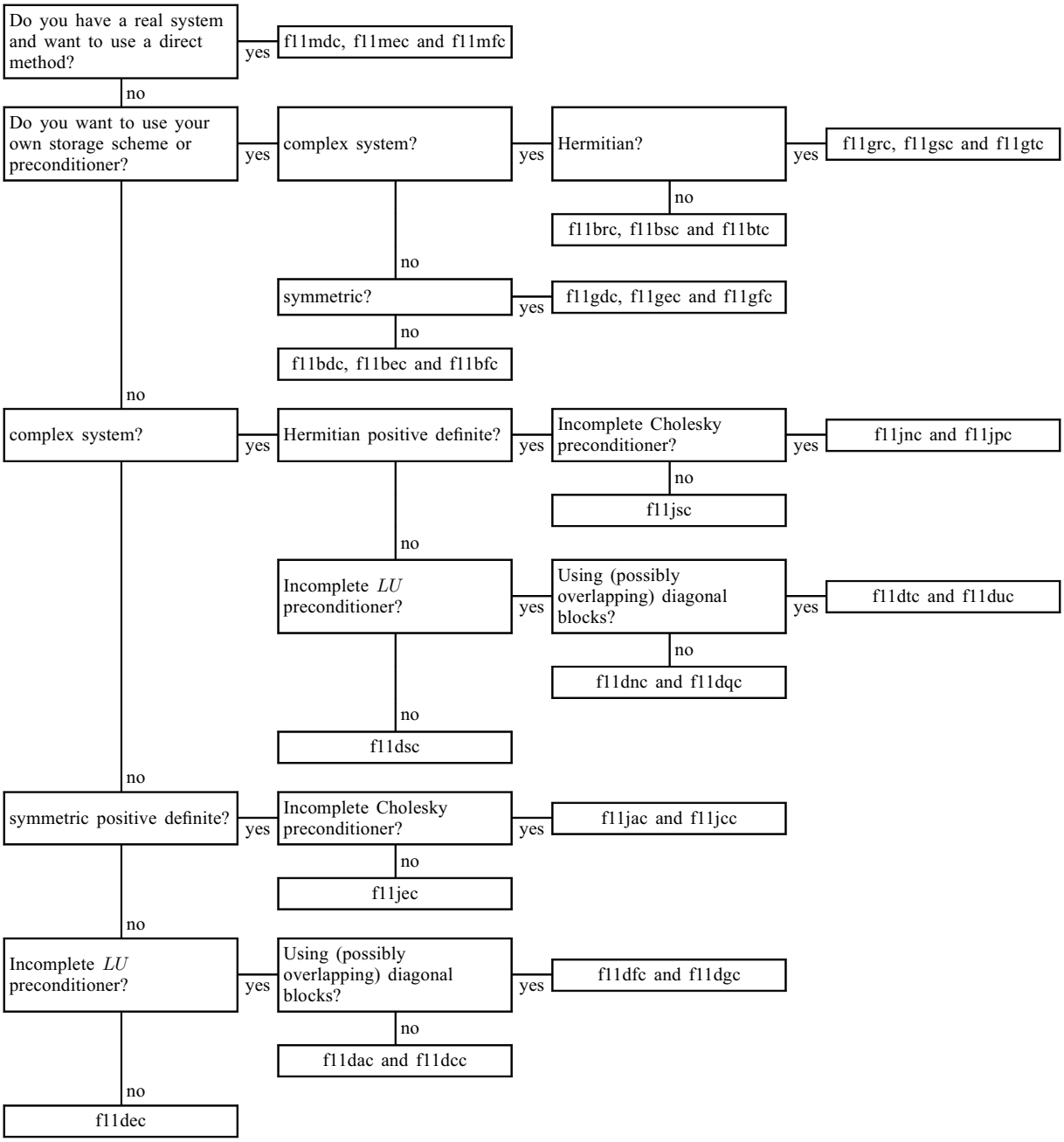
Some other functions specifically designed for direct solution of sparse linear systems can currently be found in Chapters f01, f04 and f07. In particular, the following functions allow the direct solution of symmetric positive definite systems:

Variable band (skyline)	nag_real_cholesky_skyline (f01mcc) and nag_real_cholesky_skyline_solve (f04mcc)

Functions for the solution of band and tridiagonal systems can be found in Chapters f04 and f07.

4 Decision Tree

Tree 1: Solvers



5 Functionality Index

Basic functions for real nonsymmetric linear systems,
- diagnostic function ..... nag_sparse_nsym_basic_diagnostic (f11bfc)
- reverse communication RGMRES, CGS, Bi-CGSTAB(ℓ) or TFQMR solver function ..... nag_sparse_nsym_basic_solver (f11bec)
- setup function ..... nag_sparse_nsym_basic_setup (f11bdc)

Basic functions for real symmetric linear systems,
- diagnostic function ..... nag_sparse_sym_basic_diagnostic (f11gfc)
- reverse communication CG or SYMMLQ solver ..... nag_sparse_sym_basic_solver (f11gec)
- setup function ..... nag_sparse_sym_basic_setup (f11gdc)

Black Box functions for complex Hermitian linear systems,
- CG or SYMMLQ solver,
  - with incomplete Cholesky preconditioning ..... nag_sparse_herm_chol_sol (f11jqc)
  - with no preconditioning, Jacobi or SSOR preconditioning ..... nag_sparse_herm_sol (f11jsc)

Black Box functions for complex non-Hermitian linear systems,
- RGMRES, CGS, Bi-CGSTAB(ℓ) or TFQMR solver,
  - with block Jacobi or additive Schwarz preconditioning ..... nag_sparse_nherm_precon_bdilu_solve (f11duc)
  - with incomplete LU preconditioning ..... nag_sparse_nherm_fac_sol (f11dqc)
  - with no preconditioning, Jacobi, or SSOR preconditioning ..... nag_sparse_nherm_sol (f11dsc)

Black Box functions for real nonsymmetric linear systems,
- RGMRES, CGS, Bi-CGSTAB(ℓ) or TFQMR solver,
  - with block Jacobi or additive Schwarz preconditioning ..... nag_sparse_nsym_precon_bdilu_solve (f11dgc)
  - with incomplete LU preconditioning ..... nag_sparse_nsym_fac_sol (f11dcc)
  - with no preconditioning, Jacobi, or SSOR preconditioning ..... nag_sparse_nsym_sol (f11dec)

Black Box functions for real symmetric linear systems,
- CG or SYMMLQ solver,
  - with incomplete Cholesky preconditioning ..... nag_sparse_sym_chol_sol (f11jcc)
  - with no preconditioning, Jacobi, or SSOR preconditioning ..... nag_sparse_sym_sol (f11jec)

Direct methods for real sparse nonsymmetric linear systems in CCS format,
- apply iterative refinement to the solution and compute error estimates, after factorizing the matrix of coefficients ..... nag_superlu_refine_lu (f11mhc)
- condition number estimation, after factorizing the matrix of coefficients, ... nag_superlu_condition_number_lu (f11mgc)
- LU factorization,
  - diagnostic ..... nag_superlu_diagnostic_lu (f11mmc)
  - factorize ..... nag_superlu_lu_factorize (f11mec)
  - setup ..... nag_superlu_column_permutation (f11mdc)
- solution of simultaneous linear equations, after factorizing the matrix of coefficients, ... nag_superlu_solve_lu (f11mfc)
- utility,
  - compute a norm or the element of largest absolute value, ... nag_superlu_matrix_norm (f11mlc)
  - matrix-matrix multiplier ..... nag_superlu_matrix_product (f11mkc)

Utility function for complex Hermitian linear systems,
- incomplete Cholesky factorization ..... nag_sparse_herm_chol_fac (f11jnc)
- matrix-vector multiplier for complex Hermitian matrices in SCS format ..... nag_sparse_herm_matvec (f11xsc)
- solver for linear systems involving preconditioning matrix from nag_sparse_herm_chol_fac (f11jnc) ..... nag_sparse_herm_precon_ichol_solve (f11jpc)
- solver for linear systems involving SSOR preconditioning matrix ..... nag_sparse_herm_precon_ssor_solve (f11jrc)
- sort function for complex Hermitian matrices in SCS format ..... nag_sparse_herm_sort (f11zpc)

Utility function for complex linear systems,
- solver for linear systems involving iterated Jacobi method ..... nag_sparse_nherm_jacobi (f11dxc)

Utility function for complex non-Hermitian linear systems,
incomplete *LU* factorization ..... nag_sparse_nherm_fac (f11dnc)
incomplete *LU* factorization of local or overlapping diagonal blocks
..... nag_sparse_nherm_precon_bdilu (f11dtc)
matrix-vector multiplier for complex non-Hermitian matrices in CS format
..... nag_sparse_nherm_matvec (f11xnc)
solver for linear systems involving preconditioning matrix from nag_sparse_nherm_fac (f11dnc)
..... nag_sparse_nherm_precon_ilu_solve (f11dpc)
solver for linear systems involving SSOR preconditioning matrix
..... nag_sparse_nherm_precon_ssor_solve (f11drc)
sort function for complex non-Hermitian matrices in CS format ..... nag_sparse_nherm_sort (f11znc)

Utility function for real linear systems,
solver for linear systems involving iterated Jacobi method ..... nag_sparse_nsym_jacobi (f11dkc)

Utility function for real nonsymmetric linear systems,
incomplete *LU* factorization ..... nag_sparse_nsym_fac (f11dac)
incomplete *LU* factorization of local or overlapping diagonal blocks
..... nag_sparse_nsym_precon_bdilu (f11dfc)
matrix-vector multiplier for real nonsymmetric matrices in CS format
..... nag_sparse_nsym_matvec (f11xac)
solver for linear systems involving preconditioning matrix from nag_sparse_nsym_fac (f11dac)
..... nag_sparse_nsym_precon_ilu_solve (f11dbc)
solver for linear systems involving SSOR preconditioning matrix
..... nag_sparse_nsym_precon_ssor_solve (f11ddc)
sort function for real nonsymmetric matrices in CS format ..... nag_sparse_nsym_sort (f11zac)

Utility function for real symmetric linear systems,
incomplete Cholesky factorization ..... nag_sparse_sym_chol_fac (f11jac)
matrix-vector multiplier for real symmetric matrices in SCS format
..... nag_sparse_sym_matvec (f11xec)
solver for linear systems involving preconditioning matrix from nag_sparse_sym_chol_fac (f11jac)
..... nag_sparse_sym_precon_ichol_solve (f11jbc)
solver for linear systems involving SSOR preconditioning matrix
..... nag_sparse_sym_precon_ssor_solve (f11jdc)
sort function for real symmetric matrices in SCS format ..... nag_sparse_sym_sort (f11zbc)

Utility function for real symmetric linear systems, compute bandwidth-reducing reverse Cuthill–McKee permutation ..... nag_sparse_sym_rcm (f11yec)

6 Auxiliary Functions Associated with Library Function Arguments

None.

7 Functions Withdrawn or Scheduled for Withdrawal

None.

8 References

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

Demmel J W, Eisenstat S C, Gilbert J R, Li X S and Li J W H (1999) A supernodal approach to sparse partial pivoting *SIAM J. Matrix Anal. Appl.* **20** 720–755

Duff I S, Erisman A M and Reid J K (1986) *Direct Methods for Sparse Matrices* Oxford University Press, London

Freund R W (1993) A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems *SIAM J. Sci. Comput.* **14** 470–482

Freund R W and Nachtigal N (1991) QMR: a Quasi-Minimal Residual Method for Non-Hermitian Linear Systems *Numer. Math.* **60** 315–339

Golub G H and Van Loan C F (1996) *Matrix Computations* (3rd Edition) Johns Hopkins University Press, Baltimore

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

Meijerink J and Van der Vorst H (1977) An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix *Math. Comput.* **31** 148–162

Meijerink J and Van der Vorst H (1981) Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems *J. Comput. Phys.* **44** 134–155

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

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

Sleijpen G L G and Fokkema D R (1993) BiCGSTAB(ℓ) for linear equations involving matrices with complex spectrum *ETNA* **1** 11–32

Sonneveld P (1989) CGS, a fast Lanczos-type solver for nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **10** 36–52

Van der Vorst H (1989) Bi-CGSTAB, a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **13** 631–644

Young D (1971) *Iterative Solution of Large Linear Systems* Academic Press, New York