LAPACK: A Portable Linear Algebra Library for High-Performance Computers

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen

Abstract

The goal of the LAPACK project is to design and implement a portable linear algebra library for efficient use on a variety of high-performance computers. The library is based on the widely used LINPACK and EISPACK packages for solving linear equations, eigenvalue problems, and linear least-squares problems, but extends their functionality in a number of ways. The major methodology for making the algorithms run faster is to restructure them to perform block matrix operations (e.g., matrix-matrix multiplication) in their inner loops. These block operations may be optimized to exploit the memory hierarchy of a specific architecture. The LAPACK project is also working on new divide-and-conquer algorithms for certain eigenproblems, and new algorithms that yield higher relative accuracy for a variety of linear algebra problems.

1 Introduction

The University of Tennessee, the Courant Institute for Mathematical Sciences, the Numerical Algorithms Group Ltd., Rice University, Argonne National Laboratory, and Oak Ridge National Laboratory are developing a transportable linear algebra library in Fortran 77. The library is intended to provide a uniform set of subroutines to solve the most common linear algebra problems and to run efficiently on a wide range of high-performance computers.

The LAPACK library (shorthand for Linear Algebra Package) will provide routines for solving systems of simultaneous linear equations, least-squares solutions of overdetermined systems of equations, and eigenvalue problems. The associated matrix factorizations (LU,

Cholesky, QR, SVD, Schur, generalized Schur) will also be provided, as will related computations such as updating or reordering of the factorizations and condition numbers (or estimates thereof). Dense and banded matrices will be provided for, but not general sparse matrices. In all areas, similar functionality will be provided for real and complex matrices.

The new library will be based on the successful EISPACK [32, 25] and LINPACK [14] libraries, integrating the two sets of algorithms into a unified, systematic library. A great deal of effort has also been expended to incorporate design methodologies and algorithms that make the LAPACK codes more appropriate for today's high-performance architectures. The LINPACK and EISPACK codes were written in a fashion that, for the most part, ignored the cost of data movement. Most of today's high-performance machines, however, incorporate a memory hierarchy [16, 27, 34] to help disguise the difference in speed of memory accesses and vectorized floating-point operations. As a result, codes must be careful about reusing data in order not to run at memory speed instead of floating-point speed. LAPACK codes have been carefully restructured to reuse as much data as possible in order to reduce the cost of data movement. Further improvements are the incorporation of new and improved algorithms for the solution of eigenvalue problems [10, 19]. LAPACK is also designed to exploit the parallel processing capabilities of many high-performance machines, especially shared memory machines.

LAPACK is designed to be efficient and transportable across a wide range of computing environments, with special emphasis on modern high-performance computers. While we do not expect LAPACK codes to be optimal for all architectures, we anticipate high performance over a wide range of machines. By relying on the Basic Linear Algebra Subprograms (BLAS) [21, 15, 29] the codes can be "tuned" to a given architecture by efficient—and, in all likelihood, machine-dependent—implementations of these kernels. Machine-specific optimizations are limited to those kernels, and the user interface is uniform across machines. We shall also distribute test and timing routines to verify the installation of the LAPACK codes on a particular architecture and to allow for easy comparison with existing software.

We aim for the new library to be easily available. *Netlib* [17] has proven to be an extremely convenient mechanism for distributing software by electronic mail, and we shall make LAPACK available

through netlib, for users who want copies of selected routines. We shall also make arrangements for the complete package to be distributed on magnetic tape, for a nominal cost only. Finally we shall encourage vendors to incorporate LAPACK in their own mathematical libraries.

The rest of this paper is outlined as follows. Section 2 describes the BLAS and explains why their use can speed up algorithms. Section 3 describes block algorithms and shows in some detail how to reorganize Gaussian elimination and QR factorization. Section 4 contains benchmark results for linear equation solving and QR factorization on a variety of machines. Section 5 outlines our general approach to achieving high accuracy; in general, we have replaced absolute error bounds (either on the backward or forward error) with relative error bounds, which better respect the sparsity and scaling structure of the original problems. Section 6 discusses a new bidiagonal singular value decomposition (SVD) which computes singular values and vectors much more accurately than previously thought possible, and Section 7 describes how the Jacobi method (with a modified stopping criterion) is uniformly more accurate than QR-based algorithms for the symmetric positive definite eigenvalue problem and SVD. Section 8 reviews the target machines for which LAPACK is designed to run most efficiently. Section 9 discusses the overall structure of the LAPACK library and explains the choice of programming language and style. Finally, Section 10 outlines future plans to extend the library, including the challenges faced in adapting the codes to distributed-memory machines.

2 Basic Linear Algebra Subroutines

The original set of Basic Linear Algebra Subroutines [29], known as the BLAS, were first proposed in 1973. After some refinement of the proposal, they were used in the construction of LINPACK. These BLAS did operations only on vectors of data, such as a dot product or a saxpy (adding a scalar multiple of one vector to another). We refer to these vector-vector operations as Level 1 BLAS. The Level 1 BLAS permit efficient implementation on scalar machines, but the granularity is too low for effective use on most vector or parallel machines.

More recently, higher level BLAS have been specified that perform operations of higher granularity and so offer more opportunity for optimization on different architectures. The Level 2 BLAS [21] perform matrix-vector operations such as matrix-vector multiplication and rank-one updates. The Level 3 BLAS [15] perform matrix-matrix operations such as matrix-matrix multiplication, solving triangular systems with multiple right hand sides, and rank-k matrix updates.

To appreciate why these Level 2 and Level 3 BLAS with larger granularity offer better opportunities for efficiency, one must understand memory hierarchies. All machines (not just supercomputers) have a hierarchy of memory levels—for example, with registers at the top, followed by cache, main memory, and finally disk storage at the bottom. Toward the top of the hierarchy, memory is smaller, more expensive, and faster. Since operations such as multiplication and addition must be done at the top level, data has to move up through the various levels to the top to be processed, and then down again to be stored. The result is that data at higher levels is available only after some delay and may not be available at a rate fast enough to feed the arithmetic units. Clearly, an algorithm that minimizes the memory traffic in the hierarchy will run faster.

One way to measure the amount of this memory traffic is the ratio of flops (floating point operations) to memory references in an algorithm. The larger this ratio, the longer a piece of data may be kept at the top of the hierarchy on average. Let us use this measure to compare the three operations of saxpy (Level 1 BLAS), matrix-vector multiplication (Level 2 BLAS), and matrix-matrix multiplication (Level 3 BLAS), where all vector and matrices are of dimension n. Simple counting yields the ratios 2/3 for saxpy, 2 for matrix-vector multiplication, and $n/2$ for matrix-matrix multiplication. The large ratio for matrix-matrix multiply represents a surface-to-volume effect, doing $O(n^3)$ operations on $O(n^2)$ data. Hence, matrix-matrix multiplication offers greater opportunity for exploiting the memory hierarchy than the lower-level BLAS routines. Table 1 illustrates this fact with some benchmark results.

Table 1: Speed of the BLAS on various architectures (all values are in Mflops)

	Alliant FX/8 (8 processors)	IBM 3090/VF (1 processor)	CRAY-2S (1 processor)
Peak Speed	94	108	488
Level 1 BLAS	14	26	121
Level 2 BLAS	26	60	350
Level 3 BLAS	43	80	437

Another advantage of Level 3 BLAS is that they offer greater scope for exploiting parallel processors on shared memory machines.

Fortran implementations of all the BLAS are available; to get the full benefit, however, the BLAS should be optimized for each architecture. We encourage the computer manufacturers to perform these optimizations; the data in Table 1 are for such optimized implementations. We also expect that the LAPACK project will reveal the need for a few additional basic routines whose performance may need to be optimized for different architectures and may be regarded as extensions to the current set of BLAS (e.g., applying a sequence of plane rotations to a matrix).

3 Block Algorithms

To exploit the Level 3 BLAS, one usually must express the algorithm in terms of operations on submatrices, or "blocks," as compared to vector- or scalar-oriented operations [22, 24]. We have developed such block algorithms for LU factorization, Cholesky factorization, Bunch-Kaufman factorization of a symmetric indefinite matrix, QR factorization (with and without pivoting), the nonsymmetric eigenproblem (reduction to Hessenberg form and QR iteration), the symmetric eigenproblem (reduction to tridiagonal form and reduction of a symmetric-definite generalized problem to standard form), and SVD (reduction to bidiagonal form). See [18] for details of some of these algorithms. Work is continuing on block algorithms for generalized eigenproblems. In this section we illustrate two blocked algorithms: LU factorization and QR factorization.

3.1 LU Factorization

First we show how to organize the algorithm mainly in terms of matrix-vector (Level 2) operations. Then we derive the block algorithm using matrix-matrix (Level 3) operations.

We want to factorize an n by n nonsingular matrix A as the product PLU of a permutation matrix P (representing the row interchanges), a unit lower triangular matrix L, and an upper triangular matrix U.

Assuming for simplicity that $P = I$, we write the factorization in partitioned form as:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}$$

Let the numbers of rows and columns in the blocks of the partitioned matrices be $j-1$, b and $n-j-b+1$.

To derive an algorithm using Level 2 BLAS, we assume $n_b = 1$. Suppose that the first $j-1$ columns of L and U (that is, L_{11}, L_{21}, L_{31} and U_{11}) have already been computed. To show how the j-th column may be computed, we equate components of this column as follows:

1. $A_{12} = L_{11}U_{12}$. Hence the vector U_{12} can be obtained by solving a lower triangular system of equations (a Level 2 BLAS operation).

2. $A_{22} - L_{21}U_{12} = L_{21}U_{12}$ and
   $A_{32} - L_{31}U_{12} = L_{31}U_{12}$.
   First we update A_{22} and A_{32} (treating them as a single vector) by computing

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} \leftarrow \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{12}$$

   which is a rectangular matrix-vector multiplication (another Level 2 BLAS routine). At this point we can incorporate a row interchange so that the single element A_{22} (call it α) is the largest element in A_{22} and A_{32}. Since L is unit lower triangular, $L_{22} = 1$ and $U_{22} = \alpha$. Finally $L_{32} = \alpha^{-1} A_{32}$ (a Level 1 BLAS operation).

Thus each column of L and U can be computed in turn, by performing steps 1 and 2 in a loop with j running from 1 to n. Most of the work in this loop can be performed by calls to two Level 2 BLAS routines.

To derive the block form of the algorithm, we assume that $b > 1$. As before, suppose that L_{11}, L_{21}, L_{31} and U_{11} have already been computed. The same approach as before yields the following method for computing the next block of b columns of L and U.

1. $A_{12} = L_{11}U_{12}$: the matrix U_{12} can be obtained by solving a lower triangular system of equations with b right hand sides (a Level 3 BLAS operation).

2. $A_{22} - L_{21}U_{12} = L_{22}U_{22}$ and
   $A_{32} - L_{31}U_{12} = L_{32}U_{22}$.
   First we update A_{22} and A_{32} by a rectangular matrix-matrix multiplication (another Level 3 BLAS operation). Then we factorize the updated matrix as

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = P' \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}$$

   using the Level 2 BLAS form of the algorithm described earlier. The blocks A_{22} and A_{32} are treated together in order to allow the necessary row interchanges (represented by the matrix P). These row interchanges must then be applied to the rest of the matrix.

This is just one way in which LU factorization may be organized in terms of Level 2 or Level 3 BLAS operations. There are in fact at least three ways, involving different patterns of data movement and BLAS operations. Their relative performance is machine-dependent, but our experience to date has shown that the Level 3 BLAS forms do not vary much in performance (there are greater differences in performance between the Level 2 BLAS forms).

Table 1a shows the speed in megaflops on one processor of a Cray X-MP of:

- the LINPACK routine SGEFA

- our LAPACK routine SGETRF, calling assembly language BLAS
- a highly optimized assembly-language algorithm provided by Cray

Table 1a: LU factorization on a Cray X-MP			
n	LINPACK	LAPACK	Cray
25	18	18	65
100	55	105	163
500	129	203	213

Thus, for large matrices, the Level 3 BLAS code ran at over 95% of the speed of the highly optimized assembly-language code. This result shows that we can achieve 95% efficiency from the machine, using portable Fortran code, with machine-specific coding confined to the BLAS.

3.2 QR Factorization

As another example of a block algorithm, we consider an algorithm for computing the QR factorization

$$A = QR$$

of a dense matrix. Here an $m \times n$ (w.l.o.g. $m \geq n$) matrix A is decomposed into an orthogonal $m \times m$ matrix Q and an upper triangular $m \times n$ matrix R. This decomposition is one of the basic tools of numerical linear algebra; for applications, see [26]. The traditional algorithm for computing the QR factorization [26, p. 148] employs a sequence of Householder reductions

$$H = I - 2uu^T, \quad \|u\|_2 = 1. \tag{1}$$

Application of H to a given matrix A involves a matrix-vector multiplication $z \leftarrow A^T u$ and a rank-one update $A \leftarrow A - 2uz^T$. Each of these Level 2 BLAS operations requires $O(n^2)$ floating-point operations and uses $O(n^2)$ data.

To arrive at a block formulation of the Householder QR algorithm, we must be able to express a series of Householder reductions in a convenient closed form. Bischof and Van Loan [7] expressed the product

$$Q = H_1 \cdots H_b$$

of a series of $m \times m$ Householder matrices (1) in the so-called *WY representation*

$$Q = I + WY^T \tag{2}$$

where W and Y are $m \times b$ matrices. Schreiber and Van Loan [30] refined this representation by expressing $W = YT$ where T is a $b \times b$ upper triangular matrix. Schreiber and Van Loan called the resulting representation

$$Q = I + YTY^T \tag{3}$$

the *compact WY representation* since it requires only about half as much storage as the original WY representation (2) in the typical case where $m \gg b$. Compared to the traditional Householder algorithm, the accumulation of T requires $O(mb^2)$ extra flops and $\frac{b^2}{2}$ extra words for storage. Since typically $m \gg b$, this is a low-order term in the overall algorithmic complexity. The advantage of the compact WY representation is that the computation of $A \leftarrow QA$ now involves two matrix-matrix multiplications

$$Z \leftarrow A^T Y T \tag{4}$$

and a rank-b update

$$A \leftarrow A + Y Z^T \tag{5}$$

instead of a series of b matrix-vector multiplications and rank-one updates.

We can now express the block Householder QR algorithm in terms of the primitives *gencwy* (generate compact WY factor) and *appcwy* (apply compact WY factor):

$$[Y, T] \leftarrow gencwy(A)$$

returns the compact WY factors T and Y such that

$$A = (I + YTY^T) \begin{pmatrix} R \\ 0 \end{pmatrix}.$$

The primitive *gencwy* first computes the QR factorization of A using the traditional Householder QR algorithm and then accumulates T. Next,

$$A \leftarrow appcwy(Y, T, A)$$

performs the updates (4) and (5). Figure 1 shows the block Householder algorithm using the compact WY representation. Here A is partitioned as an $M \times N$ block matrix, and for simplicity we assume that all blocks are of the same size $b \times b$, so $m = Mb$ and $n = Nb$. We use the notation $A(i, j)$ to refer to block entry (i, j) and $A(i : j, k : l)$ to refer to the submatrix of A consisting of block row entries i to j and block column entries k to l.

for $i = 1$ **to** N **do**
  $[Y, T] \leftarrow gencwy\ (A(i : M, i))$
  $A(i : M, i : N) \leftarrow appcwy\ (Y, T, A(i : M, i : N))$
end for

Figure 1: The Block Householder QR Factorization Algorithm

This algorithm illustrates some important features of block algorithm. For one, the block algorithm may require more floating point operations than its unblocked counterpart. We invest more work in accumulating a block transformation, but this is more than made up for by the application of the transformation, which will run at close to optimum speed since it is not slowed down by excessive data movement overhead. This reasoning is true up to the point where adding more columns to a block transformation will not result in a faster update.

This relates to the subtle issue of partitioning a given matrix into blocks. The block partitioning resulting in the fastest execution of the code (the "optimal" block partitioning) is problem-dependent (we can use larger blocks for larger matrices), but it also depends on the architecture of a given machine. Furthermore, on multiprocessor machines, possibly conflicting issues of individual processor performance and overall load balancing must be reconciled. A discussion of these issues and a suggestion for a methodology to overcome this problem can be found in [6]. Determining optimal, or near optimal, block sizes for different environments is a major research topic for the LAPACK project.

The algorithm in Figure 1 constructs a block transformation and then immediately applies it to the remaining sub-matrix. We shall call this a *"right-looking algorithm"*. Notice that at each of the N stages

we are updating a submatrix of size $(m-(i-1)b+1) \times (n-(i-1)b+1)$. We can further reduce the amount of data movement by the following algorithm:

```
for i = 1 to N do
    for j = 1 to i - 1 do
        A(j : M, i) ← appcuy, (Y, A(j : M, i))
    end for
    [Y_i, T_i] ← gencuy (A(i : M, i))
end for
```

Figure 2: Left Looking Block Householder QR Factorization Algorithm

At each stage of this algorithm we are only modifying a $m \times b$ matrix. We shall call this algorithm the *"left-looking algorithm"*. Compared to the algorithm in figure 1, the pattern of writes is more localized, and this may result in a substantial reduction in the number of writes to lower levels in the memory hierarchy. This is particularly important in shared-memory multiprocessors where cache consistency is guaranteed by the use of "write-through" caches [27, 33]. On those architectures read accesses to cached data can be satisfied in one cycle, but write accesses are immediately flushed to memory. As a result, write accesses are much slower than read accesses.

4 Benchmarks

The first preliminary version of LAPACK software was released to test sites in April 1989. This software included routines for general, positive definite, and symmetric indefinite systems and for QR decomposition without pivoting.

In Tables 2-4 we present results for which most or all of the BLAS were optimized for the particular architecture. SGETRF is the LAPACK routine for triangular factorization of a general matrix with partial pivoting, SPOTRF performs Cholesky factorization of a positive definite symmetric matrix, and SGEQRF does QR factorization without pivoting. Also shown are SGEMV (matrix-vector multiply)

and SGEMM (matrix-matrix multiply), since these are the "speed limits" for the algorithms written in terms of the Level 2 BLAS and Level 3 BLAS, respectively. All codes were run in single precision (32 bits on the Convex and 64 bits on the Cray). All results are in Mflops.

Table 2: LAPACK on a Convex C210

Routine	Matrix Dimension				
	32	64	128	256	512
SGEMV	34	43	47	47	47
SGEMM	38	44	47	47	47
SGETRF	6	12	21	30	36
SPOTRF	8	20	33	40	44
SGEQRF	12	21	27	33	38

Table 2 gives the results for the Convex C210, with an algorithm block size of $nb=1$. Since matrix-vector and matrix-matrix multiply are equally fast on this machine in their current implementations, nothing is gained by going to the Level 3 BLAS.

Table 3: LAPACK on a CRAY Y-MP
1 Processor

Routine	Matrix Dimension					
	32	64	128	256	512	1024
SGETRF	40	108	195	260	290	304
SPOTRF	34	95	188	259	289	301
SGEQRF	54	139	225	275	294	301

Table 4: LAPACK on a CRAY Y-MP
8 Processors

Routine	Matrix Dimension					
	32	64	128	256	512	1024
SGETRF	32	90	205	375	1039	1974
SPOTRF	29	84	273	779	1592	2115
SGEQRF	50	133	328	807	1476	1937

Tables 3 and 4 give results for the CRAY Y-MP for 1 and 8 processors, respectively. Here $nb=64$ for SGETRF and SPOTRF and $n_b = 16$ for SGEQRF. The maximum speed of a single processor of a CRAY Y-MP is 333 Mflops. Thus, we see that for large-enough matrix dimensions, the single-processor code runs at at 90% efficiency. When all 8 processors are used, the code attains 73% to 80% efficiency.

5 Target Machines

The LAPACK library will be designed primarily to perform efficiently on machines with a modest number of processors (say, 1-100), each having a powerful vector-processing capability. These machines include all of the most powerful computers currently available and in use for general-purpose scientific computing: CRAY-2, CRAY X-MP, CRAY Y-MP, Fujitsu VP, IBM 3090/VF, NEC SX, Hitachi S-820, Alliant FX/80, Convex C-1, Convex C-2, Stardent, Sequent Symmetry, Encore Multimax, and BBN Butterfly. On conventional serial machines, the performance of the library is expected to be at least as good as that of the current LINPACK and EISPACK codes. Thus the library will be suitable across the whole range of machines from personal computers to supercomputers to experimental architectures.

We do not claim that the strategy of using Level 2 or Level 3 BLAS will necessarily attain optimal performance on all these machines; indeed, some algorithms can be structured in several different ways, all calling Level 3 BLAS, but with different performance characteristics. In such cases we shall choose the structure that provides the best "average" performance over the range of target machines. Currently we are limiting machine-dependent optimizations to the BLAS to retain portability across architectures. We encourage vendors to provide implementations of the BLAS kernels that are optimized for their particular architectures. While users will be free to develop their own versions of the LAPACK codes, we believe that the possible performance gain will be limited on the more conventional architectures.

On the more experimental architectures (in particular, distributed-memory machines), the restriction of optimization to the BLAS might be too limiting. In particular, it might be advantageous to introduce parallelism at the top-level of the algorithm instead of inside the BLAS. To aid users in experimenting on their particular architecture,

the LAPACK codes have been carefully designed in a modular fashion and with the objective of minimizing data movement. Since data movement is the key issue in distributed-memory as well as shared-memory machines, the LAPACK codes should be easily "tunable" to more experimental architectures.

Several of the algorithms we intend to implement [19] will require more than loop-based parallelism. These algorithms will rely upon the simplified SCHEDULE mechanism [20] to invoke parallelism. These ideas might also be used to express top-level parallelism in a portable fashion. We are also closely following the activities of the Parallel Computing Forum [23] which has been formed by computer vendors, software developers, national laboratories, and universities to exchange technical information and to document agreements on constructs for programming parallel applications for shared-memory multiprocessors.

6 High-Accuracy Linear Algebra Algorithms

So far we have concentrated on the speed of the algorithms in LAPACK. But a secondary objective of the LAPACK project is to develop new or improved algorithms that provide increased accuracy, specifically near-optimum relative accuracy (in a sense to be defined). To present our discussion, we shall need some notation.

We let H denote the problem for which we seek a solution; we denote the solution by $f(H)$. For example, $f(H)$ may denote the eigenvalues, eigenvectors, singular values, or singular vectors of the matrix H. If H denotes the pair (A, b), then $f(H)$ may denote the solution of the linear system $Ax = b$, perhaps in a least-squares sense if A is singular or not square. In general, $f(H)$ cannot be computed exactly and hence is approximated by an algorithm whose output we denote $\hat{f}(H)$. We also let ε denote the machine precision.

Analyzing the accuracy of an algorithm $\hat{f}$ for f consists of two parts. First, we use *perturbation theory*, where we bound the difference $f(H + \delta H) - f(H)$ in terms of δH. This part depends only on f and not the algorithm that approximates it. Second, we use *error analysis*, which attempts to show that the computed solution $\hat{f}(H)$ is close to $f(H + \delta H)$ for some bounded δH. Showing that $\hat{f}(H) = f(H + \delta H)$ for some bounded δH is called *backward error*

analysis, but is by no means the only way to proceed.

There is a great deal of choice in the measures we choose to bound errors and measure distances. In conventional error analysis as introduced by Wilkinson, we bound $\| f(H + \delta H) - f(H) \|$ in terms of $\| \delta H \|$, and show $\hat{f}(H) = f(H + \delta H)$ where $\| \delta H \| \leq O(\varepsilon) \| H \|$. Here, $\| \cdot \|$ denotes a norm, like the one-norm or Frobenius norm. Typically one proves a formula of the form $\| f(H + \delta H) - f(H) \| \leq \kappa(f, H) \cdot \| \delta H \| + O(\|\delta H\|^2)$, where $\kappa(H)$ is called the *condition number of H with respect to f*. In this formulation, it is easy to see that $\kappa(f, H)$ is simply the norm of the gradient of f at H: $\| \nabla f(H) \|$; other scalings are possible. Thus, combining the perturbation theory and error analysis, one can write

$$\| f(H + \delta H) - f(H) \| \leq O(\varepsilon) \kappa(f, H) \cdot \| H \| + O(\varepsilon \|\delta H\|^2)$$

The drawback of this approach is that it does not respect the structure of the original data. In particular, if the original data is sparse or graded (large in some entries, small in others), bounding δH only by norm can give very pessimistic results. A trivial example is solving a diagonal system of equations. Each component of the solution is computed to full accuracy by a single divide operation, but the conventional condition number is the ratio of the largest to smallest diagonal entries and may be arbitrarily large.

Instead of bounding δH by its norm $\| \delta H \|$, one may instead use the measure $relerr_H(\delta H) \equiv \max_{i,j} |\delta H_{ij}| / |H_{ij}|$, the largest relative change in any entry (we use the notation $relerr_H$ to indicate the dependence on H). This measure respects sparsity, since δH_{ij} must be zero if H_{ij} is zero, and also grading, since every entry is perturbed by an amount small compared to its magnitude. For example, in the case of diagonal linear equation solving, one can easily see that a perturbation δH of size $relerr_H(\delta H)$ in the matrix can only change the solution relatively by $relerr_H(\delta H)$ in each component, and that the algorithm is backward stable with $relerr_H(\delta H) \leq \varepsilon$. Thus, the new perturbation theory and error analysis with respect to $relerr_H(\delta H)$ accurately predict that each component of the solution is computed to full relative accuracy.

We have successfully developed new perturbation theory, algorithms, and error analysis for the measure $relerr_H(\delta H)$ for much of numerical linear algebra. We cannot always guarantee to solve problems as though we had a small $relerr_H(\delta H)$, but the algorithms can in all

cases monitor their accuracy and produce useful error bounds. The algorithms are usually small variations on conventional algorithms, perhaps with a slightly different stopping criterion, although the bidiagonal SVD algorithm has a quite new component. In all cases the algorithms run approximately as fast as their conventional counterparts, sometimes a little slower and sometimes a little faster. Since they are based on the conventional algorithms, all the techniques using the Level 3 BLAS apply to them.

This approach has been applied to linear equation solving [2], linear least-squares problems [3], the bidiagonal SVD [11, 9], the tridiagonal symmetric eigenproblem [28, 4], the dense symmetric positive definite eigenproblem [12], and the dense definite generalized eigenproblem [4, 12]. We have similar but slightly weaker results for the dense SVD and generalized SVD [12]. These algorithms either will be included directly in LAPACK or can be easily constructed by using LAPACK subroutines as "building blocks."

In the next two sections, we shall describe these new algorithms for the bidiagonal SVD and dense symmetric positive definite eigenproblem, respectively. The LAPACK routine SBDSQR implements the new bidiagonal SVD algorithm; the work on the symmetric eigenproblem is a more recent development, and software has not yet been prepared for inclusion in LAPACK.

7 The Bidiagonal SVD

A bidiagonal matrix is one that is nonzero only on the main diagonal and first superdiagonal. The problem of computing its SVD arises both as the final stage of computing the SVD of a general matrix [14] and in the symmetric positive definite tridiagonal eigenproblem [4]. We begin by reviewing the conventional perturbation theory, algorithm, and error analysis and then discuss the new approach. See [11, 9] for details and proofs.

The conventional perturbation theory is the same for the SVD of a bidiagonal matrix as for a dense matrix. Suppose B is the n by n bidiagonal matrix and δB a perturbation. Let σ_i, u_i and v_i be the singular values, unit left singular vectors, and unit right singular vectors of B, respectively, where we assume $\sigma_1 \geq \cdots \geq \sigma_n$. Let σ_i', u_i' and v_i' be the corresponding values for $B' = B + \delta B$. Then the

following results are well known [26, 8]:

$$\begin{aligned} |\sigma - \sigma_i'| &\leq \|\delta B_2\| \\ \max(\sin\theta(u, u_i'), \sin\theta(v, v_i')) &\leq \frac{\|\delta B_2\|}{gap - \|\delta B_2\|} \end{aligned} \tag{6}$$

where the *absolute gap* between singular values is

$$gap \equiv \min_{j \neq i} |\sigma - \sigma_i| \tag{7}$$

Here $\sin\theta(x, y)$ denotes the sine of the acute angle between the vectors x and y.

The conventional algorithm for the bidiagonal SVD is QR iteration, which can be shown to be backward stable: it computes the exact SVD of $B + \delta B$ where $\|\delta B_2\| = O(\varepsilon)\|B_2\|$. Thus, large singular values (those near $\sigma = \|B\|_2$) are computed with high relative accuracy, but small ones ($\sigma \ll \sigma_n$) may be totally inaccurate if $\varepsilon \geq \sigma_j$. Similarly, if there are at least two tiny singular values, $\sigma_i \ll \sigma_n$, then both gap_i and gap are small and the error bound on the singular vectors is large.

In contrast, the new perturbation theory uses the following measure:

$$\eta \equiv (2n - 1) \cdot \max_{ij} \left| \log \frac{B_{ij} + \delta B_{ij}}{B_{ij}} \right|$$

which is approximately $(2n - 1) r_B(\delta B)$ when both are small. One can prove [11, 9]

$$\begin{aligned} \frac{|\sigma - \sigma_i'|}{\sigma_i} &\leq e^{\eta} - 1 \\ \max(\sin\theta(u, u_i'), \sin\theta(v, v_i')) &\leq \frac{2^{1/2}\eta(1 + \eta)}{rel\_gap - \eta} \end{aligned} \tag{8}$$

where the *relative gap* between singular values is

$$rel\_gap \equiv \min_{j \neq i} \frac{|\sigma - \sigma_i|}{\sigma_j + \sigma_i} \tag{9}$$

One can easily see that bounds (8) are at least about as strong as their conventional counterparts (6). To see how much stronger they may be, consider making relative perturbations of size 10^{-10} in a 3

by 3 bidiagonal matrix with singular values $\sigma_1 = 1$, $\sigma_2 = 2 \cdot 10^{-20}$, and $\sigma_3 = 10^{-20}$. Note that $gap_3 = gap_2 = 10^{-20}$ and that $rel\ gap_3 = rel\ gap_2 = 1/3$. Since the norm of this perturbation is about 10^{-10}, we may apply (6) to get the absolute error bound $10^{-10} \gg \sigma_3$ for σ_3; and, since $10^{-10} \gg gap_2$, no error bound for the singular vectors at all. Applying (8), we get a relative error bound of about $5 \cdot 10^{-10}$ in σ_3. Thus, we have at least 9 decimal digits of accuracy in σ_3, whereas (6) predicts changes 10^{20} times larger. Applying (8) again, we get an error bound of about $2 \cdot 10^{-9}$ in the direction of the singular vectors, whereas (6) provides no error bound at all. The same results hold for σ_2 and its singular vectors.

In summary, *absolute* uncertainties in the entries of a general matrix yield *absolute* error bounds on its singular values, and error bounds depending on the *absolute gap* for its singular vectors. In contrast, *relative* uncertainties in the entries of a bidiagonal matrix yield *relative* error bounds on its singular values, and error bounds depending on the *relative gap* for its singular vectors.

We have developed a new bidiagonal SVD algorithm that computes singular values and singular vectors within the accuracy bounds of this perturbation theory. The algorithms are not backward stable in the usual sense, as indeed they cannot be, since setting an offdiagonal entry of B to zero (convergence) is a large relative change in an entry of B. Nonetheless, we can show that the combined effects of roundoff and the stopping criterion permit all singular values to be computed with relative accuracy $O(\varepsilon)$ and singular vectors and v_i to be computed with accuracy $O(\varepsilon)/rel_{\sigma_i} gap$

The algorithm is a hybrid of the conventional, shifted QR algorithm and a new, stable implementation of QR with a zero shift [11]. There is also a new stopping criterion (criterion for setting offdiagonal entries of B to zero). The zero-shift QR algorithm has the property that it computes each entry of the transformed matrix to high relative accuracy. It is used on deflated submatrices of B whose condition number $\sigma_{max}/\sigma_{min}$ is so large that the roundoff introduced by the standard shifted QR, $\varepsilon \cdot \sigma_{max}$, would make unacceptably large changes in the computed σ_{min}. Standard shifted QR is used on submatrices where $\sigma_{max}/\sigma_{min}$ is small.

In numerical tests, this new algorithm was approximately as fast and occasionally much faster than its conventional counterpart. Zero-shift QR is only linearly convergent, whereas shift QR is cubically

convergent; nevertheless, no speed is lost, because it is applied only when $\sigma_{min}/\sigma_{max}$, its convergence factor, is quite small. Thus, convergence is guaranteed to be fast, even though linear. See [11] for detailed results of numerical tests.

8 Jacobi's Method for the Symmetric Positive Definite Eigenproblem

We begin by reviewing the conventional perturbation theory, algorithms and error analysis for the symmetric positive definite eigenproblem, and then discuss the new approach. See [12] for details and proofs.

The conventional perturbation theory, algorithms and error analysis work equally well for all symmetric matrices. Let H be an n by n symmetric matrix and δH a perturbation. Let λ_i and v_i denote the eigenvalues and unit eigenvectors of H, respectively, where we assume $\lambda_1 \leq \cdots \leq \lambda_n$. Let λ_i' and v_i' be the corresponding values for $H' = H + \delta H$. Then the following results are well known [26, 8]:

$$\begin{aligned} |\lambda_i - \lambda_i'| &\leq \|\delta H\|_2 \\ \sin\theta(v_i, v_i') &\leq \frac{\|\delta H\|_2}{gap_i - \|\delta H\|_2} \end{aligned} \tag{10}$$

where the *absolute gap* between eigenvalues is

$$gap_i \equiv \min_{j \neq i} |\lambda_j - \lambda_i| \tag{11}$$

As in the last section, $\sin\theta(x, y)$ denotes the sine of the acute angle between the vectors x and y.

Any of the conventional algorithms for the symmetric eigenproblem, including QR iteration, Jacobi, bisection and inverse iteration, are known to be backward stable: they compute the exact eigendecomposition of $H + \delta H$, where $\|\delta H\| = O(\varepsilon)\|H\|$. Thus, just as with the conventional SVD algorithm, large eigenvalues (those near $\lambda_n = \|H\|_2$) are computed with high relative accuracy, but small ones ($\lambda_j \ll \lambda_n$) may be totally inaccurate if $\varepsilon \geq \lambda_j$. Similarly, if there are at least two tiny eigenvalues, $\lambda_i \ll \lambda_n$, then both gap_i and gap_j are small and the error bound on the eigenvectors is large.

In contrast, the new perturbation theory uses the measure $\eta = rel_H(\delta H)$ introduced in section 6, but applies only to positive definite matrices. To present the results, we rewrite H as $H = DAD$, where $D = \mathrm{diag}(H_{11}^{1/2}, \ldots, H_{nn}^{1/2})$ is diagonal and A has unit diagonal. It is known that A's condition number $\kappa(A) \equiv \|A\|_2 \cdot \|A^{-1}\|_2$ satisfies $\kappa(A) \leq \min_{\hat{D}} \kappa(\hat{D}H\hat{D})$, where the minimum is over all nonsingular diagonal matrices $\hat{D}$ [31]. In other words, A is nearly the best diagonal scaling of H. Then we may show [12]

$$\begin{aligned} \frac{|\lambda_i - \lambda_i'|}{\lambda_i} &\leq n \cdot \eta \cdot \kappa(A) \\ \sin\theta(v_i, v_i') &\leq \frac{(n-1)^{1/2} \cdot \eta \cdot \kappa(A)}{rel\_gap_i} + O(\eta^2) \end{aligned} \tag{12}$$

where the *relative gap* between eigenvalues is

$$rel\_gap_i = \min_{j \neq i} \frac{|\lambda_j - \lambda_i|}{(\lambda_j \cdot \lambda_i)^{1/2}} \tag{13}$$

There is also a nonasymptotic version of the eigenvector bound, but we omit it for simplicity of presentation.

Now we contrast the new with the old bounds. From (10) we get the conventional relative error bound

$$\frac{|\lambda_i - \lambda_i'|}{\lambda_i} \leq n \cdot \eta \cdot \kappa(H)$$

to contrast with (12). To see how much stronger (12) may be, consider the symmetric positive definite matrix $H = DAD$ where

$$H = \begin{bmatrix} 10^{40} & 10^{29} & 10^{19} \\ 10^{29} & 10^{20} & 10^{9} \\ 10^{19} & 10^{9} & 1 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & .1 & .1 \\ .1 & 1 & .1 \\ .1 & .1 & 1 \end{bmatrix}$$

and $D = \mathrm{diag}(10^{20}, 10^{10}, 1)$. Here $\kappa(H) \approx 10^{40}$ and $\kappa(A) \approx 1.33$. Thus, η relative perturbations in the matrix entries only cause 4η relative perturbations in the eigenvalues according to the new theorem, and $3 \cdot 10^{40} \cdot \eta$ relative perturbations according to the conventional theorem. Also, the absolute gaps for the eigenvalues of H are $gap_{2,3} \approx 10^{-20}$, 10^{20}, 1, whereas the relative gaps $rel\_gap_{1,2,3}$ are all approximately 10^{10}. Thus the new theory predicts errors in v_1 and v_2 of norm $2 \cdot 10^{10} \cdot \eta$, whereas the old theory predicts errors of $\eta 10^{20}$.

We can show that Jacobi's method (with a modified stopping criterion) will attain these new error bounds, but in general QR will not. For the example in the last paragraph, QR computes two out of the three eigenvalues as negative, whereas H is positive definite. In contrast, Jacobi computes all the eigenvalues to nearly full machine precision. In fact for this example we can show Jacobi computes all components of all eigenvectors to nearly full relative accuracy, even though they vary by 16 orders of magnitude; again QR will not even get the signs of many small components correct.

All these results may be extended to the dense SVD, except that the notion of perturbation used is that each column of δH must be small in norm compared to the corresponding column of H.

9 LAPACK Structure, Language, and Style

9.1 Contents

In this subsection, we outline the planned contents of the LAPACK library; see [5] for details.

For solving *linear equations*, LAPACK will perform triangular factorization, solution via forward and back substitution, iterative refinement [10, 2] and equilibration. It will handle general matrices (dense only), positive definite matrices (dense or banded), symmetric indefinite matrices (dense or banded), and triangular matrices (dense or banded). In addition, symmetric and triangular matrices may be stored in a packed format. It will also update triangular factorizations.

For solving *least squares*, LAPACK will do QR decomposition with and without pivoting, as well as least squares solutions using this decomposition. It will also compute the generalized QR decomposition of two matrices (this is needed for the generalized SVD below) and update the QR decomposition.

For solving *eigenproblems*, LAPACK will compute eigenvalues, the Schur form, and eigenvectors for general matrices or matrix pencils (i.e., the generalized eigensystem of $A - \lambda B$). For symmetric matrices (dense or banded) it will compute eigenvalues and eigenvectors. It will also handle the generalized symmetric eigenproblem $A - \lambda B$ for A and B symmetric (dense or banded) and B positive definite. For general matrices (dense or banded) and triangular matrices, it will

compute the SVD; and for pairs of matrices, it will compute the generalized SVD. It will also do rank-1 updates of the bidiagonal SVD and tridiagonal symmetric eigenproblem.

LAPACK will include condition estimators for linear systems, eigenvalues, eigenvectors, and invariant subspaces. It will solve the Sylvester equation and generalized Sylvester equation and can reorder the eigenvalues along the diagonal of a matrix in Schur form.

The nonsymmetric eigenroutines will be based on the QR algorithm. Inverse iteration will be available if only a few eigenvectors are desired. For the symmetric tridiagonal eigenproblem (the final phase of the dense symmetric eigenproblem) algorithms based on QR, divide-and-conquer [13], and bisection will be included; these algorithms have different attainable accuracies, require different amounts of storage, and run at different speeds on different architectures and depending on whether some or all eigenvalues and eigenvectors are desired. In other words, no single algorithm is best in all cases [10]. The same is true for the bidiagonal SVD, which is the final phase of the general SVD. We also plan to include Jacobi's method for the dense symmetric eigenproblem and SVD in a later release, for reasons given earlier.

9.2 Language and Style

The software is being developed in portable Fortran 77, with extensions to the standard only where necessary. Single- and double-precision versions will be prepared; conversion between different precisions will be performed automatically by software tools from Toolpack/1.

Routines for complex matrices will use the COMPLEX data type (like LINPACK, but unlike EISPACK); hence the availability of a double-precision complex (COMPLEX*16 or DOUBLE COMPLEX) data type will be assumed as an extension to Fortran 77. Routines for real and complex matrices will be written to maintain a close correspondence between the two, however, in some algorithms (e.g., unsymmetric eigenvalue problems) the correspondence will necessarily be weaker.

We realize that Fortran 90 is likely to have a number of features that would improve the design and coding of the library. In particular, its built-in array features would replace some of the BLAS kernels

and result in cleaner and easier-to-read code. Another useful feature is the dynamic allocation of workspace. Almost all block routines need workspace, with the optimal amount of storage determined depending on the problemparameters at run-time. Currently, the user will pass a work array in the argument list in the hope that it will be big enough; if it is not, the block size used may be less than optimal. In Fortran 90, workspace could be allocated dynamically at run-time in a fashion that is transparent to the user. This would significantly shorten calling sequences and avoid some common programming mistakes resulting frompassing too little workspace.

In a future project we plan to develop a Fortran 90 version of LAPACK, and also a C version, using tools for automatic transformation as far as possible. However, for the current phase of initial development and testing, Fortran 77 was the only reasonable choice.

10 Future Work

Our first preliminary software release in April 1989 distributed codes for linear equation solving and QR decomposition to over 65 test sites. Our second preliminary release occured in April 1990 and includes software for iterative refinement, the nonsymmetric eigenproblem, symmetric eigenproblem, and SVD. Banded eigenproblems, generalized eigenproblems and the generalized SVD, condition estimation for the eigenproblem, and updates of various decompositions will follow in 1990. We are working toward the first public release of LAPACK at the beginning of 1991.

For the longer term, we have identified a number of research directions. First, we are interested in extending our approach to distributed-memory machines. These are more challenging than the shared-memory machines we have been working on, because of the additional cost of communication between different processors and memories. Second, we would like to systematically develop parameterized software that is both portable and efficient. In section 3 we identified the block size, nb, as one such parameter. Third, we wish to identify features of computer architectures that either help or hinder production of good numerical software. Two examples of helpful features are the ability to access rows and columns of matrices with similar speeds, and friendly error recovery such as the overflow flag

in IEEE standard floating point arithmetic [1]. Finally, we wish to provide performance evaluation tools for new architectures.

Acknowledgments

The work is supported by NSF grant grant ASC-8715728 and by DARPA grant F49620-87-C0065.

References

[1] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[2] M Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.*, 10(2):165–190, April 1989.

[3] M Arioli, I. S. Duff, and P. P. M de Rijk. On the augmented system approach to sparse least-squares problems. *Num Math.*, 55:667–684, 1989.

[4] Jesse Barlow and James Demmel. Computing accurate eigensystems of scaled diagonally dominant matrices. Computer Science Dept. Technical Report 421, Courant Institute, New York, NY, December 1988. (to appear in SIAM J. Num Anal., also LAPACK Working Note #7).

[5] Chris Bischof, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. Lapack provisional contents. Mathematics and Computer Science Division Report ANL-88-38, Argonne National Laboratory, Argonne, IL, September 1988. (LAPACK Working Note #5).

[6] Christian H. Bischof. Adaptive blocking in the QR factorization. *The Journal of Supercomputing*, 3(3):193–208, 1989.

[7] Christian H. Bischof and Charles F. Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Stat and Sci Compt*, 8:s2–s13, 1987.

[8] Chandler Davis and W. Kahan. The rotation of eigenvectors by a perturbation iii. *SIAM Journal on Numerical Analysis*, 7:248–263, 1970.

[9] P. Deift, J. Demmel, L.-C. Li, and C. Tomei. The bidiagonal singular values decomposition and Hamiltonian mechanics. Computer Science Dept. Technical Report 458, Courant Institute, New York, NY, July 1989. (submitted to SIAM J. Num. Anal.).

[10] James Demmel, Jeremy Du Croz, Sven Hammarling, and Danny Sorensen. Guides for the design of symmetric eigenroutines, SVD, iterative refinement and condition estimation. Mathematics and Computer Science Division Report ANL/MCS-TM-111, Argonne National Laboratory, Argonne, IL, February 1988. (LAPACK Working Note #4).

[11] James Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. Computer Science Dept. Technical Report 326, Courant Institute, New York, NY, March 1988. (to appear in SIAM J. Sci. Stat. Comp.; also LAPACK Working Note #3).

[12] James Demmel and K. Veselić. Jacobi's method is most accurate. Computer science dept. technical report, Courant Institute, New York, NY, September 1989.

[13] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comp.*, 8(2):139–154, March 1987.

[14] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM Press, 1979.

[15] Jack Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A set of level 3 basic linear algebra subprograms. Technical Report MCS-P1-0888, Argonne National Laboratory, August 1988, To appear ACM TOMS March 1990.

[16] Jack Dongarra and Iain S. Duff. Advanced computer architectures. Technical Report CS-89-90, University of Tennessee, 1989.

[17] Jack Dongarra and Eric Grosse. Distribution of mathematical software by electronic mail. *Communications of the ACM*, 30(5):403–407, 1987.

[18] Jack Dongarra, Sven Hammarling, and Danny Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27, 1989.

[19] Jack Dongarra and Danny Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM Journal on Stat and Sci Compt*, 8(2):s139–s154, 1987.

[20] Jack Dongarra and Danny Sorensen. A portable environment for developing parallel programs. *Parallel Computing*, 5(1&2):175–186, 1987.

[21] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[22] Jack J. Dongarra and Danny C. Sorensen. Linear algebra on high-performance computers. In Udo Schendel, editor, *High-Performance Computers 85*, pages 3–32, Amsterdam, 1986. North-Holland.

[23] The Parallel Computing Forum. *PCF Fortran: Language Definition*. Kuck and Associates, Champaign, IL 61820, 1988.

[24] K. Gallivan, R. Plemmons, and A. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32:54–135, 1990.

[25] B. Garbow, J. Boyle, J. Dongarra, and C. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*, volume 51 of *Lecture Notes in Computer Science*. Springer Verlag, New York, 1977.

[26] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins Press, Baltimore, Maryland, 2nd edition, 1989.

[27] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.

[28] W. Kahan. Accurate eigenvalues of a symmetric tridiagonal matrix. Computer Science Dept. Technical Report CS41, Stanford University, Stanford, CA, July 1966 (revised June 1968).

[29] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[30] Robert Schreiber and Charles Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAMJ. Sci Stat Comp.*, 10(1):53–57, 1989.

[31] A. Van Der Sluis. Condition numbers and equilibration of matrices. *Numerische Mathematik*, 14:14–23, 1969.

[32] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*. Springer-Verlag, second edition, 1976.

[33] Per Stenström. Reducing contention in shared-memory multiprocessors. *IEEE Computer*, 21(11):26–37, 1988.

[34] Harold Stone. *High-Performance Computer Architecture*. Addison-Wesley, Reading, Massachussetts, 1987.