Matrix Methods[1]

Iain S. Duff[2]

ABSTRACT

We consider techniques for the solution of linear systems and eigenvalue problems. We are concerned with large-scale applications where the matrix will be large and sparse.
We discuss both direct and iterative techniques for the solution of sparse equations, contrasting their strengths and weaknesses and emphasizing that combinations of both are necessary in the arsenal of the applications scientist.
We briefly review matrix diagonalization techniques for large-scale problems.

Keywords: sparse matrices, sparse linear equations, direct methods, iterative methods, large-scale applications, matrix diagonalization, eigenproblems, mathematical software.

AMS(MOS) subject classifications: 65F05, 65F50.

---

[1]This paper is based on an invited presentation at the meeting "Supercomputing, Collision Processes and Applications" that was held at The Queen's University of Belfast from 14th to 16th September 1998 to mark the retirement of Professor P G Burke CBE FRS. The Proceedings will be published by Plenum Publishing Corporation.

[2]I.S.Duff@rl.ac.uk, `http://www.dci.clrc.ac.uk/Person/I.S.Duff`

Current reports available by anonymous ftp from ftp.numerical.rl.ac.uk in the directory "pub/reports". This report is in file duffRAL98076.ps.gz.

Department for Computation and Information
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

9.1 Introduction

The intention of this paper is to describe current matrix methods for large-scale problems to an audience of computational physicists and chemists. We will discuss both the solution of the linear equations

$$\mathbf{Ax} = \mathbf{b}, \tag{9.1}$$

and the solution of the eigensystem

$$\mathbf{Ax} = \lambda\mathbf{Bx}, \tag{9.2}$$

where the matrices $\mathbf{A}$ and $\mathbf{B}$ are large and sparse. The problem (9.2) is called the generalized eigenproblem. The particular, commonly occurring, case where $\mathbf{B} = \mathbf{I}$, viz.

$$\mathbf{Ax} = \lambda\mathbf{x}, \tag{9.3}$$

is called the eigenproblem, or *standard* eigenproblem. The solution of the eigenproblem for all values of λ and $\mathbf{x}$ corresponds to finding a similarity transformation for diagonalizing the matrix and is often called *matrix diagonalization.* In many cases, however, the full diagonal is not required and only a few eigenvalues λ are needed.

Although it is possible to use the solution of the eigenproblem to facilitate solutions of the linear system (9.1), and I have known people to use this route, I should stress that the problem (9.3) is usually much more complicated to solve than the problem (9.1), not least because (9.3) is nonlinear in the unknowns λ and $\mathbf{x}$. Thus, if the solution to (9.1) is all that is required (even for several right-hand sides $\mathbf{b}$), then this should be tackled directly.

Sparse systems arise in very many application areas. We list just a few such areas in Table 9.1.

Table 9.1: **A list of some application areas for sparse matrices**

acoustic scattering	4	demography	3	network flow	1
air traffic control	1	economics	11	numerical analysis	4
astrophysics	2	electric power	18	oceanography	4
biochemical	2	electrical engineering	1	petroleum engineering	19
chemical engineering	16	finite elements	50	reactor modeling	3
chemical kinetics	14	fluid flow	6	statistics	1
circuit physics	1	laser optics	1	structural engineering	95
computer simulation	7	linear programming	16	survey data	11

This table, reproduced from [1], shows the number of matrices from each discipline present in the Harwell-Boeing Sparse Matrix Collection. This standard set of test problems is currently being upgraded to a new Collection called the Rutherford-Boeing Sparse Matrix Collection [2] that will include far larger systems and matrices from an even wider range of disciplines. This new Collection will be available from `netlib` (`http://www.netlib.org`) and the Matrix Market (`http://math.nist.gov/MatrixMarket`).

The definition of a large sparse matrix is a matter for some debate. Suffice it to say that we regard a matrix as large if it cannot be factorized efficiently using a code for general linear systems from a standard package for dense systems, such as LAPACK [3]. The order of a matrix that is considered large is thus a function of time depending on the development of both dense and sparse codes and advances in computer architecture. Partly for amusement, we show in Table 9.2 the order of general unstructured matrices which sparse methods have been used to solve as a function of the date. I think this alone serves to illustrate some of the advances in sparse solution methods over the last 25 years.

Table 9.2: **Order of general sparse matrices solved by direct methods as a function of date**

date	order
1970	200
1975	1000
1980	10000
1985	100000
1990	250000
1995	1000000

The matrix is sparse if the presence of zeros within the matrix enables us to exploit this fact and obtain an economical solution.

There are two main classes of techniques for solving (9.1), iterative methods and direct methods. In a direct method, we use a factored representation and solve the system using these factors in a predetermined amount of memory and time, usually to a high degree of accuracy. In iterative methods, we construct a sequence of approximations to the solution, often the "best" approximation in subspaces of increasing dimension. The work is generally low per iteration but the number of iterations is usually not known *a priori* and may be high, particularly if an accurate solution is required. We consider general aspects of the solution of large sparse linear equations in Section 9.2. We discuss direct methods of solution in Section 9.3 and iterative techniques in Section 9.4 and compare and combine these approaches in Section 9.5. We briefly review matrix diagonalization in Section 9.6 indicating the relationship of techniques used in this case with those used in iterative methods for solving linear equations. We make a few comments on software availability and concluding remarks in Sections 9.7 and 9.8, respectively.

9.2 The solution of linear equations

It is a notational convenience to denote by $\mathbf{A}^{-1}$ the inverse of the matrix $\mathbf{A}$ so that the solution to (9.1) is given by $\mathbf{A}^{-1}\mathbf{b}$. However, there is almost no occasion when it is appropriate to compute the inverse in order to solve a set of linear equations. Even if explicit entries of the inverse are required, for example for sensitivity analysis, there are usually far more computationally efficient ways of doing this than to compute the inverse. For example, the ith column of the inverse can be obtained by solving a set of equations with $\mathbf{e_i}$, the ith column of the identity matrix, as the right-hand side vector and, if specified entries are required, for example the diagonal of $\mathbf{A}^{-1}$, then advantage can be taken of sparsity to compute this efficiently [4].

9.2.1 Accuracy, stability, and conditioning

As a numerical analyst, I am of course concerned about the accuracy of the solution, a concern which I hope is shared by the applications scientist or engineer. Before we continue our discussion on accuracy, it might be useful to first distinguish two concepts which are often confused, namely conditioning and stability. Conditioning is a property of the problem being solved. If the problem is badly conditioned, then small perturbations to the given data could give large changes to the solution, even if it is computed exactly. Stability is a property of the algorithm that is used to effect the solution. An algorithm is *backward* stable if the solution it computes in finite precision arithmetic is the exact solution of a slightly perturbed problem.

A good measure of accuracy would be to measure the difference between computed and exact solutions in some norm, say the l_2 norm, but that is rather difficult since, if you already know the exact solution, there seems little point in going to the trouble of solving the system. A measure which is more easy to compute is the residual $\mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}$, where $\tilde{\mathbf{x}}$ is the computed solution. The residual is a measure of how well your computed solution satisfies the equation. As is common when we do not know (or cannot control) the scaling of the system, we use a relative measure of the residual (dividing by $(||\mathbf{A}||\ ||\tilde{\mathbf{x}}||)$ or some such quantity[2]). This is directly related to the perturbation to the original data that would be needed to ensure that we have computed an exact solution to the perturbed system and is called the backward error, a concept which was pioneered by Jim Wilkinson (for example, [5]) and which revolutionized the thinking of numerical analysts. Now, the backward error is related to the actual (or forward) error through the relationship

$$\text{Forward error} \leq \text{Condition number} \times \text{Backward error} \tag{9.4}$$

where, as we remarked earlier, the *Condition number* is a property of the problem (not the solution technique). There are many different condition numbers [6] and one of the most common is given by

$$\text{Condition number} = ||\mathbf{A}||\ ||\mathbf{A}^{-1}||, \tag{9.5}$$

often denoted by $\kappa(\mathbf{A})$.

A major problem for large sparse systems is that the condition number (even if the original system is scaled) can be far greater than the reciprocal of machine precision. Thus the bound (9.4) indicates that, even if we solve with a backward error of machine precision, our solution may contain no correct digits. It is then a mute point how one decides whether the problem has been solved or not. Usually it is apparent from the underlying problem, so often the applications scientist can judge this better than the numerical analyst. Before you lose all faith in numerical analysis, I should say that this is more alarming than it might at first appear. A simple scaling[3] often helps and sometimes a more appropriate condition number, for example a component-wise one, might give a more realistic bound. However, I should stress that a small (scaled) residual does mean that we have not introduced instability in the solution process. In a sense, we are doing as well as we can even if the solution is far from what was expected.

[2]When we use norm signs, $||..||$ without a suffix, then the actual norm used is not of great importance, although one would normally use consistent norms within a single analysis or computation.

[3]By *scaling* we mean choosing diagonal matrices $\mathbf{D}_1$ and $\mathbf{D}_2$ so that the nonzero entries of the scaled matrix $\mathbf{D}_1\mathbf{A}\mathbf{D}_2$ have similar magnitude.

9.2.2 Effect of symmetry

When an applications scientist or engineer is deciding which algorithm or software to choose, one of the first questions is to ask if the matrix $\mathbf{A}$ is symmetric (or is Hermitian in the complex case). This makes a crucial difference whether direct or iterative techniques are being used for solution. For direct methods, not only are work and storage nearly halved but, particularly in the commonly occurring positive definite case, more efficient sparse data structures and sparsity based orderings can be used. For iterative methods, not only are the algorithms and software more reliable and robust, but there is often some theory to guarantee convergence. For matrix diagonalization, the normality of symmetric matrices (see Section 9.6) means that robust and accurate methods of determining eigenvalues and eigenvectors exist.

9.3 Direct methods

Direct methods use a factorization of the coefficient matrix to facilitate the solution. The most common factorization for unsymmetric systems is an *LU* factorization where the matrix $\mathbf{A}$ (or rather a permutation of it) is expressed as the product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$. Thus

$$\mathbf{PAQ} = \mathbf{LU}, \tag{9.6}$$

where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices. This factorization can then be used to solve the system (9.1) through the two steps:

$$\mathbf{Ly} = \mathbf{Pb}, \tag{9.7}$$

and

$$\mathbf{Uz} = \mathbf{y}, \tag{9.8}$$

whence the solution $\mathbf{x}$ is just a permutation of the vector $\mathbf{z}$, viz.

$$\mathbf{x} = \mathbf{Qz}. \tag{9.9}$$

This use of *LU* factorization to solve systems of equations is usually termed *Gaussian elimination* and indeed the terms are often used synonymously. Another way of viewing Gaussian elimination is as a multistage algorithm which processes the equations in some order. At each stage, a variable is chosen in the equation and is eliminated from all subsequent equations by subtracting an appropriate multiple of that equation from all subsequent ones. The coefficient of the chosen variable is called the *pivot* in Gaussian elimination and the multiple of the pivot row or equation is called the *multiplier*. Clearly, there must be some reordering performed (called *pivoting*) if a pivot is zero but equally pivoting will normally be necessary if the pivot is very small (in fact if the multipliers are large) relative to other entries since then original information could be lost (from adding very large numbers to relatively very small numbers in finite-precision arithmetic) and we could solve a problem quite different from that originally intended. In the sparse case, pivoting is also required to preserve sparsity in the factors. For example, if the matrix $\mathbf{A}$ is an arrowhead matrix[4], then selecting entry (n, n) as pivot will give dense triangular factors while choosing pivots from the diagonal in any order with entry (n, n) chosen last will give no

[4]An arrowhead matrix, $\mathbf{A}$, has nonzero entries only in positions $a_{ii}, a_{i,n}$, and $\mathrm{a}_{\mathrm{n,i}}$, $\mathrm{i} = 1, \ldots, \mathrm{n}$

fill-in (that is, there will be no entries in positions that were not entries in the original matrix). Of course, such a choice could be bad for the numerical criterion just mentioned above. The reconciliation of these possibly conflicting goals of pivoting has been a topic for research. We touch on this briefly below.

If the matrix $\mathbf{A}$ is symmetric positive definite, it is normal to use the factorization

$$\mathbf{PAP}^T = \mathbf{LL}^T. \tag{9.10}$$

The factorization (9.10) is called a Cholesky factorization. For more general symmetric matrices, the factorization

$$\mathbf{PAP}^T = \mathbf{LDL}^T, \tag{9.11}$$

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

9.3.1 Phases in solution

In both the case of sparse and dense matrices, the factorization (9.6) is more expensive than the forward elimination and backsubstitution phases, (9.7) and (9.8) respectively. This is less significant if several sets of equations with the same matrix but differing right-hand sides need to be solved. The higher cost of the factorization can then be amortized over the cost of the multiple solutions. In the sparse case, a further distinction is important. Often much of the work concerning handling sparse data structures and choosing pivots can be performed once only for a particular matrix structure and the subsequent factorization of matrices with the same structure can be performed much more efficiently using information from this first factorization. In some cases, the differences in execution time can be quite dramatic as we illustrate in Table 9.3 where there is an order of magnitude difference in time for the three phases. The code `MA48` is a general sparse unsymmetric solver from the Harwell Subroutine Library and GRE 1107 is a test matrix from the Harwell-Boeing Sparse Matrix Collection. The ability to refactorize efficiently subsequent matrices is not present in all software packages but is very important particularly when solving nonlinear systems when the Jacobian will retain the same structure although the numerical values change.

Table 9.3: **Execution times (in seconds) for code `MA48` for matrix GRE 1107 on a single processor of a CRAY Y-MP**

First factorization	.66
Subsequent factorizations	.075
Back and forward substitution	.0068

9.3.2 Sparsity preservation and numerical stability

In the sparse case, it is crucial that the permutation matrices of (9.6) are chosen to preserve sparsity in the factors as well as to maintain stability and many algorithms have been developed to achieve this. For general unsymmetric matrices the most popular method is called "Markowitz with *threshold pivoting*". Threshold pivoting ensures that pivots are within a certain threshold of the largest entry in the row or column. The threshold is often an input parameter and a typical value for it is 0.1. We control sparsity by choosing the pivot to be an entry satisfying the threshold criterion that has the fewest product of number of other entries in its row and column. The suggestion of using this sparsity control is due to [7]. In the symmetric case, the Markowitz analogue is *minimum degree* where one chooses as pivot a diagonal entry with the least number of entries in its row.

9.3.3 Arithmetic complexity

Although the LU factorization has a similar $\mathcal{O}(n^3)$ complexity[5] to matrix inversion for dense systems (and storage requirements of $\mathcal{O}(n^2)$), the complexity of the factorization process and storage requirements for sparse matrices depend on the structure and can be significantly less. For example the LU factorization of a tridiagonal matrix can be done in $\mathcal{O}(n)$ operations whereas the calculation of the inverse is at best $\mathcal{O}(n^2)$. Furthermore, the storage for the factors of a tridiagonal matrix are the same as the original matrix ($3n - 2$ reals) but the inverse of a tridiagonal matrix is dense. In fact, if we consider structure only and do not allow numerical cancellation, the inverse of an irreducible sparse matrix[6] is always dense [8]. An archetypal example, is a five-diagonal matrix[7] as obtained, for example, from the finite-difference discretization of a two-dimensional Laplacian. If the discretization has k grid points in each direction (so that the order of the matrix is k^2), the LU factorization would require $\mathcal{O}(k^6)$ if considered as a dense system but $\mathcal{O}(k^4)$ if considered banded (and the storage requirement reduced from k^4 to k^3). Although this figure is often quoted when comparing the complexity of LU factorization with other methods on such matrices, by using a nested dissection ordering algorithm, the work and storage can be reduced to $\mathcal{O}(k^3)$ and $\mathcal{O}(k^2 \log k)$ respectively. The bad news is that one can prove that, for a wide range of matrices including the five-diagonal one, this is asymptotically the best that can be done for any direct method based on LU factorization.

9.3.4 Indirect addressing and the use of the BLAS

For arbitrarily structured sparse matrices, complicated data structures are needed for efficient execution (for example, [9]). Although this is hidden from the user of the sparse code, it can significantly affect the efficiency of the computation. Even for computers with hardware indirect addressing, access to data of the form `A(IND(I)), I = 1, K` carries a heavy penalty in terms of additional memory traffic and non-localization of data. When this is added to the fact that most loops are of the order of number of nonzero entries in a row rather than the order of the system, general sparse codes can perform particularly badly on some high performance computers relative to their dense counterpart. Much recent research on sparse direct techniques has been to develop algorithms and codes that use the same kernels as dense codes at the innermost loops.

[5]See, however, the comments on Strassen's algorithm which follow.

[6]A matrix is irreducible if it is not possible to reorder the matrix rows and columns so the reordered form has a nontrivial block triangular form.

[7]By five-diagonal matrix, we mean a matrix that has nonzeros only in five diagonals.

The BLAS, or Basic Linear Algebra Subprograms, are well established standard operations on dense matrices and vectors, and include computations such as scalar products, solution of triangular systems, and multiplication of two matrices ([10], [11], [12]). The important thing is that the interface to each subprogram has been standardized and most vendors have developed optimized code for these kernels. For example, the matrix-matrix multiplication routine (`_GEMM`) performs at close to peak performance on many computers, even those with quite sophisticated architectures involving vector processing, memory hierarchy, caches etc. Much of the recent work in dense linear algebra has centred round the use of these kernels. However, in spite of some early work by [13] and others, it is only quite recently that it has become appreciated that these dense linear algebra kernels can be used effectively within direct methods for sparse matrices.

Table 9.4: **Performance of `_GEMM` kernel in Mflop/s on a range of machines (single processor performance)**

Machine	*Peak*	`_GEMM`
Meiko CS2-HA	100	88
IBM SP-2	266	213
Intel PARAGON	75	68
DEC Turbo Laser	600	450
CRAY 2	459	449
CRAY YMP	333	313

We show, in Table 9.4, the performance of the Level 3 BLAS kernel `_GEMM` on a range of computers with various floating-point chips and memory organizations. In many cases, this kernel attains about 90% or more of the peak performance of the chip and in every case more than 75% of peak is achieved. This remarkable performance is obtained by the fact that a (dense) matrix-matrix multiply performs $2n^3$ arithmetic operations but only requires $3n^2$ data references. This enables data that is brought into the memory hierarchy (say onto an on-chip cache) to be reused, thus amortizing the cost of transferring it from main memory or even further afield. Since the memory level closest to the floating-point unit can usually supply data at the same speed as the unit computes, we can then get close to the speed of the floating-point unit.

As a footnote to the complexity issue, we should record the fact that methods have been developed for multiplying dense matrices in $\mathcal{O}(n^\alpha)$ operations, where $\alpha < 3$, based on Strassen's method [14]. `_GEMM` has been implemented using this algorithm (for example, [15]) and so the use of this kernel in dense Gaussian elimination can reduce the complexity accordingly. Note that the added complexity of Strassen's algorithm and the need to pay more care to stability [16] means that this is not the panacea to the "n^3 problem". Also the lowest value of α that has been so far obtained is 2.376 so that dense matrix computations still quickly become infeasible when the matrix order becomes very large.

The trick is now to develop sparse matrix techniques that can take advantage of these fast dense matrix kernels. Of course, it is possible just to solve the sparse system using a code for dense systems, and some people have advocated this approach arguing that the greater "peak" speed and memory of modern computers makes this feasible. I must stress that the complexity discussions we had earlier makes this really non-viable for all but the smallest sparse matrices. We illustrate the wide difference in execution times for sparse and dense codes on sparse matrices by the results in Table 9.5. Although there are now much faster machines than a CRAY Y-MP, the matrices used in this table are quite small by current standards.

Table 9.5: **Comparison between `MA48` (a sparse code) and LAPACK (`SGESV`) (a dense code) on a range of matrices from the Harwell-Boeing Sparse Matrix Collection.** Times are for factorization and solution (in seconds on one processor of a CRAY Y-MP)

Matrix	Order	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
ORSREG 1	2205	14133	2.65	24.25
ORANI678	2529	90158	1.17	36.37

9.3.5 Frontal and multifrontal methods

A more viable approach is to order the sparse matrix so that its nonzero entries are clustered near the diagonal (called bandwidth minimization) and then regard the matrix as banded, treating zeros within the band as nonzero. However, this is normally too wasteful as even the high computational rate of the Level 3 BLAS does not compensate for the extra work on the zero entries. A variable band format is used to extend the range of applicability of this technique. A related, but more flexible scheme, is the frontal method which owes its origin to computations using finite elements.

Here we assume that $\mathbf{A}$ is of the form

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

where each element matrix $\mathbf{A}^{[l]}$ has nonzeros in only a few rows and columns and is normally held as a small dense matrix representing contributions to $\mathbf{A}$ from element l. If a_{ij} and $a_{ij}^{[l]}$ denote the (i, j)th entry of $\mathbf{A}$ and $\mathbf{A}^{[l]}$, respectively, the basic assembly operation when forming $\mathbf{A}$ is of the form

$$a_{ij} \Leftarrow a_{ij} + a_{ij}^{[l]}. \tag{9.12}$$

It is evident that the basic operation in Gaussian elimination

$$a_{ij} \Leftarrow a_{ij} - a_{ip}[a_{pp}]^{-1} a_{pj} \tag{9.13}$$

may be performed as soon as all the terms in the triple product (9.13) are *fully summed* (that is, are involved in no more sums of the form (9.12)). The assembly and Gaussian elimination processes can therefore be interleaved and the matrix $\mathbf{A}$ is never assembled explicitly. Variables that are internal to a single element can be immediately eliminated (called *static condensation*) and this can be extended to a submatrix from a set of elements, that is a sum of several element matrices. In this scheme, all intermediate working can be performed within a dense matrix, termed the *frontal matrix*, whose rows and columns correspond to variables that have not yet been eliminated but occur in at least one of the elements that have been assembled.

We can partition the frontal matrix, $\mathbf{F}$, as:

$$\mathbf{F} = \begin{bmatrix} \mathbf{F}_{11} & \mathbf{F}_{12} \\ \mathbf{F}_{21} & \mathbf{F}_{22} \end{bmatrix} \tag{9.14}$$

where the fully summed variables correspond to the rows and columns of the block $\mathbf{F}_{11}$, from where the pivots can be chosen. The kernel computation in a frontal scheme is then of the form

$$\mathbf{F}_{22} \leftarrow \mathbf{F}_{22} - \mathbf{F}_{21}\mathbf{F}_{11}^{-1}\mathbf{F}_{12} \tag{9.15}$$

and can be performed using Level 3 BLAS. (We note that this expression is notational and the inverse of $\mathbf{F}_{11}$ is not explicitly calculated.)

The frontal method can be easily extended to non-element problems since any set of rows of a sparse matrix can be held in a rectangular array whose number of columns is equal to the number of columns with nonzero entries in the selected rows. A variable is regarded as fully summed whenever the equation in which it last appears is assembled. These frontal matrices can often be quite sparse but are suitable for computations involving Level 3 dense BLAS. A full discussion of the equation input can be found in [17].

If the frontal scheme is combined with an ordering to preserve sparsity and reduce the number of floating-point operations and if a new frontal matrix can be formed independently of already existing frontal matrices, we can develop a scheme that combines the benefits of using Level 3 BLAS with the gains from using sparsity orderings. This is developed in *multifrontal schemes* where the computation can be viewed as a tree, whose nodes represent computations of the form (9.15) and whose edges represent the transfer of data from the child to the parent (data of the form of the $\mathbf{F}_{22}$ matrices generated by (9.15)). Another approach that combines sparse ordering schemes with higher level dense BLAS is the supernodal approach (for example, [18]).

9.3.6 Parallelization of direct methods

In the late 80's and early 90's, it was almost impossible to obtain research funding in linear algebra unless parallelism was mentioned in the abstract if not the title. The topic was also embraced by computer scientists who could theorize on what the complexity of elimination techniques might be on a range of hypothetical computers and exotic parallel metacomputers with $\mathcal{O}(n^p)$ processors. I daresay some of this work was useful as other than an intellectual exercise, but the last few years have seen a more mature study of realistic parallel algorithms that can be implemented on actual available computers and can (and are sometimes) even used by applications people or industry.

Although algorithms and software for the parallel solution of dense systems of equations have been developed (for example, the ScaLAPACK package of [19]), the irregularity of sparse matrix structures makes it much more difficult to efficiently parallelize methods for sparse equations. The PARASOL Project is an ambitious attempt in this direction.

PARASOL[8] is an ESPRIT IV Long Term Research Project (No 20160) for "An Integrated Environment for Parallel Sparse Matrix Solvers". The main goal of this Project, which started on January 1 1996, is to build and test a portable library for solving large sparse systems of equations on distributed memory systems. There are twelve partners in five countries, five of whom are code developers, five end users, and two software houses. The software is written in Fortran 90 and uses MPI for message passing. The solvers being developed in this consortium are: two domain decomposition codes by Bergen and ONERA, a multigrid code by GMD, and a parallel multifrontal method (called MUMPS) by CERFACS and RAL. The final library will be in the public domain.

It is common, when examining implementations on parallel computers, to stress the speedup, or how many times faster the application runs on multiple processors than a single processor, although it is recognized that a more important measure is the execution time relative to the fastest method on a uniprocessor. However, an often more important reason for parallel computation is the the benefit of having access to more memory. This is particularly true on machines which have memory entirely local to each processor and which use message passing to share data between processors. In this case, the more processors; the more memory. A side effect of this is that it may only be possible to run a large problem on several processors so a comparison with a uniprocessor code is inappropriate. Furthermore, the memory system may be inefficient when stressed (for example, because of memory paging) and so the speedup may be superlinear. We illustrate this amusing effect by some runs of a parallel multifrontal code from the PARASOL Project on a PARASOL test problem on an IBM SP2 and an SGI Origin 2000 in Table 9.6. The speedups on the Origin reflect the true parallelism of the software, whereas those on the SP2 include a memory effect.

9.4 Iterative methods

In contrast to direct methods, iterative methods do not normally modify the matrix and do not form any representation of the inverse. Furthermore, most iterative methods do not require the matrix to be represented explicitly, sometimes it is sufficient to be able to form the product of the matrix with a vector, although this may restrict the preconditioning available (see Section 9.4.1).

Iteration techniques such as successive approximation have been around since the first days of scientific computing and the early iterative techniques for solving sparse linear equations were based on such approaches (Gauss-Seidel, SOR etc). While these methods are still used (for example, as smoothers in multigrid techniques), it is true to say that most modern methods for the iterative solution of sparse equations are based on Krylov sequences of the form

$$sp\{\mathbf{v}, \mathbf{A}\mathbf{v}, \mathbf{A}^2\mathbf{v}, ...\}, \tag{9.16}$$

[8]For more information on the PARASOL project, see the web site at `http://www.genias.de/parasol`.

Table 9.6: **Results for the symmetric version of the MUMPS code on** CRANKSEG2.

Machine	Working processors	Time for factorization
SP2	16	1045.3
	24	457.3
	32	139.7
Origin	1	635.4
	2	411.0
	3	275.7
	4	220.4
	5	175.1
	6	158.3
	7	142.9
	8	135.7

with the approximate solution at each step the "best" solution that lies in the subspace of increasing dimension. What constitutes "best" determines which of the many methods is defined.

The residual at the i-th iteration of an iterative method can be expressed as

$$\mathbf{r^{(i)}} = P_i(\mathbf{A})\mathbf{r^{(0)}}, \tag{9.17}$$

where P_i is a polynomial such that $P_i(0) = 1$. If we expand $\mathbf{r^{(0)}}$ in terms of the eigenvectors of $\mathbf{A}$ we see that we want P_i to be small on the eigenvalues of $\mathbf{A}$ so that the spectrum of $\mathbf{A}$ is crucial in determining how quickly our method converges. For example, if there are many eigenvalues close to zero or if the spectrum is widely distributed, the degree of polynomial will have to be high in order to be small on all eigenvalues and so the number of iterations required will be large.

A major feature of most Krylov based methods for symmetric systems are that they can be implemented using short term recurrences which means that only a few vectors of length n need be kept and the amount of computation at each stage of the iterative process is modest. However, Faber and Manteuffel (1984) have shown that, for general matrices, one must either lose the cheap recurrences or lose the minimization property. Thus for the solution of general unsymmetric systems, the balance between these, added to the range of quantities that can be minimized and the differing norms that can be used, has led to a veritable alphabet soup of methods [20], for example GMRES(k), CGS, Bi-CGSTAB(ℓ), TFQMR, FGMRES, GMRESR, ...

9.4.1 Preconditioning

The key to developing iterative methods for the solution of realistic problems lies in preconditioning, where by this we mean finding a matrix $\mathbf{K}$ such that

1. $\mathbf{K}$ is an approximation to $\mathbf{A}$.
2. $\mathbf{K}$ is cheap to construct and store.
3. $\mathbf{Kx} = \mathbf{b}$ is much easier to solve than the system (9.1).

We then solve the preconditioned system

$$\mathbf{K}^{-1}\mathbf{Ax} = \mathbf{K}^{-1}\mathbf{b}, \tag{9.18}$$

where we have chosen $\mathbf{K}$ so that our iterative method converges more quickly when solving equation (9.18) than equation (9.1). Lest this seem too much of a black art (which to some extent it is), if $\mathbf{K}$ were chosen as the product of the factors $\mathbf{LU}$ from an *LU* factorization (admittedly violating point 2 above), then the preconditioned matrix $\mathbf{B} = \mathbf{K}^{-1}\mathbf{A}$ would be the identity matrix and any sane iterative method would converge in a single iteration. From our earlier discussion, we would like the preconditioned matrix $\mathbf{B}$ to have a better distribution of eigenvalues than $\mathbf{A}$.

The preconditioning can also be applied as a right-preconditioning $\mathbf{AK}^{-1}$ or as a two-sided preconditioning $\mathbf{K}_1^{-1}\mathbf{AK}_2^{-1}$, when the matrix $\mathbf{K}$ can be expressed as a product $\mathbf{K}_1\mathbf{K}_2$. Common preconditioners include using the diagonal of $\mathbf{A}$ or a partial or incomplete factorization of $\mathbf{A}$. A recent simple discussion of the merits of different forms of preconditioners and their implementation can be found in the report [21] that is a preprint of a chapter in a forthcoming book [22].

A very interesting aspect of this is that convergence can occur in very few iterations if the eigenvalues are well clustered. This is, of course, true whether the matrix is dense or sparse. In the dense case, a direct method will require $\mathcal{O}(n^3)$ work whereas a single matrix vector multiplication only $\mathcal{O}(n^2)$. Thus, if our iterative method converges in only a few iterations, it may be a very attractive method for solving the dense system. An example where dense systems with well clustered eigenvalues are found is given by [23]. However, as in the case of sparse systems, preconditioning is normally needed to obtain a good eigenvalue distribution. This is less attractive in the dense case because another n^2 multiplications are required and, if it is required to solve for the preconditioning matrix, our subproblem is as hard as the original problem. It is sometimes possible, however, to develop sparse preconditioning matrices for the dense problem (for example, [24]) so that the cost of this preconditioning is small compared to the multiplication by the original matrix. In some cases, the structure of the problem can be used to reduce the matrix-vector multiplication itself, for example by using multipole methods [25].

9.4.2 Parallelization of iterative methods

In contrast to direct methods, each step of an iterative method is relatively easy to parallelize since the only numerical computations involved are:

- $\mathbf{Av}$
- $\mathbf{v}^T\mathbf{w}$

- $\mathbf{v} - \alpha \mathbf{w}$

and, for the preconditioning:

- $\mathbf{K}^{-1}\mathbf{v}$

The first operations can be performed with high efficiency on parallel computers, for example [26], although scalar products require synchronization and much communication. The achilles heel for performance is, however, usually in the efficient implementation of the preconditioning, on which much research is still being done (see for example, [22]). We should add that it is vital that the preconditioner is effective in reducing the number of iterations because, if the convergence is so slow as to be meaningless, no amount of parallelism can make it viable.

9.5 Direct or iterative or ?

I am often asked whether it is better to use an iterative or a direct method to solve a set of linear equations. The answer is often quite simple, and not just because of the predilection of my research interest. One should use a direct method! This is even more true if you have several right-hand sides to solve with the same coefficient matrix. However, since your computer is unlikely to be blessed with an infinite amount of memory (unless you are a theoretician from the 80's), the range of problems for which such a technique is applicable is limited, significantly so if your underlying problem is three-dimensional. In such cases, one has to resort to an iterative method. However, since for all but the simplest cases, your chosen iterative method (even with a standard preconditioner) is unlikely to converge ... what do you do?

The answer is to use a technique that combines elements of both direct and iterative methods. Sophisticated preconditioners come into this category. The most obvious method to combine these approaches is the block Jacobi method where the matrix is treated as a block matrix, the solution of the subproblems corresponding to the diagonal blocks are performed using a direct method, and the system is solved using a block Jacobi algorithm. Clearly this inherits much of the parallelism of the point case but should have better convergence properties. Indeed a variant corresponding to using a block Jacobi scheme on the normal equations, termed the block Cimmino algorithm, has proven quite effective (for example, [27]). As in the point case, we can sacrifice parallelism a little and gain faster convergence through the use of a block Gauss-Seidel method. The counterpart to block Cimmino is then block Kacmarz (for example, [28]).

In the framework of the solution of partial differential equations, a more general technique for combining direct and iterative methods is to use domain decomposition [29, 30], where the problem is divided into separate domains (either overlapping or non-overlapping) and the local problems can be solved using direct methods, which of course is trivial to do in parallel since these subproblems are decoupled. In the case of non-overlapping domains, usually the main issue is the solution of the problem for the interface variables. It is normal to use an iterative method for this but the main problem is then to define a preconditioner, particularly if the matrix corresponding to the interface problem is not computed explicitly.

One way in which direct methods can be used as preconditioners for iterative methods are to perform only a partial factorization, as is the case for ILU(k) preconditioners, where a limited amount of fill-in is allowed to the matrix factors so that an incomplete factorization is performed. Another approach is to perform a "full" factorization but one of a reduced or simpler problem to the original, for example for a simpler differential equation than the original.

Multigrid methods have become very popular in recent years largely because of their provably optimal performance on simple elliptic problems. In these techniques, a sequence of grids is used. The solution is required on the finest grid. A few passes of a simple iterative method are performed and the residual is projected onto the next coarser grid. A correction is obtained by projecting back the solution of the residual equations on this coarser grid to the finer grid. Since the solution of the residual equations can be performed using an even coarser grid, multiple grids can be used. The essence of the method is that the simple iterative method (or smoother) tends to efficiently remove error components that vary as quickly as the grid size and the use of coarser grids enables the smoother components of the error to be reduced. While the individual sweeps of the smoothers can be parallelized, it is harder to parallelize across the grids but there has been some recent work on this, for example by the PARASOL Project mentioned earlier.

9.6 Matrix diagonalization

What physicists call matrix diagonalization, numerical analysts call the eigenproblem. In the case of small matrices, there are many techniques for both symmetric and unsymmetric matrices that involve computing transformations of the matrix to diagonal form usually in a two-step process, the first step transforming the matrix to tridiagonal or Hessenberg form, respectively. Although the use of high Level BLAS in these computations improves their efficiency, the transformations involved destroy sparsity in a way that is not normally as recoverable as in the case of LU factorization. Additionally, in the large sparse case, normally only a few eigenvalues and eigenvectors are required. Thus for large sparse matrices, we use other methods for obtaining the eigenvalues.

In fact the basis of eigensolution techniques for large sparse matrices is the same as we just discussed for the iterative solution of sparse equations, namely the Krylov sequence (9.16). Clearly, this is a generalization of the classical power method for computing eigenvalues and eigenvectors, but differs significantly because the previous powers are taken into account so that the size of the subspace increases as iterations are performed. Additionally, in the Krylov based methods, we are free to choose appropriate bases for the subspaces.

In our brief discussion that follows, we will be concerned with the standard eigenproblem (9.3), although we note that normally the generalized problem (9.2) is first converted to a standard one, for example on a matrix of the form $(\mathbf{A} - \sigma\mathbf{B})^{-1}\mathbf{B}$ or $\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T}$, when $\mathbf{B} = \mathbf{L}\mathbf{L}^T$ is symmetric positive definite.

Whereas for linear systems, we seek solutions of the form

$$\mathbf{Q}_k\mathbf{y}_k,$$

where $\mathbf{Q}_k = [\mathbf{q}_1\mathbf{q}_2....\mathbf{q}_k]$ is the basis of the Krylov subspace of dimension k, for the eigenproblem, we solve a reduced eigenproblem for the projected subspace

$$\mathbf{Q}_k^*\mathbf{A}\mathbf{Q}_k. \tag{9.19}$$

For the eigenproblem, we can again use preconditioning techniques to accelerate convergence. In this case, the subspaces are generated for a simple function of $\mathbf{A}$ so that the eigenvalues required are better represented in subspaces of the form (9.19) of low dimension.

Since the kernels of Krylov subspace methods for finding eigenvalues are exactly those listed for the iterative solution of linear equations at the end of Section 9.4, eigenvalue calculations are also relatively easy to parallelize with possible bottlenecks in preconditioning.

A recently investigated phenomenon is that of non-normality [31]. Formally a matrix is normal if

$$\mathbf{A}^*\mathbf{A} = \mathbf{A}\mathbf{A}^*$$

and so all symmetric (or Hermitian) matrices are normal. The most important feature of normal matrices are that there exists a unitary matrix $\mathbf{Q}$ such that $\mathbf{Q}^*\mathbf{A}\mathbf{Q}$ is diagonal. This in turn means that the eigenproblem is well-conditioned in the sense that a small perturbation to the matrix produces a small perturbation of the eigenvalues. However, for non-normal but diagonalizable matrices the diagonalization $\mathbf{X}^{-1}\mathbf{A}\mathbf{X}$, implies nothing about the condition number, $\kappa(\mathbf{X})$, of $\mathbf{X}$, so that, by the Bauer-Fike theorem, if

$$\mathbf{X}^{-1}\mathbf{A}\mathbf{X} = \mathbf{D} = \mathrm{diag}\{\lambda_1, ..., \lambda_\mathrm{n}\}$$

and μ is an eigenvalue of the perturbed matrix $\mathbf{A} + \mathbf{E}$, then

$$\min_{\lambda} |\lambda - \mu| \leq \kappa(\mathbf{X})||\mathbf{E}||.$$

A similar result holds for non-diagonalizable matrices.

This means that even a matrix very close to $\mathbf{A}$ might have very different eigenvalues. This phenomenon can be viewed in a diagram containing contours of the resolvent

$$||(\mathbf{A} - \lambda\mathbf{I})^{-1}||$$

and pseudo-eigenvalues are defined by regions in the plane where the resolvent is large. There is now some debate about whether physical processes are governed by pseudo-eigenvalues rather than eigenvalues but certainly one should be very wary of making a qualitative judgement of a process or its stability on the strength of a single or few eigenvalues [32]. One should note that a complex symmetric matrix is not Hermitian and can be non-normal.

9.7 Sources of Software

The origins of much specialist work in numerical software stem from the requirement of computational physicists and chemists, and libraries and collections of software were developed to avoid duplication of effort and provide a sound base of portable codes. I would not advise the applications scientist or engineer to try to reverse this historical trend by coding his or her own matrix algorithm, even using a numerical recipes crib sheet.

There are many sources of software for sparse matrices, from the do-it-yourself approach of the Templates book for iterative methods [33] (a similar one for eigenproblems is forthcoming), to supported proprietary codes as are present in the Harwell Subroutine Library, `http://www.dci.rl.ac.uk/Activity/HSL` [34]. Some codes are available through `netlib`, `http://www.netlib.org`, and others from the Web pages of the researchers developing the code. The problem with the latter source is that, in addition to a lack of quality control, the researcher in question will often have no compunction against editing the code almost as you are downloading it.

We discuss sources of sparse matrix software in [35] and a recent report by [36] discusses, in some detail, software for iterative methods for solving linear systems. The reports by [37] ([38], [37]) discuss sparse eigenvalue software.

9.8 Brief Summary

We have presented a few obvious, and hopefully some less obvious, comments on the solution of large sparse systems and large eigenproblems. We have emphasized the similarity between iterative solution techniques and matrix diagonalization procedures, and indicated briefly how direct and iterative methods can be combined to solve really large problems. We see considerable promise in both frontal and multifrontal methods on machines with memory hierarchies or vector processors and reasonable possibilities for exploitation of parallelism by multifrontal methods. A principal factor in attaining high performance is the use of dense matrix computational kernels, which have proved extremely effective in the dense case.

Finally, we have tried to keep the narrative flowing in this presentation by avoiding an excessive number of references. For such information, we recommend, for linear systems, the recent review by [35], where 215 references are listed. [39] has written a book on large-scale eigenproblems with an emphasis on computational aspects and more recent references on iterative solution of equations and eigenproblems can be found in [40] and [41] respectively.

Acknowledgement

I would like to thank my colleagues, Valérie Frayssé and Luc Giraud from CERFACS, and Nick Gould, John Reid, and Jennifer Scott from the Rutherford Appleton Laboratory for their comments on a draft of this paper and to Walter Temmerman of the Daresbury Laboratory for checking its accessibility to my main target audience.

References

[1] Duff, I. S., Grimes, R. G. and Lewis, J. G. (1989), ‘Sparse matrix test problems’, *ACM Trans. Math. Softw.* **15**(1), 1–14.

[2] Duff, I. S., Grimes, R. G. and Lewis, J. G. (1997), The Rutherford-Boeing Sparse Matrix Collection, Technical Report RAL-TR-97-031, Rutherford Appleton Laboratory. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS, Toulouse.

[3] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S. and Sorensen, D. (1995), *LAPACK Users' Guide, second edition*, SIAM Press.

[4] Erisman, A. M. and Tinney, W. F. (1975), 'On computing certain elements of the inverse of a sparse matrix', *Communications of the Association for Computing Machinery* **18**, 177–179.

[5] Wilkinson, J. H. (1961), 'Error analysis of direct methods of matrix inversion', *J. ACM* **8**, 281–330.

[6] Higham, N. J. (1996), *Accuracy and Stability of Numerical Algorithms*, SIAM Press, Philadelphia.

[7] Markowitz, H. M. (1957), 'The elimination form of the inverse and its application to linear programming', *Management Science* **3**, 255–269.

[8] Duff, I. S., Erisman, A. M., Gear, C. W. and Reid, J. K. (1988), 'Sparsity structure and Gaussian elimination', *SIGNUM Newsletter* **23**(2), 2–8.

[9] Duff, I. S., Erisman, A. M. and Reid, J. K. (1986), *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, England.

[10] Lawson, C. L., Hanson, R. J., Kincaid, D. R. and Krogh, F. T. (1979), 'Basic linear algebra subprograms for Fortran usage', *ACM Trans. Math. Softw.* **5**, 308–323.

[11] Dongarra, J. J., Du Croz, J. J., Hammarling, S. and Hanson, R. J. (1988), 'An extented set of Fortran Basic Linear Algebra Subprograms', *ACM Trans. Math. Softw.* **14**, 1–17.

[12] Dongarra, J. J., Du Croz, J., Duff, I. S. and Hammarling, S. (1990), 'A set of Level 3 Basic Linear Algebra Subprograms.', *ACM Trans. Math. Softw.* **16**, 1–17.

[13] Duff, I. S. (1981), The design and use of a frontal scheme for solving sparse unsymmetric equations, *in* J. P. Hennart, ed., 'Numerical Analysis, Proceedings of 3rd IIMAS Workshop. Lecture Notes in Mathematics 909', Springer Verlag, Berlin, pp. 240–247.

[14] Strassen, V. (1969), 'Gaussian elimination is not optimal', *Numerische Mathematik* **13**, 354–356.

[15] Douglas, C. C., Heroux, M., Slishman, G. and Smith, R. M. (1992), GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm, Technical Report RC 18026 (79130), IBM T. J. Watson Research Centre, P. O. Box 218, Yorktown Heights, NY 10598.

[16] Higham, N. J. (1990), 'Exploiting fast matrix multiplication within the Level 3 BLAS', *ACM Trans. Math. Softw.* **16**, 352–368.

[17] Duff, I. S. (1984), 'Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core', *SIAM J. Scientific and Statistical Computing* **5**, 270–280.

[18] Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S. and Liu, J. W. H. (1995), A supernodal approach to sparse partial pivoting, Technical Report UCB//CSD-95-883, Computer Science Division, U. C. Berkeley, Berkeley, California.

[19] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1997), *ScaLAPACK Users' Guide*, SIAM Press.

[20] Saad, Y. (1996), *Iterative methods for sparse linear systems*, PWS Publishing, New York, NY.

[21] Duff, I. S. and van der Vorst, H. A. (1998), Preconditioning and parallel preconditioning, Technical Report RAL-TR-1998-052, Rutherford Appleton Laboratory.

[22] Dongarra, J. J., Duff, I. S., Sorensen, D. C. and van der Vorst, H. A. (1998), *Numerical Linear Algebra for High-Performance Computers*, SIAM Press, Philadelphia.

[23] Rahola, J. (1996), Efficient solution of dense systems of linear equations in electromagnetic scattering calculations, PhD Thesis, CSC Research Reports R06/96, Center for Scientific Computing, Department of Engineering Physics and Mathematics, Helsinki University of Technology.

[24] Alléon, G., Benzi, M. and Giraud, L. (1997), 'Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics', *Numerical Algorithms* **16**(1), 1–15.

[25] Rahola, J. (1998), Experiments on iterative methods and the fast multipole method in electromagnetic scattering calculations, Technical Report TR/PA/98/49, CERFACS, Toulouse, France.

[26] Erhel, J. (1990), 'Sparse matrix multiplication on vector computers', *Int J. High Speed Computing* **2**, 101–116.

[27] Arioli, M., Duff, I. S., Ruiz, D. and Sadkane, M. (1995), 'Block Lanczos techniques for accelerating the Block Cimmino method', *SIAM J. Scientific Computing* **16**(6), 1478–1511.

[28] Bramley, R. and Sameh, A. (1992), 'Row projection methods for large nonsymmetric linear systems', *SIAM J. Scientific and Statistical Computing* **13**, 168–193.

[29] Chan, T. F. and Mathew, T. P. (1994), *Domain Decomposition Algorithms*, Vol. 3 of *Acta Numerica*, Cambridge University Press, Cambridge, pp. 61–143.

[30] Smith, B., Björstad, P. and Gropp, W. (1996), *Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations*, 1st edn, Cambridge University Press, New York.

[31] Chaitin-Chatelin, F. and Frayssé, V. (1996), *Lectures on Finite Precision Computations*, SIAM Press, Philadelphia.

[32] Trefethen, L. N., Trefethen, A. E., Reddy, S. C. and Driscoll, T. A. (1993), 'Hydrodynamics stability without eigenvalues', *Science* **261**, 578–584.

[33] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and van der Vorst, H., eds (1993), *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia.

[34] HSL (1996), *Harwell Subroutine Library. A Catalogue of Subroutines (Release 12)*, AEA Technology, Harwell Laboratory, Oxfordshire, England. For information concerning HSL contact: Dr Scott Roberts, AEA Technology, 477 Harwell, Didcot, Oxon OX11 0RA, England (tel: +44-1235-432682, fax: +44-1235-432023, email: Scott.Roberts@aeat.co.uk).

[35] Duff, I. S. (1997), Sparse numerical linear algebra: Direct methods and preconditioning, *in* I. S. Duff and G. A. Watson, eds, 'The State of the Art in Numerical Analysis', Oxford University Press, Oxford, pp. 27–62.

[36] Eijkhout, V. (1998), 'Overview of iterative linear system solver packages', *NHSE Review.* `http://www.nhse.org/NHSEreview/98-1.html`.

[37] Lehoucq, R. B. and Scott, J. A. (1996*b*), An evaluation of subspace iteration software for sparse nonsymmetric eigenproblems, Technical Report RAL-TR-96-022, Rutherford Appleton Laboratory.

[38] Lehoucq, R. B. and Scott, J. A. (1996*a*), An evaluation of Arnoldi based software for sparse nonsymmetric eigenproblems, Technical Report RAL-TR-96-023, Rutherford Appleton Laboratory.

[39] Saad, Y. (1992), *Numerical Methods for Large Eigenvalue Problems*, Manchester University Press, Manchester, UK.

[40] Golub, G. H. and van der Vorst, H. A. (1996), Closer to the solution: Iterative linear solvers, *in* I. S. Duff and G. A. Watson, eds, 'The State of the Art in Numerical Analysis', Oxford University Press, Oxford, pp. 63–92.

[41] van der Vorst, H. A. and Golub, G. H. (1996), 150 years old and still alive: Eigenproblems, *in* I. S. Duff and G. A. Watson, eds, 'The State of the Art in Numerical Analysis', Oxford University Press, Oxford, pp. 93–119.