Sparse Direct Methods: An Introduction

J. A. Scott

Department for Computation and Information, Rutherford Appleton Laboratory, Chilton, Didcot, Oxon OX11 0RA, England. (J.Scott@rl.ac.uk)

Abstract. The solution of large-scale linear systems lies at the heart of many computations in science, engineering, industry, and (more recently) finance. In this paper, we give a brief introduction to direct methods based on Gaussian elimination for the solution of such systems. We discuss the methods with reference to the sparse direct solvers that are available in the Harwell Subroutine Library. We briefly consider large sparse eigenvalue problems and show how the efficient solution of such problems depends upon the efficient solution of sparse linear systems.

1 Introduction

Sparse matrices arise in very many application areas, including such diverse fields as structural analysis, chemical engineering, surveying, and economics. A matrix is sparse if many of its coefficients are zero and there is an advantage in exploiting the zeros. In this paper, we present a brief introduction to direct methods for the solution of large sparse linear systems of equations

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

We are concerned with methods that are based on Gaussian elimination. That is, we compute an LU factorization of a permutation of A

$$PAQ = LU,$$

where P and Q are permutation matrices and L and U are lower and upper triangular matrices, respectively. These factors are used to solve the system (1) through the forward elimination

$$Ly = Pb$$

followed by the back substitution

$$Uz = y.$$

The required solution x is then the permuted vector $x = Qz$. When A is symmetric positive definite, it is normal to use the Cholesky factorization

$$PAP^T = LL^T.$$

For more general symmetric matrices, the factorization

$$PAP^T = LDL^T,$$

is more appropriate. For a stable decomposition in the indefinite case, the matrix D is block diagonal with blocks of order 1 or 2, and L is unit lower triangular.

Several different approaches to Gaussian elimination for sparse matrices have been developed. They can each be divided into a number of phases:

1. Preordering to exploit structure eg preordering to block triangular form

$$PAQ = \begin{pmatrix} B_{11} & & & & \\ B_{21} & B_{22} & & & \\ B_{31} & B_{32} & B_{33} & & \\ . & . & . & & \\ B_{N1} & B_{N2} & B_{N3} & \dots & B_{NN} \end{pmatrix}$$

   so that only the diagonal blocks B_{ii} need to be factorized.
2. Analyse - the sparsity pattern of A is analyzed to produce a suitable ordering and data structures for efficient factorization.
3. Factorize - the numerical factorization of A is performed.
4. Solve - the factors are used to solve one or more systems (1) using forward elimination and back substitution.

Some codes combine the analyse and factorize phases so that numerical values are available when the ordering is being generated. Phase 3 (or the combined phase 2 and 3) generally requires most computational time. If more than one matrix with the same sparsity pattern is to be factorized, the analyse phase only needs to be performed once. Thus in contrast to dense solvers, for sparse solvers, it is potentially much faster to perform subsequent factorizations.

In the following sections, we introduce three approaches: the general approach, frontal methods, and multifrontal methods. We illustrate these different approaches using software from the Harwell Subroutine Library (HSL) [1] and use numerical examples from a range of scientific and industrial applications. A useful reference is the book by Duff, Erisman, and Reid [2] and for an extensive list of references to recent developments in the area, we recommend the review by Duff [3].

We end this section by noting that the order of a matrix that is considered large is a function of time depending on both the development of dense and sparse codes and advances in computer architecture. However, by today's standards, A need not be very large for it to be worthwhile to exploit sparsity. This is illustrated in Table 1 (taken from Duff [4]), in which we compare the performance of the HSL sparse solver `MA48` with that of the dense solver `SGESV` from LAPACK. The problems are all from the Harwell-Boeing Sparse Matrix Collection [5]. The experiments were performed on a single processor of a CRAY Y-MP vector supercomputer and the timings are given in seconds.

Table 1. A comparison of timings on the CRAY Y-MP for `MA48` and `SGESV` on Harwell-Boeing matrices.

Identifier	Order	Number of entries	`MA48`	`SGESV`
FS 680 3	680	2646	0.06	0.96
PORES 2	1224	9613	0.54	4.54
BCSSTK27	1224	56126	2.07	4.55
NNC1374	1374	8606	0.70	6.19
WEST2021	2021	7353	0.21	18.88
ORANI678	2529	90158	1.17	36.37

2 The General Approach

The principal features of the general approach are:

- sparse data structures are used throughout
- numerical and sparsity pivoting are performed at the same time (phases 2 and 3 are combined).

The efficient implementation of techniques for handling sparse data structures is of crucial importance. The most common sparse data structure and the one used in most general-purpose codes holds the matrix by rows. All rows are stored in the same way with the real values and column indices in two arrays with a one-to-one correspondence between the arrays, so that the real value in position k, say, is in the column indicated by the entry in position k of the column index array. A sparse matrix can then be stored as a collection of sparse rows in two arrays; one integer, the other real, both of length nz, where nz is the number of entries in A. A third "pointer" array of length $n+1$ (n is the order of A) is used to identify the position in the first two arrays of the data structure for each row.

To illustrate this scheme, consider the following 4×4 sparse matrix:

$$A = \begin{pmatrix} 6 & 0 & 1 & 0 \\ 2 & 4 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 7 & 0 & 0 & -3 \end{pmatrix}$$

Storing this matrix as a collection of sparse row vectors we have

column index	1	3	2	4	1	2	1	4
value	6.	1.	4.	-1.	2.	1.	7.	-3.
row pointer	1	3	6	7	9			

Note that the column indices within each row need not be held in order. The advantages of the scheme are that it is a simple and compact method of storing the matrix and it is straightforward to access the matrix by rows. A disadvantage is that it is difficult to insert entries, which is needed in Gaussian elimination when a multiple of one row (the pivot row) of the matrix is added to the other (non-pivot) rows with different sparsity patterns.

In general, the matrix factors L and U are denser than A. For efficiency, in terms of both storage and floating-point operations (flops), it is essential to try and restrict the amount of "fill-in" (that is, the number of entries in L and U that correspond to zeros in A). The rows and columns of A need to be ordered to preserve sparsity. For example, consider the symmetric matrix with sparsity pattern

$$A = \begin{pmatrix} x & x & x & x & x \\ x & x & & & \\ x & & x & & \\ x & & & x & \\ x & & & & x \end{pmatrix}.$$

Choosing the pivots in order down the diagonal, the Cholesky factorization of A is LL^T, where L has the form

$$L = \begin{pmatrix} x & & & & \\ x & x & & & \\ x & x & x & & \\ x & x & x & x & \\ x & x & x & x & x \end{pmatrix}.$$

Thus all sparsity has been lost. However, reordering the rows of A in reverse order

$$PA = \hat{A} = \begin{pmatrix} x & & & & x \\ & x & & & x \\ & & x & & x \\ & & & x & x \\ x & x & x & x & x \end{pmatrix},$$

and $\hat{L}$ retains the sparsity of $\hat{A}$

$$\hat{L} = \begin{pmatrix} x & & & & \\ & x & & & \\ & & x & & \\ & & & x & \\ x & x & x & x & x \end{pmatrix}.$$

A simple but effective strategy for maintaining sparsity is due to Markowitz [6]. We motivate this strategy by considering the first step of Gaussian elimination for the matrix A partitioned in the form

$$A = A^{(0)} = \begin{pmatrix} \alpha & a_r^T \\ a_c & A_R^{(0)} \end{pmatrix}.$$

Assuming α is a suitable pivot choice, the first step in the matrix factorization is

$$A = \begin{pmatrix} 1 & 0 \\ a_c\alpha^{-1} & A^{(1)} \end{pmatrix} \begin{pmatrix} \alpha & a_r^T \\ 0 & I \end{pmatrix},$$

where

$$A^{(1)} = A_R^{(0)} - \frac{a_c a_r^T}{\alpha}. \tag{2}$$

Fill-in occurs when an entry in the rank-one matrix $a_c a_r^T$ is a zero in $A_R^{(0)}$. Clearly the dominant cost in the update (2) is that of forming the outer product $a_c a_r^T$. This cost is proportional to the product of the number of nonzeros in a_c and the number of nonzeros in a_r^T. At each stage of Gaussian elimination, Markowitz therefore selects as the pivot the nonzero entry of the remaining reduced submatrix that minimizes this product. That is, $a_{ij}^{(k)}$ is chosen as the pivot for the k-th stage to minimize

$$(r_i^{(k)} - 1)(c_j^{(k)} - 1),$$

where $r_i^{(k)}$ and $c_j^{(k)}$ denote, respectively, the number of nonzeros in row i and column j of $A^{(k)} = \{a_{ij}^{(k)}\}$.

For stability, pivot candidates must also satisfy some numerical criteria. In particular, assuming row and column interchanges have been performed to bring the pivot candidate selected by the Markowitz criteria to the diagonal, the pivot is only acceptable if it satisfies the inequality

$$|a_{kk}^{(k)}| \geq u\, |a_{ik}^{(k)}|, \quad i \geq k,$$

where u is a preset threshold parameter in the range $0 < u \leq 1$. The choice $u = 1$ corresponds to partial pivoting but, in general, this value is too restrictive and leads to a large number of pivots being rejected and to unnecessary fill-in. If u is chosen to be too small, instability can result. A common choice is $u = 0.1$. Experience has shown that this value usually provides a good compromise between maintaining stability and preserving sparsity.

In the Harwell Subroutine Library, the package `MA48` of Duff and Reid [7] (and its complex counterpart `ME48`) is a general sparse solver which uses Markowitz pivoting. It uses the numerical values in the analyse phase and its default value for the threshold parameter is 0.1. The code offers a "fast factorization" for matrices with exactly the same sparsity pattern as one that has already been factorized. One of the ways in which the code achieves high performance, particularly on vector or super scalar machines, is by switching to full-matrix processing and using Level 3 BLAS [8] once the matrix is sufficiently dense. `MA48` is frequently used as a benchmark against which new sparse solvers are judged.

The strength of the general approach is that it gives satisfactory performance for many matrix structures and is often the method of choice for very sparse unstructured problems. Some gains and simplifications are possible if A is symmetric. In particular, the Markowitz ordering is replaced by minimum degree ordering. At the kth stage the pivot is chosen to be $a_{ii}^{(k)}$, where

$$r_i^{(k)} = \min_l r_l^{(k)},$$

and $r_l^{(k)}$ is the number of nonzero entries in row l of $A^{(k)}$. If the matrix is additionally positive definite, numerical pivoting is not needed.

3 Frontal Methods

Frontal methods have their origins in the solution of finite-element problems for structural analysis, in which the matrix is symmetric and positive definite. The method can, however, be extended to general unsymmetric systems and need not be restricted to finite-element applications (see Duff [9]).

To describe the method, we assume that A is a sum of finite-element matrices

$$A = \sum_{l=1}^{m} A^{[l]} \tag{3}$$

where each element matrix $A^{[l]}$ has nonzeros only in a few rows and columns and corresponds to the matrix from element l. The main feature of the frontal method is that the contributions $A^{[l]}$ are assembled one at a time and the construction of the assembled coefficient matrix A is avoided by interleaving assembly and elimination operations. An assembly operation is of the form

$$a_{ij} \Leftarrow a_{ij} + a_{ij}^{[l]}, \tag{4}$$

where $a_{ij}^{[l]}$ is the (i,j)th nonzero entry of the element matrix $A^{[l]}$. A variable is *fully summed* if it is involved in no further sums of the form (4) and is *partially summed* if it has appeared in at least one of the elements assembled so far but is not yet fully summed. The Gaussian elimination operation

$$a_{ij} \Leftarrow a_{ij} - a_{il}[a_{ll}]^{-1}a_{lj} \tag{5}$$

may be performed as soon as all the terms in the triple product in (5) are fully summed. At any stage during the assembly and elimination processes, the fully and partially summed variables are held in a dense matrix, termed the *frontal* matrix. Assuming k variables are fully summed, the frontal matrix F can be partitioned in the form

$$F = \begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix},$$

where F_{11} is a square matrix of order k. Pivots may be chosen from anywhere in F_{11}. For symmetric positive-definite systems, they can be taken from the diagonal in order but in the unsymmetric case, pivots must be chosen to satisfy a threshold criteria. Assuming k pivots can be chosen, F_{11} is factorized as $L_{11}U_{11}$. Then F_{12} and F_{21} are updated as

$$\hat{F}_{21} = F_{21}U_{11}^{-1} \quad \text{and} \quad \hat{F}_{12} = L_{11}^{-1}F_{12}$$

and finally the Schur complement

$$\hat{F}_{22} = F_{22} - \hat{F}_{21}\hat{F}_{12}$$

is formed. The factors L_{11} and U_{11}, as well as $\hat{F}_{12}$ and $\hat{F}_{21}$, are stored as parts of L and U, before further elements are assembled with the Schur complement to form a new frontal matrix.

The power of frontal schemes comes from the following observations:

- since the frontal matrix is held as a dense matrix, dense linear algebra kernels (in particular, the BLAS) can be used during the numerical factorization,
- the matrix factors need not be held in main memory, which allows large problems to be solved using only modest amounts of high-speed memory.

There are a number of frontal codes in the Harwell Subroutine Library: `MA42` (Duff and Scott [10]) and its complex counterpart `ME42` are for general unsymmetric systems and `MA62` (Duff and Scott [11]) is for symmetric positive definite finite-element problems. Both codes use Level 3 BLAS in the innermost loop and optionally store the matrix factors in auxiliary storage. High level BLAS are also used in the solve phase, and this increases the efficiency when the codes are used to solve for multiple right-hand sides.

On a machine with fast Level 3 BLAS, the performance of the Harwell frontal solvers can be impressive. This is illustrated in Table 2. Here `MA42` is used to solve a standard finite-element test problem on a single processor of a CRAY

Table 2. Performance (in Mflop/s) of `MA42` on a standard test problem running on a CRAY Y-MP.

Dimension of element grid	16×16	32×32	48×48	64×64	96×96
Max order frontal matrix	195	355	515	675	995
Total order of problem	5445	21125	47045	83205	186245
Mflop/s	145	208	242	256	272

Y-MP, whose peak performance is 333 Mflop/s and on which the Level 3 BLAS matrix-matrix multiply routine `SGEMM` runs at 313 Mflop/s on sufficiently large matrices. It is important to realize that the Mflop rates given in Table 2 include all overheads for holding the factors in auxiliary storage.

The performance of frontal methods, in terms of both the number of floating-point operations and the storage requirements, is dependent upon the order in which the elements are assembled. Many of the proposed algorithms for element ordering are similar to those for profile reduction of assembled matrices (see, for example, Reid and Scott [12]). Routine `MC63` by Scott [13] is a new element ordering routine in the Harwell Subroutine Library.

For non-element problems, the frontal method proceeds by assembling the rows of the matrix one at a time. In this case, efficiency depends on the ordering of the rows. For matrices with a symmetric or almost symmetric sparsity pattern, a profile reduction algorithm using the sparsity pattern of the Boolean sum of the patterns of A and A^T can be used. For highly unsymmetric problems, new algorithms for row ordering have recently been introduced by Scott [14], and implemented as routine `MC62` in the Harwell Subroutine Library.

4 Multifrontal Methods

Although high Megaflop rates can be achieved by frontal solvers, there are several important deficiences with the method:

- for some problems, many more flops are performed than are needed by other methods
- the factors can be denser than those produced by other methods
- there is little scope for parallelism other than that which can be obtained within the higher level BLAS.

These problems can be at least partially overcome through the use of more than one front.

The multiple front approach partitions the underlying "domain" into subdomains, performs a frontal decomposition on each subdomain separately and then factorizes the remaining "interface" variables, perhaps by also using a frontal scheme. The strategy corresponds to a bordered block diagonal ordering of the matrix and can be nested. With judicious ordering within each subproblem, the amount of work required can be reduced and, since the factorizations of the subproblems are independent, there is much scope for parallelism. Preliminary results presented by Duff and Scott [15] are encouraging.

Multifrontal methods are a further extension of the frontal method. In place of a small number of frontal matrices corresponding to the number of subdomains used, many frontal matrices are used. Each is used for one or more pivot steps, and the resulting Schur complement is summed with other Schur complements to generate a further frontal matrix. To illustrate this idea, assume A is a sum of finite-element matrices (3). Assuming the elements are assembled in the natural order 1,2,..., the frontal method uses the summation

$$((..((\mathbf{A}^{[1]} + \mathbf{A}^{[2]}) + \mathbf{A}^{[3]}) + \mathbf{A}^{[4]}) + ...).$$

However, there are other ways in which the summation can be performed. One possible alternative is to sum the elements in pairs, and then sum the pairs in pairs, and so on

$$((\mathbf{A}^{[1]} + \mathbf{A}^{[2]}) + (\mathbf{A}^{[3]} + \mathbf{A}^{[4]})) + ((\mathbf{A}^{[5]} + \mathbf{A}^{[6]}) + (\mathbf{A}^{[7]} + \mathbf{A}^{[8]})) + ...$$

A judicious ordering of the summation can reduce the work involved in the factorization and the density of the resulting factors. This added freedom in the way in which the assemblies are organized gives the multifrontal method an advantage over frontal methods. By using a sparsity ordering technique for symmetric systems (usually based on minimum degree), the method can be used efficiently for any matrix whose sparsity pattern is symmetric or almost symmetric. The restriction to nearly symmetric patterns arises because the initial ordering is performed using the sparsity pattern of $A + A^T$. The approach can, however, be used on any system. If the matrix is very unsymmetric (that is for many entries $a_{ij} \neq 0$ but $a_{ji} = 0$), numerical pivoting in the factorization phase can significantly perturb the ordering given by the analyse phase. This is much reduced if the matrix is permuted to have a zero-free diagonal before the analyse phase and further gains can sometimes be obtained by permuting entries with large modulus to the diagonal. For further details, see Duff and Koster [16]. This enables the multifrontal method to perform well on a wide range of matrices.

As in the frontal method, multifrontal methods use dense matrices in the innermost loops. There is, however, more data movement than in the frontal scheme, and the innermost loops are not so dominant.

In the Harwell Subroutine Library, `MA41` by Amestoy and Duff [17] is a multifrontal code for non-element matrices while the `MA46` code of Damhaug and Reid [18] is designed for element problems. For symmetric problems, routines `MA27` and `MA47` (and complex versions `ME27` and `ME47`) are available.

Although sparsity pivoting is usually separated from numerical pivoting, a recent variant of the multifrontal approach due to Davis and Duff [19] combines the analyse and factorize phases. This approach is implemented as subroutine `MA38` in the Harwell Subroutine Library.

5 A Comparison of Codes

The performance of the codes `MA42`, `MA41`, `MA48`, and `MA38` is compared in Table 3. The timings are in seconds on a Sun ULTRA-1 workstation. `MA42` is used with the row reordering package `MC62`. The results show that no single code is clearly better than the others. The choice of code is dependent on the problem being solved. `MA41` generally has the fastest factorize time for problems such as PORES 3, which have a nearly symmetric structure. For problems that are far from symmetric in structure (for example, LHR14C), `MA38` is very competitive, while `MA48` performs well when the matrix is very sparse (problem WEST2021). `MA42` has the advantage of requiring much less in-core storage than the other codes and this reduction in main memory requirement can mean that it is feasible to solve problems with the frontal code which cannot be solved using the other codes.

Finally, we remark that the comparative behaviour of the codes in terms of timings is, to some extent, dependent on the computing environment. In particular, the performance of `MA42` is impressive on vector machines and if out-of-core storage is used, its performance is significantly affected by the speed

Table 3. A comparison of HSL codes on unsymmetric assembled problems (Sun ULTRA-1).

Identifier	Order	No. of entries	Code	Factor time	Factor ops $(*10^6)$	Storage (Kwords) In-core	Factor
BAYER04	20545	159082	`MA42`	10.1	123.8	32	2333
			`MA41`	14.9	159.0	2757	1928
			`MA48`	7.8	15.7	1130	926
			`MA38`	11.9	55.7	1557	1138
LHR07C	7337	156508	`MA42`	8.0	48.7	22	936
			`MA41`	11.2	151.6	2416	1387
			`MA48`	11.8	56.3	1553	1253
			`MA38`	9.1	33.9	1317	936
LHR14C	14270	307858	`MA42`	17.7	129.1	94	2041
			`MA41`	48.1	315.2	5025	3498
			`MA48`	24.4	88.8	2978	2399
			`MA38`	16.6	63.5	2201	1728
NNC1374	1374	8606	`MA42`	0.52	5.3	5	137
			`MA41`	0.58	9.2	209	154
			`MA48`	0.63	5.0	155	135
			`MA38`	1.03	5.6	184	129
PORES 3	532	3474	`MA42`	0.15	0.3	1	19
			`MA41`	0.06	0.2	33	16
			`MA48`	0.08	0.9	26	18
			`MA38`	0.11	0.2	34	20
WEST2021	2021	7353	`MA42`	0.32	1.32	2	82
			`MA41`	0.19	0.35	104	47
			`MA48`	0.10	0.05	44	25
			`MA38`	0.24	0.06	73	43

of the i/o. It is important to exploit machine characteristics, such as cache, for efficient implementation. Some experiments using other computing environments are reported on by Duff and Scott [20].

6 Computing the Inverse of a Sparse Matrix

Once the LU factors of a matrix have been computed they can be used to solve for any number of right-hand sides b. Some software, including `MA42`, allows the user to input multiple right-hand sides. BLAS 3 routines can then be exploited in the solve phase and this allows a single call with $k > 1$ right-hand sides to be significantly faster than k calls with a single right-hand side (see Duff and Scott [10]).

An important special case of more than one right-hand side is where the inverse A^{-1} is required, since this can be obtained by solving

$$AX = I$$

by taking columns of I as successive right-hand side vectors. If a sequence of problems with the same matrix but different right-hand sides b is to be solved, it is tempting to calculate the inverse and use it to form the product

$$x = A^{-1}b. \tag{6}$$

However, there is almost no occasion when it is appropriate to compute the inverse in order to solve a linear system. In general, the inverse of a sparse matrix is dense, whereas the L and U factors are usually sparse, so using the relation (6) can be many times more expensive than using the relation $LUx = b$.

There are many applications where only specified entries of the inverse of A are required. For example, the diagonal entries of A^{-1} may be needed. Since solving (6) computes the entries of A^{-1} a column at a time, the entire lower triangle of A^{-1} would have to be computed to obtain all the diagonal entries. This may be avoided using the algorithm of Erisman and Tinney [21], which allows advantage to be taken of sparsity. Let $Z = A^{-1}$ and suppose a sparse factorization of A

$$A = LDU$$

has been computed, where L and U are unit lower and upper triangular matrices, respectively, and D is diagonal. It can be shown that

$$Z = D^{-1}L^{-1} - (I - U)Z$$

and

$$Z = U^{-1}D{-}1 + Z(I - L).$$

Since $(I - L)$ and $(I - U)$ are strictly lower and upper triangular matrices, respectively, the following relations hold:

$$z_{ij} = [(I - U)Z]_{ij}, \quad i < j,$$

$$z_{ij} = [Z(I - L)]_{ij}, \quad i > j,$$

$$z_{ii} = d_i^{-1}i + [(I - U)Z]_{ii},$$

and

$$z_{ii} = d_i^{-1}i + [Z(I - L)]_{ii}.$$

Using the sparsity of L and U, these formulae provide a means of computing particular entries of Z from previously computed ones. Further details are given in the book by Duff, Erisman, and Reid [2].

7 Eigenvalue Problems

Solution methods for large sparse linear systems of equations are important in eigenvalue calculations. Large-scale generalized eigenvalue problems of the form

$$Ax = \lambda Bx \tag{7}$$

arise in many application areas, including structural dynamics, quantum chemistry, and computational fluid dynamics. In many cases, only a few eigenvalues are required (for example, the largest or smallest eigenvalues). Solution techniques involve iterative methods based on Krylov subspaces. Well-known approaches include subspace iteration, the Lanczos method and Arnoldi's method. The book by Saad [22] provides a useful introduction to numerical methods for large-scale eigenvalues.

Before applying a Lanczos or Arnoldi method, we transform (7) into a standard eigenvalue problem of the form

$$Tx = \theta x.$$

Iterative eigensolvers rapidly provide approximations to well-separated extremal eigenvalues. T should therefore be chosen so that the sought-after eigenvalues of (A, B) are transformed to well-separated extremal eigenvalues of T that are easily recoverable from the eigenvalues of T. Additionally, because the iterative eigensolvers involve matrix-vector products and for large problems, these can represent the dominant cost, we need to select T so that $y = Tv$ can be computed efficiently.

A frequently used transformation is the shift-invert transformation

$$T_{SI}(\sigma) = (A - \sigma B)^{-1}B.$$

The scalar σ is the *shift* or *pole*. Because eigenvalues close to σ are mapped away from the origin while those lying far from σ are mapped close to zero, T_{SI} is useful for computing eigenvalues of (A, B) lying close to σ. Performing matrix-vector products $y = T_{SI}v$ is equivalent to solving the linear system

$$(A - \sigma B)y = b, \tag{8}$$

where $b = Bv$. Thus the efficiency of the eigensolver depends on the efficiency with which linear systems can be solved.

In recent years, a number of software packages have been developed for large-scale eigenvalue problems, including `EB12` and `EB13` in the Harwell Subroutine Library (Duff and Scott [23], Scott [24]) for unsymmetric problems, and the very general ARPACK package of Lehoucq, Sorensen, and Yang [25]. These codes use a reverse communication interface so that, each time a matrix-vector product $y = Tv$ is required, control is returned to the user. This approach allows the user to exploit the sparsity and structure of the matrix and to take full advantage of parallelism and/or vectorization. Additionally, the user can incorporate different preconditioning techniques in a straightforward way. For shift-invert transformations, if a direct method of solution is used for the linear system (8), the LU factorization of $(A - \sigma B)$ need be performed only once for each shift.

8 Brief Summary

We have given a brief introduction to direct methods for solving large sparse systems of linear equations. Numerical examples have shown that there is no single method and no single code that is the best for all applications. The size of problem that can be solved using direct methods is constantly growing with the development of more sophisticated numerical techniques and advances in computer architecture. However, direct methods cannot be used for really large systems (particularly those for which the underlying problem is three-dimensional). In this case, iterative methods or techniques that combine elements of both direct and iterative methods will have to be used.

9 Availability of Software

All the codes highlighted in this paper are written in ANSI Fortran 77 and all except `ARPACK` are available through the Harwell Subroutine Library. Anybody interested in using any of the HSL codes should contact the HSL Manager: Scott Roberts, AEA Technology, Building 477 Harwell, Didcot, Oxfordshire OX11 0RA, England, tel. +44 (0) 1235 432682, fax +44 (0) 1235 432023, email Scott.Roberts@aeat.co.uk, who will provide licencing information. Academic licences are available at a nominal cost. Further information may also be found on the World Wide Web at `http://www.dci.clrc.ac.uk/Activity/HSL`.

The `ARPACK` code is in the public domain and may be accessed at `http://www.caam.rice.edu/software/ARPACK/`.

There is a limited amount of sparse matrix software that implements direct methods available within the public domain. Some codes can be obtained through `netlib` (`http://www.netlib.org`) and others from the Web pages of the researcher developing the code. The problem with the latter source is that, in general, there is no guarantee of quality control or of software maintenance and user support.

Further information on sources of software for sparse linear systems may be found in the recent report of Duff [4].

Acknowledgements

I am grateful to Iain Duff of the Rutherford Appleton Laboratory and Walter Temmerman of Daresbury Laboratory for helpful comments on a draft of this paper.

References

1. Harwell Subroutine Library, A Catalogue of Subroutines (Release 12), Advanced Computing Department, AEA Technology, Harwell Laboratory, Oxfordshire, England (1996).
2. I.S. Duff, A.M. Erisman, and J.K. Reid, Direct Methods for Sparse Matrices, Oxford University Press, England (1986).
3. I.S. Duff, in The State of the Art in Numerical Analysis, I.S. Duff and G.A. Watson, eds., Oxford University Press, England (1997).
4. I.S. Duff, Technical Report, RAL-TR-1998-054, Rutherford Appleton Laboratory (1998).
5. I.S. Duff, R.G. Grimes, and J.G. Lewis, Technical Report, RAL-TR-92-086, Rutherford Appleton Laboratory (1992).
6. H.M. Markowitz, Management Science, **3**, 255 (1957).
7. I.S. Duff and J.K Reid, ACM Trans. Math. Soft., **22**, 187 (1996).
8. J.J. Dongarra, J. DuCroz, I.S. Duff, and S. Hammarling, ACM Trans. Math. Soft., **16**, 1 (1990).
9. I.S. Duff, Report AERE R10079, Her Majesty's Stationery Office, London (1981).
10. I.S. Duff and J.A. Scott, ACM Trans. Math. Soft., **22**, 30 (1996).
11. I.S. Duff and J.A. Scott, Technical Report RAL-TR-97-012, Rutherford Appleton Laboratory (1997).
12. J.K. Reid and J.A. Scott, Technical Report RAL-TR-98-016, Rutherford Appleton Laboratory (1998).
13. J.A. Scott, Technical Report RAL-TR-1998-031, Rutherford Appleton Laboratory (1998).
14. J.A. Scott, Technical Report RAL-TR-1998-056, Rutherford Appleton Laboratory (1998).
15. I.S. Duff and J.A. Scott, in Proceedings of the Fifth SIAM Conference on Applied Linear Algebra, J.G. Lewis, ed., SIAM, Philadelphia (1994).
16. I.S. Duff, and J. Koster, Technical Report RAL-TR-97-059, Rutherford Appleton Laboratory (1997).
17. P.R. Amestoy and I.S. Duff, Inter. J. of Supercomputer Applics, **3**, 41 (1989).
18. A.C Damhaug and J.K. Reid, Technical Report RAL-TR-96-10, Rutherford Appleton Laboratory (1996).
19. T.A. Davis and I.S. Duff, SIAM J. Matrix Analysis and Applics, **18**, 140 (1997).
20. I.S. Duff and J.A. Scott, Technical Report RAL-TR-96-102 (revised), Rutherford Appleton Laboratory (1996).
21. A. Erisman and W.F. Tinney, Communications ACM **18**, 177 (1975).
22. Y. Saad, Numerical Methods for Large Eigenvalue Problems, Halsted Press (1992).

23. I.S. Duff and J.A. Scott, ACM Trans. Math. Soft., **19**, 137 (1993).
24. J.A. Scott, ACM Trans. Math. Soft., **21**, 432 (1995).
25. R.B. Lehoucq, D.C. Sorensen and C. Yang, ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods. SIAM, Philadelphia, PA (1998).