E.G. Coffman et al., Eds., *Handbooks in OR & MS, Vol. 3*

Chapter 6

A Survey of Matrix Computations

Charles Van Loan

Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, U.S.A.

Preface

This chapter is a three-level introduction to the field of matrix computations. The first section is very informal and is designed to acquaint the reader with the basic tools of the trade. The tools range from the algorithmic (What is a Householder matrix and how can it be used to solve the least square problem?) to the analytic (What happens to the solution of a least square problem if we perturb the data?).

In the second section we discuss the matrix factorizations that figure heavily in numerical linear algebra. For each factorization we survey algorithms, associated mathematical properties, and applications. Sprinkled throughout this section are special topics that illustrate the power of the factorization paradigm.

In the final section we use one factorization (Cholesky) to illustrate various aspects of high performance matrix computations. Successful computing nowadays requires the design of codes that pay careful attention to the flow of data during execution. Our goal is to make the reader more aware of these data movement concerns.

Of course, it is impossible to do justice to the matrix computation field in just a few pages and so we have been careful to include ample pointers to the literature. But before delving into the research papers it is useful to be aware of some of the standard references. Books that attempt to cover the area include Hager [1988], Stewart [1973], Watkins [1991], and Golub & Van Loan [1989]. The last reference is particularly comprehensive – virtually everything we discuss may be found in that volume.

There are several good books that focus on some of the major problem areas such as linear systems, least squares fitting, and the eigenvalue problem. These include Forsythe & Moler [1967], Lawson & Hanson [1974], Parlett [1980], Varga [1962], and Wilkinson [1965].

Sparse matrix problems are increasingly important and so we mention the texts by Björck, Plemmons & Schneider [1981], Duff, Erisman & Reid [1986], and George & Liu [1981].

For those interested in the development or use of matrix computation software we recommend Coleman & Van Loan [1988], Smith, Boyle, Ikebe, Klema & Moler [1970], Garbow, Boyle, Dongarra & Moler [1972], Dongarra, Bunch, Moler & Stewart [1978], and Wilkinson & Reinsch [1971]. A new package called 'LAPACK' is due to be released soon and will be a milestone in practical scientific computing. The project is funded by the National Science Foundation and so all the codes will be in the public domain. LAPACK has all the functionality of the LINPACK and EISPACK packages plus many additional capabilities. LINPACK covers linear equations and least squares while EISPACK handles the eigenproblem. All of the dense matrix computations mentioned in this chapter can be handled by codes from these two packages.

Another software tool that is both widespread and indispensible is MATLAB. MATLAB is an interactive system that facilitates the design and testing of matrix algorithms. The MATLAB language is rapidly becoming the model for exposition in the field as it permits a very high level of specification.

At the time of writing (1992), it is possible to obtain software for many of the algorithms in the book by sending electronic mail to any of the following three addresses:

netlib@ornl.gov

netlib@research.att.com

research!netlib

Typical messages include

send index	{For a list of available libraries.}
send index for linpack	{For a list of LINPACK codes.}
send svd from eispack	{For the EISPACK svd code.}

The matrix computation literature is widely scattered, however, some of the more important journals include *SIAM J. Scientific and Statistical Computing*, *SIAM J. Matrix Analysis*, *SIAM J. Numerical Analysis*, *SIAM Review*, *Numerische Mathematik*, *J. Linear Algebra and its Applications*, *Mathematics of Computation*, *ACM Transactions on Mathematical Software*, and *IMA J. on Numerical Analysis*. We also mention that many interesting matrix methods surface in the engineering literature, particularly the IEEE journals.

Finally, a word about the level of linear algebra expertise that is required to understand this chapter. The reader should certainly be familiar with the concepts of basis, independence, transpose, inner product, span, null space, range, and inverse. No less important are the notions of norm, orthogonality, and eigenvalue although with these topics we start with definitions. Readers who wish to upholster their linear algebra background should consult Halmos [1958], Leon [1980], or Strang [1988].

1. Some tools of the trade

This section is an informal introduction to the analytic and computational tools that underpin numerical linear algebra. Low dimension examples are the rule with appropriate generalizations to follow. The central themes include (a) the language of matrix factorizations, (b) the art of introducing zeros into a matrix, (c) the exploitation of structure, and (d) the distinction between problem sensitivity and algorithmic stability.

1.1. Gaussian elimination

Perhaps the most important problem in scientific computing is the problem of solving a system of linear equations. The method of *Gaussian elimination* proceeds by systematically removing unknowns from equations. The core calculation is the multiplication of an equation by a constant and its addition to another equation. For example, if we are given the system

$$\begin{aligned} 2x_1 - x_2 + 3x_3 &= 13, \\ -4x_1 + 6x_2 - 5x_3 &= -28, \\ 6x_1 + 13x_2 + 16x_3 &= 37, \end{aligned}$$

then we start by multiplying the first equation by 2 and adding it to the second equation. This removes x_1 from the second equation. Likewise we can remove x_1 from the third equation by adding to it, -3 times the first equation. With these two reductions we obtain

$$\begin{aligned} 2x_1 - x_2 + 3x_3 &= 13, \\ 4x_2 + x_3 &= -2, \\ 16x_2 + 7x_3 &= -2. \end{aligned}$$

We then multiply the (new) second equation by 4 and add it to the (new) third equation, obtaining

$$\begin{aligned} 2x_1 - x_2 + 3x_3 &= 13, \\ 4x_2 + x_3 &= -2, \\ 3x_3 &= 6. \end{aligned} \tag{1}$$

In reverse order these three equations imply

$$\begin{aligned} x_3 &= 6/3 = 2, \\ x_2 &= (-2 - x_3)/4 = -1, \\ x_1 &= (3 - 3x_3 + x_2)/2 = 3. \end{aligned}$$

1.2. LU factorization

This description of Gaussian elimination can be succinctly described in matrix terms. In particular, the process finds a lower triangular matrix L and an upper triangular matrix U so that $A = LU$. In the preceding example, we have

$$A = \begin{bmatrix} 2 & -1 & 3 \\ -4 & 6 & -5 \\ 6 & 13 & 16 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 3 \\ 0 & 4 & 1 \\ 0 & 0 & 3 \end{bmatrix} \equiv LU .$$

We call this the *LU factorization*. Notice that the subdiagonal entries in L are made up of the multipliers that arise during the elimination process. The diagonal elements of L are all equal to one and so we refer to L as a *unit lower triangular* matrix.

How does b fit into this matrix description of Gaussian elimination? Note that the transformed right-hand side in (1) is the solution to the lower triangular system

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & -0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 13 \\ -28 \\ 37 \end{bmatrix} .$$

Thus, the overall technique has three stages:

- Apply the elimination process to A obtaining the factorization $A = LU$.
- Solve the lower triangular system $Ly = b$.
- Solve the upper triangular system $Ux = y$.

It all fits together nicely since $Ax = (LU)x = L(Ux) = Ly = b$.

In matrix computations, this language of 'matrix factorizations' has assumed a role of great importance. It enables one to reason about algorithms at a high level which in turn facilitates generalization and implementation on advanced machines. Thus, it is more appropriate to regard Gaussian elimination as a procedure for computing the LU factorization than as an $Ax = b$ solver.

1.3. Solving triangular systems

As evidenced by the above, the need to solve a triangular linear system of equations is an important problem. For upper triangular systems, the unknowns are resolved in reverse order by a process known as *back substitution*. Lower triangular systems are solved via an analogous process called *forward substitution*. In this case the unknowns are resolved in forward order.

1.4. Work and notation of a 'flop'

Triangular system solving is a good setting for introducing *flop counting* as a method for anticipating performance. In an n-by-n unit lower triangular system

$Ly = b$, the kth unknown is prescribed by

$$y_k = b_k - \sum_{j=1}^{k-1} l_{kj} y_j .$$

This requires $k-1$ multiplies and $k-1$ adds for a total of $2(k-1)$ *flops*. A flop is a floating point arithmetic operation (as opposed to an integer or logical operation). It follows that y requires $n^2 - n$ flops to compute. Now typically n is sufficiently large to ignore low order terms and so it is perfectly reasonable to say that n-by-n forward substitution costs n^2 flops.

A more complicated flop count reveals that Gaussian elimination requires $\frac{2}{3}n^3$ flops to produce the L and U factors. Thus, when a linear system is solved via this algorithm, the arithmetic associated with the triangular system solves is dominated by the arithmetic required for the factorization

Flop counting is a useful way to anticipate the performance of a matrix calculation but it has two shortcomings. First, on a modern computer the efficiency of a matrix code is more often determined by the nature of memory traffic than by flops. See Section 3 for further details. Second, matrix computations usually involve a large amount of integer arithmetic subscripting and this can dominate the actual cost of the floating point arithmetic. Nevertheless, the counting of flops is useful so long as one remembers that the volume of floating point arithmetic is but one of several factors that determine program efficiency.

1.5. Residual versus error

Consider the following innocuous linear system first point out to me by C. Moler:

$$\begin{bmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.217 \\ 0.254 \end{bmatrix}. \tag{2}$$

Without going into algorithmic details, suppose we apply two different methods and get two different solutions:

$$x^{(1)} = \begin{bmatrix} 0.341 \\ -0.087 \end{bmatrix}, \qquad x^{(2)} = \begin{bmatrix} 0.999 \\ -1.00 \end{bmatrix}.$$

Which one is preferred? An obvious way to compare the two solutions is to compute the associated *residuals*:

$$b - Ax^{(1)} = \begin{bmatrix} 0.0000001 \\ 0 \end{bmatrix}, \qquad b - Ax^{(2)} = \begin{bmatrix} 0.001343 \\ 0.001572 \end{bmatrix}.$$

On the basis of residuals, it is clear that $x^{(1)}$ is preferred. However, we are confronted with a dilemma when we learn that the exact solution is given by

$$x^{(\text{exact})} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Thus, $x^{(1)}$ renders a small residual while $x^{(2)}$ is much more accurate. Reasoning in the face of such a dichotomy requires care. We may be in a situation where how well Ax predicts b is paramount. In that case, we are after a small residual solution. In other settings, accuracy is critical in which case the focus is on nearness to the true solution. The point here is twofold: (a) the notion of a 'good' solution can be ambiguous, and (b) the intelligent appraisal of algorithms requires sharper tools.

1.6. Problem sensitivity and nearness

In our 2-by-2 problem above, the matrix A is very close to singular. Indeed,

$$\tilde{A} = \begin{bmatrix} 0.780 & 0.563001095\ldots \\ 0.913 & 0.659 \end{bmatrix}$$

is exactly singular. Thus, an $O(10^{-6})$ perturbation of the data renders our problem insoluble. Our intuition tells us that difficulties should arise if our given $Ax = b$ problem is 'near' to a singular $Ax = b$ problem. In that case we suspect that small changes in the problem data, i.e., A and b, will induce relatively large changes in the solution. It is clear that we need to be able to quantify notions such as 'nearness to singularity' and 'problem sensitivity.'

Notice that these concerns have *nothing* to do with underlying algorithms. They are mathematical issues associated with the $Ax = b$ problem. However, as we now show, they do clarify what we can expect from an algorithm in light of *rounding errors*.

A finite amount of hardware is devoted to the storage of each *floating point number*. A floating point number has a mantissa and an exponent. We focus on the former and for clarity assume that we are working on a base-10 machine with d-digit mantissas. We define the *unit roundoff* to be the largest floating point number such that the floating point addition of 1 and $\mathbf{u}$ equals 1. For d-digit arithmetic, $\mathbf{u} \approx 10^{-d}$. In general, a relative error of order $O(\mathbf{u})$ attends each floating point operation. However, errors arise even before we commence computation for data must be stored. For example, if $d = 4$ then π would have the representation 0.3142×10^{1}. In general, if $fl(x)$ denotes the stored version of a real number x, then

$$fl(x) = x + \varepsilon, |\varepsilon| \leq \mathbf{u}.$$

Let us return to the $Ax = b$ problem. Any algorithm that requires the storage of A (e.g., Gaussian elimination) 'sees' only the perturbed problem

$$(A + E)\hat{x} = b + f, \tag{3}$$

where $|e_{ij}| \leq \mathbf{u}|a_{ij}|$ and $|f_i| \leq \mathbf{u}|b_i|$. Since $\hat{x}$ is the 'best we can do', we ask two fundamental questions: (a) how can we guarantee that $A + E$ is nonsingular, and (b) how close is $\hat{x}$ to the true solution x?

1.7. Norms: Quantifying error and distance

To answer these questions we need the concept of a *norm*. Norms are one vehicle for measuring distance in a vector space. For vectors $x \in \mathbb{R}^n$ the 1, 2, and infinity norms are of particular importance:

$$\|x\|_1 = |x_1| + \cdots + |x_n| ,$$
$$\|x\|_2 = \sqrt{x_1^2 + \cdots + x_n^2} ,$$
$$\|x\|_\infty = \max\{|x_1|, \ldots, |x_n|\} .$$

For matrices $A \in \mathbb{R}^{m \times n}$ we have

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}| , \qquad \|A\|_2 = \max_{\|x\|_2 = 1} \|Ax\|_2 ,$$
$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}| , \qquad \|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2} .$$

It follows that the matrix of rounding errors E in (2) satisfies $\|E\|_1 \leq \mathbf{u}\|A\|_1$.

In the discussions that follow, whenever the choice of norm is a side issue, we drop subscripts. In examples with an order-of-magnitude theme, we choose whichever of the above norms is most convenient bearing in mind that for small problems, they do not differ by much. However, in serious applications the choice of a norm can be critical. This is especially true with weighted norms, e.g., $\|x\|_D = \|Dx\|_2$. Here, D is a nonsingular diagonal weighting matrix that may reflect our knowledge of underlying measurements.

1.8. Condition of linear systems

The $Ax = b$ sensitivity issues posed turn out to involve the *condition number*. In particular, if $A \in \mathbb{R}^{n \times m}$ then we say that

$$\kappa(A) = \|A\| \, \|A^{-1}\|$$

is the condition number of A *with respect to inversion*. If A is singular, then we assume $\kappa(A) = \infty$. Note that the condition number involves a choice of norm which we can stress through subscripting: $\kappa_1(A) = \|A\|_1 \, \|A^{-1}\|_1$. For $p = 1, 2, \infty$ we have $\kappa_p(A) \geq 1$.

It turns out that the perturbed system (3) is nonsingular if

$$\mathbf{u}\kappa(A) < 1 ,$$

meaning that A is not too badly conditioned with respect to the unit roundoff.

The condition number also figures in the bounding of the relative error in the stored system solution $\hat{x}$. In particular, if A and $A + E$ are nonsingular, then

$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \mathbf{u}\kappa(A) .$$

From this we reason that in the absence of additional information we have no right to expect an $Ax = b$ solver to obtain an accurate solution to an ill-conditioned linear system.

1.9. LU with pivoting

We return to Gaussian elimination. It can be shown that the computed solution $\tilde{x}$ satisfies a perturbed system $(A + \Delta A)\tilde{x} = b$, where $\|\Delta A\| \approx \mathbf{u}\| |\tilde{L}| |\tilde{U}| \|$. Here, $\tilde{L}$ and $\tilde{U}$ are the computed L and U. These quantities can be arbitrarily large as suggested by the 2-by-2 factorization

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1/\varepsilon & 1 \end{bmatrix} \begin{bmatrix} \varepsilon & 1 \\ 0 & 1 - \dfrac{1}{\varepsilon} \end{bmatrix} .$$

If we use 16-digit floating point arithmetic to solve

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 + \varepsilon \\ 2 \end{bmatrix} ,$$

which has exact solution $x = [1 \quad 1]^{\mathrm{T}}$, then in Table 1 is what we find for various ε. The troubling thing with this example is that the matrix A is well conditioned, e.g., $\kappa_2(A) = 1 + \mathrm{O}(\varepsilon)$. This means that Gaussian elimination introduces errors way above the level predicted by the mathematical sensitivity of the problem. This is because of the very small, $\mathrm{O}(\varepsilon)$, pivot that arises in the first step.

A simple way around this difficulty is to introduce *row interchanges*. For our example above, this means we apply Gaussian elimination to compute the LU

Table 1

ε	$\|x - \tilde{x}\| / \|x\|$
10^{-3}	$6 \cdot 10^{-16}$
10^{-6}	$2 \cdot 10^{-11}$
10^{-9}	$9 \cdot 10^{-8}$
10^{-12}	$9 \cdot 10^{-5}$
10^{-15}	$7 \cdot 10^{-2}$

factorization of A with its rows reversed

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1+\varepsilon \end{bmatrix}.$$

The resulting algorithm is an example of Gaussian elimination with *partial pivoting*. A full precision answer is obtained. (Note that A is well conditioned.)

In general, partial pivoting involves a linear search at each elimination step. When x_k is being removed we search for the biggest coefficient for x_k in the remaining equations. That equation is then interchanged with the current kth equation. The resulting factorization has the form $PA = LU$, where P is a permutation matrix. A *permutation matrix* is the identity with its rows permuted. To solve $Ax = b$ we solve $Ly = Pb$ and then $Ux = y$. The application of P to b is an $O(n)$ operation.

1.10. Backwards stability

We say that a linear equation solver is *backwards stable* if it produces an $\hat{x}$ that solves a nearby problem exactly, i.e., $(A + E)\hat{x} = b + f$ with $\|E\| \approx \mathbf{u}\|A\|$ and $\|f\| \approx \mathbf{u}\|b\|$. This turns out to be the case for Gaussian elimination with pivoting. Two important heuristics follow from this:

$$\|A\tilde{x} - b\| \approx \mathbf{u}\|A\|\,\|\tilde{x}\|\,, \qquad \frac{\|\tilde{x} - x\|}{\|x\|} \approx \mathbf{u}\kappa(A)\,.$$

The first result essentially says the algorithm produces small relative residuals *regardless* of problem condition. The second heuristic shows the accuracy *does* depend upon problem condition. If a standard norm (such as $\|\cdot\|_\infty$) is involved and the unit roundoff and condition satisfy $\mathbf{u} \approx 10^{-q}$, and $\kappa_\infty(A) \approx 10^{p}$, then $\tilde{x}$ has approximately $q - p$ correct digits.

For the relative error bound to be of practical interest, we need an estimate of the condition. The central problem of *condition estimation* is to obtain a good estimate in $O(n^2)$ flops assuming the availability of a factorization such as $A = LU$ or $PA = LU$. This turns out to be possible, see Cline, Molder, Stewart & Wilkinson [1979], Cline, Conn & Van Loan [1982], Grimes & Lewis [1981], Hager [1984], Higham [1987], and Van Loan [1987].

1.11. Banded systems

We now turn our attention to the exploitation of structure, a very important theme in matrix computations. A matrix $A \in \mathbb{R}^{n\times n}$ has *upper bandwidth* q and *lower bandwidth* r if a_{ij} is zero whenever $j > i + q$ or $i > j + r$. Thus,

$$A = \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ 0 & \times & \times & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}$$

has upper and lower bandwidth equal to 1. In this case we also say that A is *tridiagonal*. It is often possible to exploit band structure during a factorization. For example, the L and U factors of a tridiagonal are bidiagonal:

$$L = \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 \\ 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & \times & \times \end{bmatrix}, \qquad U = \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 \\ 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \times \end{bmatrix}.$$

If pivoting is performed, then L is not lower bidiagonal but it does have the property that there is at most one nonzero below the diagonal in each column. The number of flops required to factor a matrix with upper and lower bandwidth equal to p is $O(np^2)$.

Further details concerning the manipulation of band matrices may be found in Golub & Van Loan [1989].

1.12. Symmetric systems

Symmetry is another important property that can be exploited in matrix computations. For example, if A is symmetric ($A = A^T$), then the amount of data that defines $x = A^{-1}b$ is halved. Let us see how this reduction is reflected in the LU factorization. If $A = LU$ is a nonsingular symmetric matrix with an LU factorization, then $LU = U^T L^T$ implies that $U^{-T}L = L^{-T}U \equiv D$ is diagonal. Thus, we can rewrite the LU factorization as $A = L(DL^T)$:

$$\begin{bmatrix} 4 & -8 & -4 \\ -8 & 18 & 14 \\ -4 & 14 & 25 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 & -2 & -1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}.$$

Once obtained, the LDL^T factorization can be used to solve $Ax = b$ by solving $Ly = b$, $Dz = y$, and $Ux = z$.

This factorization need not be stable for the same reasons that the LU without pivoting may be unstable. Pivoting can rectify this but if we compute $PA = LU$ we can no longer expect U to be a diagonal scaling of L^T since PA is no longer symmetric. A possible way around this is to perform *symmetric pivoting*. In particular, we seek a permutation P so that the LDL^T factorization of PAP^T is stable. But notice that the diagonal of this matrix is a permutation of A's diagonal. It follows that if all of A's diagonal entries are small, then huge pivots can arise regardless of the choice of P.

There are two ways around this problem. In the approach of Bunch & Kaufman [1977] a permutation is found so that $PAP^{\mathrm{T}} = LDL^{\mathrm{T}}$, where L is unit lower triangular and D is a direct sum of 1-by-1 and 2-by-2 matrices, e.g.,

$$PAP^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ \times & 0 & 1 & 0 \\ \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & 0 & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & 0 & \times \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ \times & \times & 1 & 0 \\ \times & \times & \times & 1 \end{bmatrix}^{\mathrm{T}}.$$

The pivot strategy which determines P is rather complicated and involves a 2-column search at each step. The overall process is as stable as Gaussian elimination with pivoting and involves half the flops. Once obtained, the Bunch–Kaufman factorization can be used to solve a linear system in $O(n^2)$ flops as follows: $Lw = Pb$, $Dz = w$, $L^{\mathrm{T}}v = z$, $x = P^{\mathrm{T}}v$.

An alternative method for the stable factorization of an indefinite symmetric matrix is due to Aasen [1971] who developed an $\frac{1}{3}n^3$ algorithm for the factorization $PAP^{\mathrm{T}} = LTL^{\mathrm{T}}$, where T is tridiagonal and L is unit lower tridiagonal:

$$PAP^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ \times & \times & 1 & 0 \\ \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & 0 & 0 \\ \times & \times & \times & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \\ \times & \times & 1 & 0 \\ \times & \times & \times & 1 \end{bmatrix}^{\mathrm{T}}.$$

Once obtained, the Aasen factorization can be used to solve a linear system as follows: $Lw = Pb$, $Tz = w$, $L^{\mathrm{T}}v = z$, $x = P^{\mathrm{T}}v$. $O(n^2)$ flops are required.

1.13. *Positive definiteness*

A matrix $A \in \mathbb{R}^{n\times n}$ is *positive definite* if $x^{\mathrm{T}}Ax > 0$ for all nonzero $x \in \mathbb{R}^n$. Symmetric positive definite matrices constitute a particularly important class and fortunately they submit to a particularly elegant factorization called the *Cholesky factorization*. In particular, if A is symmetric positive definite, then we can find a lower triangular G such that $A = GG^{\mathrm{T}}$. In the 2-by-2 case if we equate entries in the equation

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} g_{11} & 0 \\ g_{21} & g_{22} \end{bmatrix} \begin{bmatrix} g_{11} & 0 \\ g_{21} & g_{22} \end{bmatrix}^{\mathrm{T}},$$

we obtain the equations

$$\begin{aligned} a_{11} &= g_{11}^2 && \Rightarrow\ g_{11} = \sqrt{a_{11}}\,, \\ a_{21} &= g_{21}g_{11} && \Rightarrow\ g_{21} = a_{21}/g_{11}\,, \\ a_{22} &= g_{21}^2 + g_{22}^2 && \Rightarrow\ g_{22} = \sqrt{a_{22} - g_{21}^2}\,. \end{aligned}$$

Using the definition of positive definite it is not hard to show that the two square roots are square roots of positive numbers, thereby ensuring the production of a real G. Furthermore, $|g_{ij}| \leq \sqrt{a_{ii}}$ for all i and j, ensuring that no large numbers surface during the Cholesky reduction. Thus, pivoting is not necessary and Cholesky is backwards stable.

1.14. Orthogonality and the QR factorization

Orthogonality is a central concept in matrix computations. A set of vectors $\{x_1, \ldots, x_k\}$ is *orthogonal* if $x_i^T x_j = 0$ whenever $i \neq j$. Thus,

$$\left\{ \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix}, \begin{bmatrix} 4 \\ 5 \\ 2 \end{bmatrix} \right\}$$

is an orthogonal set. If in addition $x_i^T x_i = 1$, then they are referred to as *orthonormal*.

The computation of orthogonal bases via the *Gram–Schmidt* process is well known. If a_1 and a_2 are given vectors and we set

$$x_1 = a_1 , \qquad q_1 = x_1 / \|x_1\|_2 ,$$
$$x_2 = a_2 - (q_1^T a_2) q_1 , \qquad q_2 = x_2 / \|x_2\|_2 ,$$

then q_1 is orthogonal to q_2 and $\operatorname{span}\{a_1, a_2\} = \operatorname{span}\{q_1, q_2\}$.

For the data

$$a_1 = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}, \qquad a_2 = \begin{bmatrix} 8 \\ -1 \\ 14 \end{bmatrix},$$

we obtain

$$x_1 = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}, \qquad q_1 = \tfrac{1}{3} \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix},$$
$$x_2 = \begin{bmatrix} 8 \\ -1 \\ 14 \end{bmatrix} - \tfrac{34}{9} \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}, \qquad q_2 = \tfrac{1}{3} \begin{bmatrix} -2 \\ -1 \\ 2 \end{bmatrix}.$$

This reduction can be expressed as a factorization:

$$\begin{bmatrix} 1 & -8 \\ 2 & -1 \\ 2 & 14 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & -\frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} \\ \frac{2}{3} & \frac{2}{3} \end{bmatrix} \begin{bmatrix} 3 & 6 \\ 0 & 15 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & -\frac{2}{3} & -\frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \end{bmatrix} \begin{bmatrix} 3 & 6 \\ 0 & 15 \\ 0 & 0 \end{bmatrix}.$$

We call this the *QR factorization* and it has an important role to play in least squares and eigenvalue computations.

The computations of the QR factorization is usually obtained through successive orthogonal transformations of the data. In particular, a product of 'simple' orthogonal matrices $Q = Q_1 \cdots Q_N$ is found so that $Q^T A = R$ is upper triangular. Two classes of such simple transformations are available to use: Householder reflections and Givens rotations.

1.15. Householder reflections

A *Householder reflection* has the form

$$H = I - 2vv^T, \quad v \in \mathbb{R}^n, \|v\|_2 = 1 .$$

It is easy to confirm that H is orthogonal.

Suppose $x^T = [2 \quad -1 \quad 2]$ and we set $v = u/\|u\|_2$ where

$$u = \begin{bmatrix} 5 \\ -1 \\ 2 \end{bmatrix} = x + \begin{bmatrix} \|x\|_2 \\ 0 \\ 0 \end{bmatrix}.$$

It is easy to verify that if

$$H = I - 2vv^T = \tfrac{1}{15}\begin{bmatrix} -10 & 5 & -10 \\ 5 & 14 & 2 \\ -10 & 2 & 11 \end{bmatrix},$$

then

$$Hx = \begin{bmatrix} -3 \\ 0 \\ 0 \end{bmatrix}.$$

This illustrates how a Householder matrix can be called upon to zero all but the top component of a vector x. The *Householder vector* v has the form $x \pm \|x\|_2 e_1$, where e_1 is the first column of I_n. (The sign is chosen so that no cancellation ensues when forming the first component of v.)

Householder vectors are used to zero designated sub-columns and sub-rows in a matrix. For example, if

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \\ 0 & a_{42} & a_{43} \\ 0 & a_{52} & a_{53} \end{bmatrix},$$

and we wish to zero the subdiagonal portion of the second column, then we determine a Householder vector $v \in \mathbb{R}^4$ so that the corresponding Householder

matrix $H \in \mathbb{R}^{4\times 4}$ does the following:

$$H\begin{bmatrix} a_{22} \\ a_{32} \\ a_{42} \\ a_{52} \end{bmatrix} = \begin{bmatrix} \tilde{a}_{22} \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

If we define the 5-by-5 orthogonal matrix:

$$\tilde{H} = \begin{bmatrix} 1 & 0 \\ 0 & H \end{bmatrix},$$

then

$$HA = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} \\ 0 & 0 & \tilde{a}_{33} \\ 0 & 0 & \tilde{a}_{43} \\ 0 & 0 & \tilde{a}_{53} \end{bmatrix}.$$

Here, the tildes denote updated entries. Note that $\tilde{H}$ is a Householder matrix itself associated with the vector $\tilde{v} = [0 \quad v]^{\mathrm{T}}$.

The product HA of an m-by-m Householder matrix and an n-by-k matrix involves $\mathrm{O}(mk)$ flops if we invoke the formula

$$HA = (I - 2vv^{\mathrm{T}})A = A - vw^{\mathrm{T}}, \quad w^{\mathrm{T}} = 2v^{\mathrm{T}}A.$$

We see that the new A is a rank-one update of the old A.

Householder updates are backwards stable. This means that if we compute an update of the form $A \leftarrow HA$, then the result is an exact Householder update of a matrix near to the original A.

A 4-by-3 matrix can be upper triangularized by a product $Q = H_1H_2H_3$ of Householder matrices as follows:

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \overset{H_1}{\rightarrow} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \overset{H_2}{\rightarrow} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \overset{H_3}{\rightarrow} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \end{bmatrix}.$$

This progressive introduction of zeros via Householder reflections is typical of several important factorization algorithms. These computations are backwards stable, e.g., the computed $\hat{R}$ satisfies $Q^{\mathrm{T}}(A + E)$ where Q is exactly orthogonal and $\|E\| \approx \mathbf{u}\|A\|$.

Various aspects of Householder manipulation are included in Golub & Van Loan [1989], Parlett [1971], and Tsao [1975].

1.16. Givens rotations

A 2-by-2 *Givens rotations* is an orthogonal matrix of the form

$$G(\theta) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \quad c = \cos(\theta), s = \sin(\theta).$$

If $x = [x_1 \quad x_2]^T$, then it is possible to choose (c, s) such that if $y = G^T x$, then $y_2 = 0$. Indeed, all we have to do is set

$$c = x_1/\sqrt{x_1^2 + x_2^2}, \qquad s = -x_2/\sqrt{x_1^2 + x_2^2}.$$

However, a preferred algorithm for computing the (c, s) pair would more likely resemble the following in which we assume that $x \neq 0$:

if $|x_1| > |x_2|$
  $\tau = x_2/x_1$; $c = 1/\sqrt{1+\tau^2}$; $s = c \cdot \tau$
else
  $\tau = x_1/x_2$; $s = 1/\sqrt{1+\tau^2}$; $c = s \cdot \tau$
end

In this alternative, we guard against the squaring of arbitrarily large numbers and thereby circumvent the problem of *overflow*. Overflow occurs when a floating operation leads to a result with an oversized exponent. Another benefit of this (c, s) determination is that square roots are always taken of numbers in this range $[1, 2]$. This restriction is sometimes helpful in the design of VLSI squareroot circuitry.

Givens rotations can be used to introduce zeros in a very selective fashion. Suppose that we want to compute the QR factorization of the structured matrix

$$A = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \\ v_1 & v_2 & v_3 \end{bmatrix}.$$

We first determine (c, s) so

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} r_{11} \\ v_1 \end{bmatrix} = \begin{bmatrix} \tilde{r}_{11} \\ 0 \end{bmatrix}.$$

We say that

$$G(1,4) = \begin{bmatrix} c & 0 & 0 & s \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s & 0 & 0 & c \end{bmatrix}$$

is a Givens rotation in the (1, 4) plane. Note that

$$G(1,4)^{\mathrm{T}}A = \begin{bmatrix} \tilde{r}_{11} & \tilde{r}_{12} & \tilde{r}_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \\ 0 & \tilde{v}_2 & \tilde{v}_3 \end{bmatrix}.$$

Here, the tildes denote updated matrix elements and the computation would continue with rotations in the (2, 4) and (3, 4) planes designed to remove $\tilde{v}_2$ and $\tilde{v}_3$. Note that 3 flops are associated with each updated element and so when a Givens update of the form $A \leftarrow G^{\mathrm{T}}A$ is performed, $O(n)$ work is involved where n is the number of columns. Similar comments apply to postmultiply updates of the form $A \leftarrow AG$.

Givens updates are backwards stable and various aspects of their implementation are discussed by Steward [1976a], Gentleman [1973], and Hammarling [1974]. See also Golub & Van Loan [1989].

1.17. The least squares problem

Given $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ with $m \geq n$, consider the problem of finding $x \in \mathbb{R}^n$ so that $Ax = b$. This is an *overdetermined* system of equations and one must prepare for the fact that there may be no solution. However, if we seek an x so that $Ax \approx b$, then a number of possibilities arise. The *least squares* approach involves the minimization of $\|Ax - b\|_2$. Because the 2-norm is preserved under orthogonal transformations, we see that if Q is orthogonal, then

$$\|Ax - b\|_2 = \|Q^{\mathrm{T}}(Ax - b)\|_2 = \|(Q^{\mathrm{T}}A)x - (Q^{\mathrm{T}}b)\|_2 .$$

Thus, the given least squares (LS) problem based upon (A, b) transforms to an equivalent LS problem based upon $(Q^{\mathrm{T}}A, Q^{\mathrm{T}}b)$. The transformed problem takes on a special form if $A = QR$ is the QR factorization. For example, in the $(m, n) = (4, 2)$ case that means

$$\|A_x - b\|_2 = \left\| \begin{bmatrix} r_{11}x_1 + r_{12}x_2 - c_1 \\ r_{22}x_2 - c_2 \\ -c_3 \\ -c_4 \end{bmatrix} \right\|_2 ,$$

where $Q^{\mathrm{T}}b = c$ is the transformed right-hand side. It follows that the optimum choice for x is that which solves the 2-by-2 upper triangular system

$$\begin{bmatrix} r_{11} & r_{12} \\ 0 & r_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.$$

Trouble arises in the QR solution of the LS problem if the columns of A are

dependent. If this is the case, then one of the r_{ii} is zero and the back substitution process breaks down. However, it is possible to incorporate *column interchanges* during the Householder reduction with an eye towards maximizing $|r_{kk}|$ during the kth step. As a result of this, a set of independent columns is 'pushed up front'. For example, in a 4-by-3, rank 2 problem a factorization of the form

$$Q^{\mathrm{T}}A\Pi = \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

would be obtained. In this situation, the LS problem has an infinite number of solutions but one would be selected that attempts to predict b from the first two columns of $A\Pi$.

Of course, it is not likely that exact zeros would surface during the computation. This leads to the difficult problem of *numerical rank determination*. For example, suppose in the 4-by-3 problem above we emerge with

$$Q^{\mathrm{T}}A\Pi = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 10^{-6} & 10^{-7} \\ 0 & 0 & 10^{-9} \\ 0 & 0 & 0 \end{bmatrix}.$$

Whether or not we regard this as a matrix of rank 1, 2, or 3 depends upon (a) the precision of the underlying floating point arithmetic, and (b) the accuracy of the original a_{ij}. For example, if A is exact and $\mathbf{u} = 10^{-18}$, then we are on fairly firm ground to believe that $\mathrm{rank}(A) = 3$. On the other and, if $\mathbf{u} = 10^{-8}$ then a case can be made for $\mathrm{rank}(A) = 2$. If the a_{ij} are just correct to 3 digits, then it might be best to regard A as having rank 1. This spectrum of possibilities reveals that the intelligent handling of rank determination is tricky and not without a subjective component. The stakes are high too because radically different LS solutions result for different choice of numerical rank.

1.18. Eigenvalue problems

Orthogonal transformations are also useful in eigenvalue problems. In the simplest setting we have $A \in \mathbb{R}^{n\times n}$ and seek scalars λ so that $A - \lambda I$ is singular. These scalars are called *eigenvalues* and they are the roots of the *characteristic polynomial* $p(\lambda) = \det(A - \lambda I)$. It follows that an n-by-n real matrix has n (possibly complex) eigenvalues. We denote the set of eigenvalues by $\lambda(A)$. If $\lambda \in \lambda(A)$, then there exists a nontrivial vector called an *eigenvector* so that $Ax = \lambda x$. There is also a *left eigenvector* y so that $y^{\mathrm{T}}A = \lambda y^{\mathrm{T}}$.

Note that if T is nonsingular and $Ax = \lambda x$, then $(T^{-1}AT)(T^{-1}x) = \lambda(T^{-1}x)$. This shows that the *similar* matrices A and $T^{-1}AT$ have the same eigenvalues. The matrix T is called a *similarity transformation*.

From several points of view, orthogonal similarity transformations $A \leftarrow Q^T A Q$ are extremely attractive. They are backwards stable when based upon Householder and Givens transformations, i.e., the computed $Q^T A Q$ is orthogonally similar to $A + E$, where $\|E\| \approx \mathbf{u}\|A\|$. The inverse is easily obtained and well-conditioned: $Q^{-1} = Q^T$, $\kappa_2(A) = 1$.

An important family of eigenvalue methods involve computing an orthogonal Q such that the eigenvalues of $Q^T A Q$ are 'displayed'. For example, if $A \in \mathbb{R}^{2\times 2}$ has real eigenvalues, then it is possible to find a cosine–sine pair (c, s) such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} \lambda_1 & t \\ 0 & \lambda_2 \end{bmatrix}.$$

This is the $n = 2$ version of the *Schur decomposition*. Note that λ_1 and λ_2 are the eigenvalues of A. If y is an eigenvector of T, then $x = Qy$ is an eigenvector of A. It is easy to verify that

$$y_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

and

$$y_2 = \frac{1}{\sqrt{1+\mu^2}} \begin{bmatrix} \mu \\ -1 \end{bmatrix}, \quad \mu = \frac{t}{\lambda_2 - \lambda_1},$$

are a pair unit 2-norm eigenvectors of T associated with λ_1 and λ_2, respectively. Note that if $t \neq 0$ and $\lambda_1 = \lambda_2$, then y_2 is not defined. In this case we see that T (and hence A) does not have a full set of n independent eigenvectors. Numerically, the independence of the eigenvector basis deteriorates as $\mu \to \infty$. Just observe that $y_2 \to y_1$ as $\mu \to \infty$.

The sensitivity of an eigenvalue to perturbation depends on 'how independent' its eigenvector is from the eigenvectors associated with other eigenvalues. For example, suppose $\lambda_1(\varepsilon)$ and $\lambda_2(\varepsilon)$ are the eigenvalues of the following perturbation of the T matrix above:

$$\tilde{T} = \begin{bmatrix} \lambda_1 & t \\ \varepsilon & \lambda_2 \end{bmatrix}.$$

It is not hard to show that for $i = 1, 2$, $\dot{\lambda}_i(0) \approx \mu$. Thus, for small ε, the eigenvalues are perturbed by $O(\mu\varepsilon)$.

If A is symmetric then this sensitivity cannot occur. This is because $T = Q^T A Q$ is symmetric and so t is zero. Indeed, a symmetric matrix always has a full set of orthonormal eigenvectors and it can be shown that symmetric $O(\varepsilon)$ perturbations induce just $O(\varepsilon)$ perturbations of the eigenvalues.

1.19. The singular value decomposition

We close with a few remarks about another very important factorization. If $A \in \mathbb{R}^{2\times 2}$, then it is possible to find 2-by-2 orthogonal matrices U and V such that

$$U^{\mathrm{T}}AV = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix}.$$

This is called the *singular value decomposition* and in the $n = 2$ case it has a simple geometric description. In particular, the set

$$\{y\colon\ y = Ax,\ \|x\|_2 = 1\}$$

is an ellipse having semi-axes of length σ_1 and σ_2. Moreover, A maps the first and second columns of V onto $\sigma_1 u_1$ and $\sigma_2 u_2$, where u_1 and u_2 are the first and second columns of U. It can be shown that $\kappa^2(A) = \sigma_1/\sigma_2$ and so the more elongated the ellipse, the bigger the condition.

The singular value decomposition reveals a great deal of A's structure and turns out to be useful in a host of applications.

2. A catalog of matrix factorizations

Matrix factorizations that play a central role in numerical linear algebra are now presented. The discussion for each factorization is generally structured as follows. First, we state the factorization without proof and specify some of its more important related mathematical properties. Then we sketch associated algorithms and their numerical attributes. Finally, we describe a few important applications upon which the factorization has a bearing.

The classification of the matrix computation field by factorization is not too handy if you just want to see how a particular problem might be solved. So for those readers that are 'application driven', here is a more convenient classification in which we mention the relevant factorizations for each given problem:

- *General Linear Systems*. The LU factorization with pivoting and without. See Sections 2.2 and 2.3.
- *Positive Definite Systems*. The Cholesky and $\mathrm{LDL}^{\mathrm{T}}$ factorizations. See Section 2.4 and 2.6.
- *Symmetric Indefinite Systems*. The Bunch–Kaufman and Aasen factorizations. See Sections 2.7 and 2.8.
- *Full Rank Least Squares Problems*. The Cholesky and QR factorizations. See Sections 2.4 and 2.9.
- *Rank Deficient Least Squares Problems*. The QR factorization with column pivoting and the singular value decomposition. See Sections 2.11 and 2.20.

- *The Unsymmetric Eigenvalue Problem.* The Schur, Partial Schur, Jordan, and Hessenberg decompositions. See Sections 2.12–2.15.
- *Symmetric Eigenvalue Problem.* The symmetric Schur and tridiagonal decompositions. See Sections 2.17 and 2.18.
- *The Eigenproblems* $Ax = \lambda Bx$ *and* $A^{\mathrm{T}}Ax = \mu^2 B^{\mathrm{T}}Bx$. The generalized Schur and the generalized singular value decompositions. See Sections 2.16, 2.22, and 2.23.

2.1. A word about notation

In the discussion that follows it is frequently necessary to be able to specify portions of a matrix column or row. A good notation for this is the *colon notation*. If $A \in \mathbb{R}^{m\times n}$, then $A(i, :)$ designates the ith row of A and $A(:, j)$ the jth column. This assumes that $1 \leq i \leq m$ and $1 \leq j \leq n$. We can specify parts of a row or column as well:

$$A(p{:}q, j) = [a_{pj} \cdots a_{qj}]^{\mathrm{T}},$$
$$A(i, p{:}q) = [a_{ip} \cdots a_{iq}].$$

If $A \in \mathbb{R}^{m\times n}$, then $[a_1 \cdots a_n]$ is a *column partitioning* if $a_k \in \mathbb{R}^m$ is the kth column of A.

We also need a notation for specifying submatrices. Suppose $A \in \mathbb{R}^{m\times n}$ and $u \in \mathbb{R}^M$ and $v \in \mathbb{R}^N$ are integer vectors whose components satisfy $1 \leq \mu_i \leq m$ and $1 \leq v_i \leq n$. By $A(u, v)$ we mean the M-by-N matrix whose (i, j)th entry is $A(u_i, v_j)$. Typically u and v are specified by the colon notation. For example, if $u = 2{:}4$ and $v = 7{:}8$, then

$$A(u, v) = A(2{:}4, 7{:}8) = \begin{bmatrix} a_{27} & a_{28} \\ a_{37} & a_{38} \\ a_{47} & a_{48} \end{bmatrix}.$$

A colon designation of the form $i{:}p{:}j$ means count from i to j with increment p. Thus, if $i = 4$, $p = 3$, and $j = 17$, then $u = i{:}p{:}j = (4, 7, 10, 13, 16)$. If a_j designates the jth column of $A \in \mathbb{R}^{m\times n}$, then

$$A(:, i{:}p{:}n) = [a_i, a_{i+p}, \ldots, a_r], \quad r = i + kp,\ k = \mathrm{floor}((n-1)/p).$$

Likewise, $A(1{:}2{:}m, 2{:}2{:}n)$ is the submatrix of A defined by its odd rows and even columns.

2.2. The LU factorization with pivoting

Theorem. *If* $A \in \mathbb{R}^{n\times n}$, *then there exists a permutation* $P \in \mathbb{R}^{n\times n}$ *such that* $PA = LU$, *where* L *is unit lower triangular with* $|l_{ij}| \leq 1$ *and* U *is upper triangular.*

Mathematical notes

The determinant of A is given by $\pm u_{11} \cdots u_{nn}$. [The sign depends on $\det(P)$.] If A is singular, then the factorization still exists but some diagonal entry of U will be zero, say u_{kk}. This implies a dependence among the first k columns of A.

If A is banded, then U has the same upper bandwidth as A and $L(k+1{:}n, k)$ has the same number of nonzeros as $A(k+1{:}n, k)$.

Algorithmic and numerical notes

Gaussian elimination with row pivoting is the standard means for computing this factorization and it requires $\frac{2}{3}n^3$ flops. The classical way of determining P is via *partial pivoting*. Other pivot strategies are used in the sparse matrix setting where one typically relaxes the criteria for pivoting in order to control the *fill-in* of nonzero entries during the elimination process. In this context, factorizations of the form $PAQ = LU$ are often obtained because both columns and rows are permuted.

Sometimes it is possible to improve the quality of $\hat{x}$ through a process known as *iterative improvement*. Here, one computes the residual $r = b - A\hat{x}$ and then uses the factorization to solve $Az = r$. Under certain conditions the refined solution $\bar{x} = \hat{x} + z$ satisfies a perturbed system $(A + F)\bar{x} = b$, where $|f_{ij}| \approx \mathbf{u}|a_{ij}|$. The overall solution procedure is then said to be backwards stable in the componentwise sense.

Applications

- The most common use of the $PA = LU$ factorization is the solution of general linear systems $Ax = b$. Indeed if we solve the triangular systems $Ly = Pb$ and $Ux = y$, then $Ax = b$.
- The factorization can also be used to solve the *multiple right-hand side problem* $AX = B \in \mathbb{R}^{n \times p}$ column-by-column. Setting $B = I$ specifies the inverse, but this is rarely needed explicitly. For example, to compute something like $\alpha = c^{\mathrm{T}}A^{-1}d$, compute $PA = LU$, solve $Az = d$ for z, and set $\alpha = c^{\mathrm{T}}d$.

Further reading

Iterative refinement is a way of improving a computed solution to a linear system assuming that some factorization such as $PA = LU$ is available. See Arioli, Demmel & Duff [1989], Golub & Van Loan [1989], and Skeel [1980]. Various stability issues are covered by Higham & Higham [1988], and Trefethen & Schreiber [1987].

2.3. The LU factorization

Theorem. *If $A \in \mathbb{R}^{n \times n}$ has the property that $\det(A(1{:}k, 1{:}k)) \neq 0$ for $k = 1{:}n-1$, then there exists a unit lower triangular L and upper triangular U such that $A = LU$.*

Mathematical notes

If A is banded, then L and U inherit the lower and upper bandwidth of A.

Algorithmic and numerical notes

Gaussian elimination *without* pivoting is the standard algorithm for computing this factorization. It requires the same amount of floating point work ($\frac{2}{3}n^3$ flops) but can often execute faster because pivoting can be disruptive in advanced computer systems. Moreover, it can sometimes destroy sparsity.

In the linear equation setting it is advisable to use this factorization only if it is known in advance that no small pivots arise. In particular, if Gaussian elimination without pivoting is used to solve a linear system, then it is essential that the resulting L and U factors be sufficiently bounded. Ideally, a rigorous proof covering the problem at hand would do this. Otherwise, one might be forced to rely on massive computational experience or some kind of dynamic monitoring of element growth. The crucial factor is $|\hat{L}|\,|\hat{U}|$, the product of the absolute values of the computed triangular factors. If this matrix has entries significantly larger than $O(n)$, then you probably want to invoke some form of pivoting. Currently, researchers are investigating the possibility of using iterative improvement in conjunction with 'wreckless' LU computation.

Applications

- Diagonally dominant linear systems are an important class of problems for which pivoting is not necessary. Roughly speaking, $A \in \mathbb{R}^{n\times n}$ is *diagonally dominant* if $|a_{ii}| \geq \Sigma_{j\neq i} |a_{ij}|$ for $i = 1{:}n$ with at least one strict inequality. It can be shown that if A^{T} is diagonally dominant, then the LU factorization exists and the l_{ij} are bounded by 1.
- If

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

and A_{11} is k-by-k and nonsingular, then the *Schur complement* of A_{11} in A is prescribed by $C = A_{22} - A_{21}A_{11}^{-1}A_{12}$. This matrix can be obtained as follows:

Compute the LU factorization $A_{11} = L_1U_1$.
Solve $A_{21} = XU_1$ for $X \in \mathbb{R}^{(n-k)\times k}$.
Solve $A_{12} = L_1Y$ for $Y \in \mathbb{R}^{k\times(n-k)}$.
Form $C = A_{22} - XY$.

Further reading

A sampling of papers that deal with LU factorizations of special matrices include Buckley [1974, 1977], de Boor & Pinkus [1977], Erisman & Reid [1974], Golub & Van Loan [1979], and Serbin [1980].

2.4. The Cholesky factorization

Theorem. *If $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite, then there exists a lower triangular $G \in \mathbb{R}^{n\times n}$ such that $A = GG^{\mathrm{T}}$.*

Mathematical notes

The eigenvalues of A are the squares of the singular values of G. The largest entry in G is less than the square root of the largest entry of A (which must occur on the diagonal). The factorization also exists if A is just nonnegative definite, for if $a_{kk} = 0$, then it is easy to show that $A(k{:}n, k) = G(k{:}n, k) = 0$.

Algorithmic and numerical notes

An algorithm for computing the Cholesky factorization involves $\frac{1}{3}n^3$ flops and n square roots, about half the arithmetic associated with Gaussian elimination. If A has bandwidth p, then $O(np^2)$ flops are required and G has lower bandwidth p.

Solving symmetric positive definite systems via Cholesky is backwards stable. However, the process may breakdown if the condition of A has order $1/\mathbf{u}$. Here, 'breakdown' means that a square root of a negative number arises.

For (nearly) semidefinite problems it is sometimes helpful to introduce *symmetric pivoting*. Here, we compute the Cholesky factorization of a permuted version of A: $PAP^{\mathrm{T}} = GG^{\mathrm{T}}$. P is designed so that the last $n - r$ rows of G are zero, where r is the rank of A.

Applications

- Cholesky is useful for solving positive definite systems: $Gy = b$; $G^{\mathrm{T}}x = y \Rightarrow Ax = b$.
- In exact arithmetic, the Cholesky process breaks down if and only if A has a negative eigenvalue. Thus, the Cholesky procedure can be used to detect whether or not a symmetric matrix has a negative eigenvalue.
- If $A \in \mathbb{R}^{m\times n}$ has full column rank, then the solution x_{LS} to the least squares problem $\min \|Ax - b\|_2$ is the solution of the *normal equations* $A^{\mathrm{T}}Ax = A^{\mathrm{T}}b$. If A is poorly conditioned or if $\|Ax_{\mathrm{LS}} - b\|_2$ is fairly small, then one should solve the LS problem via the QR factorization or the SVD, see Sections 2.9, 2.11, and 2.20.

Further reading

Analyses of the Cholesky process appear in Higham [1989], Kielbasinski [1987], and Meinguet [1983].

2.5. Special topic: Updating Cholesky

Quasi-Newton methods for nonlinear optimization frequently require the updating of a Cholesky factorization. For example, at the current step we may

have the Cholesky factorization of an n-by-n approximate Hessian:

$$H_c = L_c L_c^T .$$

At the next step a new approximate Hessian H_+ is obtained via a rank-2 update of H_c:

$$H_+ = H_c + U_c B_c U_c^T , \quad U_c \in \mathbb{R}^{n\times 2}, B_c = B_c^T \in \mathbb{R}^{2\times 2} .$$

In this setting, the central problem is this: *can we find a lower triangular matrix* $L_+ \in \mathbb{R}^{n\times n}$ *in* $O(n^2)$ *flops so that* $H_+ = L_+ L_+^T$? Notice that H_+ need not be positive definite and so the sought after Cholesky factorization may fail to exist.

We outline a procedure which computes L_+ if it exists. More efficient methods exist but our approach has a simplicity which makes it easy to convey the central ideas behind updating.

We begin by solving the multiple right-hand side problem $L_c W = U_c$ for W. This involves $O(n^2)$ flops and it follows that

$$H^+ = L_c(I + WB_c W^T)L_c^T .$$

Next, we find Givens rotations $G_n, \ldots, G_2, J_n, \ldots, J_3$ so that if

$$V = (G_n \cdots G_2)(J_n \cdots J_3) ,$$

then

$$V^T W = \begin{bmatrix} R \\ 0 \end{bmatrix}, \quad R \in \mathbb{R}^{2\times 2} .$$

Moreover, $G_k = G(k-1, k)$ and $J_k = J(k-1, k)$ are rotations in planes $k-1$ and k. An $n = 4$ schematic should illustrate the idea behind the reduction:

$$\begin{bmatrix} \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \end{bmatrix} \xrightarrow{G(3,4)} \begin{bmatrix} \times & \times \\ \times & \times \\ \times & \times \\ 0 & \times \end{bmatrix} \xrightarrow{G(2,3)} \begin{bmatrix} \times & \times \\ \times & \times \\ 0 & \times \\ 0 & \times \end{bmatrix} \xrightarrow{G(1,2)} \begin{bmatrix} \times & \times \\ 0 & \times \\ 0 & \times \\ 0 & \times \end{bmatrix}$$

$$\xrightarrow{J(3,4)} \begin{bmatrix} \times & \times \\ 0 & \times \\ 0 & \times \\ 0 & 0 \end{bmatrix} \xrightarrow{J(2,3)} \begin{bmatrix} \times & \times \\ 0 & \times \\ 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix} .$$

This requires $O(n^2)$ flops. It can be shown that V has upper bandwidth 2 and that if

$$\tilde{B} = I + V^T W B W^T V = \begin{bmatrix} RBR^T & 0 \\ 0 & I_{n-2} \end{bmatrix},$$

then

$$H_+ = (L_c V)\tilde{B}(L_c V)^T .$$

Let U be a rotation in the (1,2) plane so

$$U^T\tilde{B}U = \mathrm{diag}(d_1, d_2, 1, \ldots, 1) \equiv D .$$

The orthogonal matrix $Q = VU$ has upper bandwidth 2 and we have

$$H_+ = (L_c Q)D(L_c Q)^T .$$

H_+ is positive definite if and only if d_1 and d_2 are positive. Assuming this to be the case, we then form $C = L_c Q D^{1/2}$. This can be done in $O(n^2)$ flops because Q is a product of Givens rotations. Note that C has upper bandwidth 2 and that $H_+ = CC^T$.

The final step is to compute an orthogonal matrix Z (product of Givens rotations) such that $Z^T C^T = T$ is upper triangular. This proceeds as follows:

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

$$\xrightarrow{(1,2)} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} .$$

This is a Givens QR factorization process, see Section 2.10. It involves $O(n^2)$ flops and it follows that

$$H_+ = (CZ)(CZ)^T = T^T T \equiv L_+ L_+^T$$

is the required Cholesky factorization.

Further reading

We mention that there are also techniques for updating the factorizations $PA = LU$, $A = GG^T$, and $A = LDL^T$. Updating these factorizations, however, can be quite delicate because of pivoting requirements and because when we tamper with a positive definite matrix the result may not be positive definite. See Björck [1984], Bojanczyk, Brent, Van Dooren & de Hoog [1987], Bunch, Nielsen & Sorensen [1978], Gill, Golub, Murray & Saunders [1974], Stewart [1979], Bartels [1971], Daniel, Gragg, Kaufman & Stewart [1976], Gill, Murray & Saunders [1975], and Goldfarb [1976].

2.6. The LDL^T factorization

Theorem. *If $A \in \mathbb{R}^{n\times n}$ is symmetric and has the property that $\det(A(1{:}k, 1{:}k)) \neq 0$ for $k = 1{:}n-1$, then there exists a unit lower triangular L and a diagonal D such that $A = LDL^T$.*

Algorithmic and numerical notes

If A is positive definite, then $LD^{1/2}$ is the Cholesky factor. If $U = DL^T$, then $A = LU$. Thus, we can think of LDL^T as a 'square root free' Cholesky or as natural symmetrization of the LU factorization.

It is generally not advisable to use this factorization unless A is positive definite. Otherwise, the same guidance applies that we offered in connection with LU without pivoting. Namely, the method is not backwards stable unless L and D are suitably bounded. This is assured, for example, if A is diagonally dominant.

Applications

- The main application of this factorization is in the solution of symmetric positive definite systems. Indeed, if $Ly = b$, $Dz = y$, and $L^Tx = z$, then $Ax = b$.
- In some nonlinear optimization problems, symmetric linear systems $As = -g$ arise which 'should be' positive definite A. (Here, s is a step direction, A is an approximate Hessian, and g is a gradient.) If LDL^T is used, then it is possible to dynamically determine a nonnegative diagonal matrix Δ so that no negative D_{kk} arise. The result is a factorization of the form $(A + \Delta) = LDL^T$. One then solves $(A + \Delta)s = -g$ instead of $As = -g$ and obtains a preferred s.
- In general, any application that calls for Cholesky can be solved via LDL^T. However, when solving a band positive definite systems it is better to rely on the latter factorization. The n square roots that surface in Cholesky can represent a significant portion of the arithmetic when A is banded. This is especially true if A is tridiagonal.

2.7. The Bunch–Kaufman factorization

Theorem. *If $A \in \mathbb{R}^{n\times n}$ is symmetric, then there exists a permutation P such that $PAP^T = LDL^T$ where $L \in \mathbb{R}^{n\times n}$ is unit lower triangular with $|l_{ij}| \leq 1$ and D is a direct sum of 1-by-1 and 2-by-2 submatrices.*

Algorithmic and numerical notes

The factorization is discussed by Bunch & Parlett [1971], but Bunch & Kaufman [1977] discovered a clever pivot strategy that results in a 'Cholesky speed' procedure, i.e., $\frac{1}{3}n^3$ flops and $O(n^2)$ compares. It is as stable as Gaussian elimination with partial pivoting. Because of the pivot strategy, it is not possible to exploit band structure if it is present.

Applications

• The leading application of this factorization is the solution of symmetric indefinite systems: $Lw = Pb$, $Dz = w$, $L^T y = z$, $x = Py \Rightarrow Ax = b$. Note that the $Dz = w$ system involves solving 1-by-1 and 2-by-2 indefinite symmetric systems.

• The *inertia* of a symmetric matrix is a triplet of nonnegative integers (P, Z, N) which are the number of positive, zero, and negative eigenvalues, respectively. A famous theorem due to Sylvester states that the inertia of A, and $X^T AX$ are the same if X is nonsingular. Thus, A and D have the same inertia and the inertia of the latter is easily computed. (Just examine the eigenvalues of the 1-by-1 and 2-by-2 blocks in D.)

Further reading

The main papers associated with the computation of this factorization include Bunch [1971], Bunch & Kaufman [1977], Bunch, Kaufman & Parlett [1976], and Bunch & Parlett [1971].

2.8. The Aasen factorization

Theorem. *If $A \in \mathbb{R}^{n\times n}$ is symmetric, then there exists a permutation P such that $PAP^T = LTL^T$ where $L \in \mathbb{R}^{n\times n}$ is unit lower triangular with $|l_{ij}| \leq 1$ and T is tridiagonal.*

Algorithmic and numerical notes

Early papers gave an algorithm requiring about $\frac{2}{3}n^3$ flops. Aasen [1971] showed how to halve the work. No efficient version for band problems currently exists. The procedure is backwards stable.

Applications

• Any problem that can be solved by the Bunch–Kaufman factorization can also be solved via the Aasen approach.

Further reading

The original reference for the Aasen algorithm is Aasen [1971]. Comparisons with the diagonal pivoting approach are offered by Barwell & George [1976].

2.9. The QR factorization

Theorem. *If $A \in \mathbb{R}^{m\times n}$ then there exists an orthogonal $Q \in \mathbb{R}^{m\times n}$ and an upper triangular $R \in \mathbb{R}^{m\times n}$ such that $A = QR$.*

Mathematical notes

If $\text{span}\{a_1, \ldots, a_n\}$ defines an n-dimensional subspace of $\mathbb{R}^m$ and $QR = [a_1, \ldots, a_n]$ is the QR factorization, then the columns of $Q(:, 1{:}k)$ form an

orthonormal basis for $\text{span}\{a_1, \ldots, a_k\}$ for $k = 1{:}n$. The columns of the matrix $Q(:, n+1{:}m)$ are an orthonormal basis for $\text{span}\{a_1, \ldots, a_n\}^{\perp}$. If A is rank deficient, then the QR factorization still exists but then R has one or more zero diagonal entries.

Algorithmic notes

The factorization can be computed using Householder transformations. Here, $Q = H_1 \cdots H_n$ and the 'mission' of H_k is to zero the subdiagonal portion of the kth column of A. Givens rotations can also be used but this is advisable only if A is sparse or in certain parallel computation settings.

A careful implementation of the Gram–Schmidt process (known as *modified Gram–Schmidt*) can also be used to compute the QR factorization. If Householder or Givens transformations are used, then the computed Q is orthogonal to working precision. In modified Gram–Schmidt, the quality of Q's orthogonality depends upon the condition of A.

Applications

- The QR factorization is frequently used to compute an orthonormal basis for a subspace and/or its orthogonal complement. However, if the dimension is 'fuzzy', then the SVD (see Section 2.20) might be more appropriate.
- The least square solution of full rank overdetermined system of equations is a lead application for the QR factorization. Suppose $A \in \mathbb{R}^{m \times n}$ with $m \geq n$ and $b \in \mathbb{R}^m$ are given. It follows that if

$$Q^{\mathrm{T}}A = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}, \qquad Q^{\mathrm{T}}b = \begin{bmatrix} c \\ d \end{bmatrix},$$

where $R_1 \in \mathbb{R}^{n \times n}$, $c \in \mathbb{R}^n$, and $d \in \mathbb{R}^{m-n}$, then

$$\|Ax - b\|^2 = \|Q^{\mathrm{T}}(Ax - b)\|^2 = \|R_1 x - c\|^2 + \|d\|^2 ,$$

and so the minimizing x_{LS} is prescribed by the upper triangular system $R_1 x_{\mathrm{LS}} = c$.

If $\hat{x}_{\mathrm{LS}}$ is the computed solution obtained in this fashion, then $\hat{x}_{\mathrm{LS}}$ solves a nearby LS problem and its relative error has the form

$$\frac{\|\hat{x}_{\mathrm{LS}} - x_{\mathrm{LS}}\|_2}{\|x_{\mathrm{LS}}\|_2} \approx \alpha \kappa_2(A) + \beta \|Ax_{\mathrm{LS}} - b\|_2 \kappa_2(A)^2 ,$$

where α and β are small constants. Note the $\kappa(A)^2$ term.

Further reading

An excellent overall reference for all aspects of the least squares problem is Björck [1988]. The intelligent implementation of the QR process via orthogonal transformations is discussed by Businger & Golub [1965] and Golub [1965].

Perturbation theory and associated error analyses for QR solution of the least squares problem is covered by Björck [1987], Gentleman [1973], and Stewart [1977]. Sparse/banded LS solution techniques are detailed by Cox [1981], Duff & Reid [1976], and Gill & Murray [1976].

2.10. *Special topic: Equality constrained least squares*

Consider the least squares problem with linear equality constraints:

$$\min_{Bx=d} \|Ax - b\|_2 .$$

Here $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times n}$, $b \in \mathbb{R}^m$, $d \in \mathbb{R}^p$, and $\operatorname{rank}(B) = p$. We refer to this as the *LSE problem*. Assume for clarity that both A and B have full rank. Let

$$Q^T B^T = \begin{bmatrix} R \\ 0 \end{bmatrix} \begin{matrix} p \\ n-p \end{matrix}$$

be the QR factorization of B^T and set

$$AQ = \underset{p}{[A_1} \quad \underset{n-p}{A_2}], \qquad Q^T x = \begin{bmatrix} y \\ z \end{bmatrix} \begin{matrix} p \\ n-p \end{matrix} .$$

It is clear that with these transformations the LSE problem becomes

$$\min_{R^T y = d} \|A_1 y + A_2 z - b\|_2 .$$

Thus, y is determined from the lower triangular constraint equation $R^T y = d$ and the vector z is obtained by solving the *unconstrained* LS problem

$$\min_z \|A_2 z - (b - A_1 y)\|_2 .$$

Combining the above, we see that $x = Q\left[\begin{smallmatrix} y \\ z \end{smallmatrix}\right]$ is the required solution. See Golub & Van Loan [1989] for more details.

An alternative approach for solving the LSE problem, called the *method of weighting*, is attractive for its simplicity. In this method the unconstrained LS problem

$$\min_z \left\| \begin{bmatrix} A \\ \lambda B \end{bmatrix} x - \begin{bmatrix} b \\ \lambda d \end{bmatrix} \right\|_2$$

is solved. Intuitively, if λ is large, then the constraint equation $Bx = d$ will tend to be satisfied. A tricky feature associated with this method is the handling of large λ. See Barlow, Nichols & Plemmons [1988], Eldèn [1980], Lawson & Hanson [1974], and Van Loan [1985b] for further discussion.

2.11. The QR factorization with column pivoting

Theorem. *If $A \in \mathbb{R}^{m\times n}$ then there exists an orthogonal $Q \in \mathbb{R}^{m\times m}$ and a permutation matrix $P \in \mathbb{R}^{n\times n}$, such that*

$$Q^{\mathrm{T}}A\Pi = R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \\ r & n-r \end{bmatrix} \begin{matrix} r \\ m-r \end{matrix},$$

where $r = \operatorname{rank}(A)$ and R_{11} is upper triangular and nonsingular.

Mathematical notes

Any vector of the form

$$x = \Pi^{\mathrm{T}} \begin{bmatrix} -R_{11}^{-1}R_{12}z \\ z \end{bmatrix},$$

where $z \in \mathbb{R}^{n-r}$, is in the null space of A.

Algorithmic and numerical notes

The factorization amounts to a QR factorization of a column-permuted version of A. There are several ways to determine Π. The usual procedure is to determine $\Pi = \Pi_1 \cdots \Pi_n$ as a product of exchange permutations. Π_j is computed at step j so that the 2-norm of the current $A(j{:}m, j)$ is maximized. As a result of this pivot strategy we have

$$r_{jj}^2 \geqslant \sum_{i=j}^{k} r_{ik}^2$$

for $k = j+1{:}n$. Thus, the numbers in R diminish in size towards the bottom right-hand corner. This property is attractive with many triangular matrix condition number estimators.

Applications

- If $r = m < n$, then any vector of the form

$$x = \Pi \begin{bmatrix} K_{11}^{-1}Q^{\mathrm{T}}b \\ 0 \end{bmatrix}$$

solves the *underdetermined* system $Ax = b$.

- The factorization can be used to solve rank-deficient least squares problems $\min \|Ax - b\|_2$. Suppose $A \in \mathbb{R}^{m\times n}$ has rank r. QR with column pivoting produces the factorization $A\Pi = QR$, where

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \\ r & n-r \end{bmatrix} \begin{matrix} r \\ m-r \end{matrix}.$$

Given this reduction, the LS problem can be readily solved. Indeed, for any

$x \in \mathbb{R}^n$ we have

$$\begin{aligned} \|Ax - b\|_2^2 &= \|(Q^T A\Pi)(\Pi^T x) - (Q^T b)\|_2^2 \\ &= \|R_{11}y - (c - R_{12}z)\|_2^2 + \|d\|_2^2 , \end{aligned}$$

where

$$\Pi^T x = \begin{bmatrix} y \\ z \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \quad \text{and} \quad Q^T b = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} r \\ m-r \end{matrix} .$$

Thus, if x is an LS minimizer, then we must have

$$x = \Pi \begin{bmatrix} R_{11}^{-1}(c - R_{12}z) \\ z \end{bmatrix} .$$

If z is set to zero in this expression, then we obtain the *basic solution*

$$x_B = \Pi \begin{bmatrix} R_{11}^{-1}c \\ 0 \end{bmatrix} .$$

Notice that x_B has at most r nonzero components and so the predictor Ax_B involves a subset of A's columns. Indeed, QR with column pivoting can be thought of as a heuristic method for selecting a 'strongly' independent subset of A's columns.

Further reading

Using QR with column pivoting in the rank-deficient LS setting is discussed by Chan [1987], and Businger & Golub [1965]. Deducing the condition of A from the condition of R is detailed by Anderson & Karasalo [1975]. Perturbation aspects of the rank deficient LS problem are covered by Stewart [1984, 1987], and Wedin [1973].

2.12. The partial Schur decomposition

Theorem. *Suppose $A \in \mathbb{R}^{n\times n}$ and that $Q \in \mathbb{R}^{n\times n}$ is orthogonal. If the columns of $Q_1 = Q(:, 1{:}p)$ define an invariant subspace, then*

$$A^T A Q = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix}$$

and the eigenvalues of $T_{11} = Q_1^T A Q_1$ are also eigenvalues of A.

Mathematical notes

This decomposition splits the spectrum in that $\lambda(A) = \lambda(T_{11}) \cup \lambda(T_{22})$. The columns of $Q_2 = Q(:, p+1{:}n)$ do *not* define an invariant subspace unless T_{12} is zero.

If T_{11} and T_{22} have no eigenvalues in common, then the *Sylvester equation*

$$T_{11}Y - YT_{22} = -T_{12}\,, \quad Y \in \mathbb{R}^{n\times p}$$

has a solution. This permits block diagonalization for if we set

$$Z = \begin{bmatrix} I_p & Y \\ 0 & I_{n-p} \end{bmatrix},$$

then

$$Z^{-1}TZ = \begin{bmatrix} T_{11} & T_{11}Y - YT_{22} + T_{12} \\ 0 & T_{22} \end{bmatrix} = \begin{bmatrix} T_{11} & 0 \\ 0 & T_{22} \end{bmatrix}.$$

Complex versions of the above reductions are possible. The matrix Q is then complex and unitary. Note that complex quantities can arise in a real matrix eigenreduction because of possible complex conjugate eigenvalues.

Algorithmic notes

Suppose $|\lambda_1| \geqslant \cdots \geqslant |\lambda_n|$ is an ordering of $\lambda(A)$. If $|\lambda_p| > |\lambda_{p+1}|$, then a method known as *orthogonal iteration* can be used to compute a p-dimensional invariant subspace associated with the p largest eigenvalues. If $Q^{(0)} \in \mathbb{R}^{n\times p}$ is an initial guess, then the kth step in this iteration has the following form:

$$Z = AQ_1^{(k-1)}\,, \quad \text{matrix–matrix multiply}\,,$$

$$Q_1^{(k)}R^{(k)} = Z\,, \quad \text{QR factorization}\,.$$

If $Q_1^{(0)}$ is not 'deficient', then the distance between $\operatorname{ran}(Q_1^{(k)})$ and $\operatorname{ran}(Q_1^{(0)})$ is bounded by a constant times $|\lambda_{p+1}/\lambda_p|^k$.

If $p = 1$, then the technique is known as the *power method*. If we replace A by A^{-1}, then the resulting iteration is called *inverse orthogonal iteration*. It can be used to find the small eigenvalues of A but it requires the solution of a linear system at each step.

Further reading

The connections among the various power iterations are discussed by Parlett & Poole [1973], Stewart [1975, 1976b], and Golub & Van Loan [1989].

2.13. *The Schur decomposition*

Theorem. *If $A \in \mathbb{R}^{n\times n}$ has no complex eigenvalues, then there exists an orthogonal $Q \in \mathbb{R}^{n\times n}$ such that $Q^{\mathrm{T}}AQ = T$ is upper triangular.*

Mathematical notes

The diagonal elements of T are the eigenvalues of A. The columns of Q are called *Schur vectors*. For $j = 1{:}n$, $\operatorname{ran}(Q(:, 1{:}j))$ is an invariant subspace. The

column $Q(:, j)$ is an eigenvector if and only if $T(1{:}j-1, j) = 0$. T is diagonal if and only if A is *normal*, i.e., $A^{\mathrm{H}}A = AA^{\mathrm{H}}$.

If $A \in \mathbb{R}^{n\times n}$ has complex eigenvalues, then they occur in conjugate pairs and it is possible to determine an orthogonal $Q \in \mathbb{R}^{n\times n}$ so that T is *quasi-triangular*. This means that T is upper triangular with 2-by-2 'bumps' along the diagonal corresponding to the complex conjugate eigenpairs. In particular, if $t_{k,k-1} \neq 0$, then the eigenvalues of

$$T(k-1{:}k, k-1{:}k) = \begin{bmatrix} t_{k-1,k-1} & t_{k-1,k} \\ t_{k,k-1} & t_{kk} \end{bmatrix}$$

are complex conjugate eigenvalues of A. The real orthogonal reduction to quasi-triangular form is referred to as the *real Schur decomposition*.

If complex transformations are 'admissable', then we can find a unitary $Q \in \mathbb{C}^{n\times n}$ such that $Q^{\mathrm{H}}AQ = T$ is upper triangular (and complex).

Algorithmic and numerical notes

The standard method for computing this decomposition is the QR iteration with shifts. A step in this procedure has the following form:

Determine approximate eigenvalues μ_1 and μ_2.
Compute the QR factorization $QR = (A - \mu_1 I)(A - \mu_2 I)$.
$A_{\text{new}} = Q^{\mathrm{T}}AQ$.

The key to a successful implementation of this iteration involves (a) the preliminary reduction of A to upper Hessenberg form (see Section 2.14), (b) the intelligent selection of the *shifts* μ_1 and μ_2, and (c) a clever $O(n^2)$ method for computing A_{new} from Q to A. The overall reduction involves about $30n^3$ flops if Q is required, $15n^3$ if not.

Once a real Schur decomposition is obtained, the columns of Q and the diagonal entries of T can be cheaply re-ordered so that the eigenvalues appear in any prescribed order. This is important because we can then acquire an orthonormal basis for an invariant subspace associated with any subset of eigenvalues.

The QR iteration is backwards stable meaning that the computed $\hat{T}$ is exactly similar to a matrix near to A in norm. This does *not* mean that the computed eigenvalues are accurate. Indeed, if λ is a distinct eigenvalue of A, then its computed analog $\hat{\lambda}$ satisfies

$$|\lambda - \hat{\lambda}| \approx \frac{\mathbf{u}}{s(\lambda)},$$

where $s(\lambda) = |x^{\mathrm{T}}y|$. Here, x and y are unit 2-norm right and left eigenvectors: $Ax = \lambda x$, $y^{\mathrm{T}}A = \lambda y^{\mathrm{T}}$. The reciprocal of $s(\lambda)$ can be thought of as a condition for λ. If λ is not distinct, then matters get complicated, see Golub & Van Loan [1989, Section 7.2].

Applications

• Once the Schur decomposition is acquired, the eigenvectors of A can be found through a back substitution process. In particular, if t_{kk} is a distinct eigenvalue and we solve $(T(1{:}k-1, 1{:}k-1) - t_{kk}I)z = -T(1{:}k-1, k)$ for z, then

$$x = Q\begin{bmatrix} z \\ -1 \\ 0 \end{bmatrix}$$

satisfies $Ax = \lambda x$.

• The Lyapunov equation $FX + XF^{\mathrm{T}} = C$, where $F \in \mathbb{R}^{m\times m}$, and $C \in \mathbb{R}^{n\times n}$, can be solved by computing the Schur decomposition $U^{\mathrm{T}}FU = T$. (For clarity we assume that F has real eigenvalues.) Note that with the Schur decomposition the original Lyapunov equation transforms to $TY + YT^{\mathrm{T}} = \tilde{C}$ where $X = UYU^{\mathrm{T}}$ and $\tilde{C} = U^{\mathrm{T}}CU$. If $Y = [y_1, \ldots, y_n]$ and $\tilde{C} = [\tilde{c}_1, \ldots, \tilde{c}_n]$ are column partitioning, then for $k = 1{:}n$ we have

$$(T + t_{kk}I)y_k = \tilde{c}_k - \sum_{j=k+1}^{n} t_{kj}y_j \,.$$

Y can be resolved by invoking this formula for $k = n, n-1, \ldots, 1$.

Further reading

The mathematical properties of the algebraic eigenvalue problem and related perturbation theory are covered by Golub & Van Loan [1989].

Practical invariant subspace computation is detailed by Golub & Wilkinson [1976], Stewart [1976a], and Bavely & Stewart (1979). Various aspects of eigenvector computation are treated by Chan & Parlett [1977], Symm & Wilkinson [1980], and Van Loan [1987].

2.14. The Hessenberg decomposition

Theorem. *If $A \in \mathbb{R}^{n\times n}$ then there exists an orthogonal $Q \in \mathbb{R}^{n\times n}$ such that $Q^{\mathrm{T}}AQ = H$ is upper Hessenberg. In other words, $h_{ij} = 0$ whenever $i > j + 1$.*

Mathematical notes

If no subdiagonal entry of H is zero, then H is *unreduced*. If H is unreduced and $\lambda \in \lambda(A)$, then $\operatorname{rank}(A - \lambda I) = n - 1$. This implies that if H is unreduced, then there is at most one independent eigenvector per eigenvalue.

If q_i is the ith column of Q, then $\{q_1, \ldots, q_j\}$ is an orthonormal basis for the *Krylov subspace* $\operatorname{span}\{q_1, Aq_1, \ldots, A^{j-1}q_1\}$ assuming the latter subspace has dimension j.

Algorithmic and numerical notes

The standard method for computing the factorization is via Householder matrices. In particular, $Q = Q_1 \cdots Q_{n-2}$ where Q_j is a Householder matrix

whose mission is to zero $A(j+2{:}n, j)$. This procedure requires $\frac{10}{3}n^3$ flops and is backwards stable.

If A is large and sparse the method of Arnoldi is of interest. After k steps of the Arnoldi process the leading k-by-k portion of H is available and its eigenvalues can be regarded as approximate eigenvalues of A. The Arnoldi algorithm requires only matrix–vector products involving A and thus has potential for large sparse problems.

Applications

- The Hessenberg reduction is usually applied to A before computing the real Schur form. The QR iteration preserves Hessenberg form, a property that reduces work per iteration by an order of magnitude.
- If one has to solve $(A-\mu I)x = b$ for many different $\mu \in \mathbb{R}$ and $b \in \mathbb{R}^n$, then the volume of computation can be greatly reduced if A is first reduced to Hessenberg form. Note that if we solve $(H-\mu I)y = Q^{\mathrm{T}}b$ and set $x = Qy$, then $(A-\mu I)x = b$. This involves a pair of matrix–vector products and a Hessenberg system solve implying $O(n^2)$ flops.
- The *Sylvester equation* $FX + XG = C$, where $F \in \mathbb{R}^{m\times m}$, $G \in \mathbb{R}^{n\times n}$, and $C \in \mathbb{R}^{m\times n}$, can be solved by computing the Hessenberg decomposition $U^{\mathrm{T}}FU = H$ and the Schur decomposition $V^{\mathrm{T}}GV = S$. (For clarity we assume that G has real eigenvalues.) The original Sylvester equation now transforms to $HY + YS = \tilde{C}$, where $X = UYV^{\mathrm{T}}$ and $\tilde{C} = U^{\mathrm{T}}CV$. If $Y = [y_1, \ldots, y_m]$ and $\tilde{C} = [\tilde{c}_1, \ldots, \tilde{c}_m]$ are column partitionings, then for $k = 1{:}n$ we have

$$(H + s_{kk}I)y_k = \tilde{c}_k - \sum_{j=1}^{k-1} s_{jk}y_j \,.$$

Thus, if $y_1, \ldots, y_n$ are computed in turn, then we obtain Y.

- If $\hat{\lambda}$ is a computed eigenvalue of H, then a corresponding eigenvector can be found via *inverse iteration*. In this procedure, the nearly singular Hessenberg system $(H-\hat{\lambda}I)x = c$ is solved for a random c. The solution x is very often rich in the desired eigendirection.

Further reading

The Householder reduction to Hessenberg form is discussed by Dongarra, Kaufman & Hammarling [1986], and Businger [1969, 1971].

For a survey of applications where the decomposition has proven to be very useful, see Golub, Nash & Van Loan [1979], Laub [1981], and Van Loan [1982b].

2.15. Jordan canonical form

Theorem. *If $A \in \mathbb{C}^{n\times n}$ then there exists a nonsingular $X \in \mathbb{C}^{n\times n}$ such that $X^{-1}AX = \mathrm{diag}(J_1, \ldots, J_t)$ where*

$$J_i = \begin{bmatrix} \lambda_i & 1 & \cdots & 0 \\ 0 & \lambda_i & \ddots & \vdots \\ \vdots & & \ddots & 1 \\ 0 & \cdots & 0 & \lambda_i \end{bmatrix}$$

is n_i*-by-*n_i *and* $n_1 + \cdots + n_t = n$.

Mathematical notes

The J_i are referred to as *Jordan blocks*. The number and dimensions of the Jordan blocks associated with each distinct eigenvalue is unique, although their ordering along the diagonal is not. If a Jordan block is bigger than 1-by-1, then the associated eigenvalue is said to be *defective*. This means that A has fewer than n independent eigenvectors. If this is not the case, then A is said to be *diagonalizable*.

Algorithmic and numerical notes

Because it involves a set of numerical rank decisions, computation of the JCF is tricky and involves use of the singular value decomposition (see Section 2.20) to resolve difficult questions that concern eigenvalue multiplicity.

Further reading

Some intelligent algorithms that address the difficulties associated with JCF computation include Ruhe [1969], Kagstrom & Ruhe [1980a,b], and Demmel [1983].

2.16. *The generalized Schur decomposition*

Theorem. *If* $A \in \mathbb{C}^{n\times n}$ *and* $B \in \mathbb{C}^{n\times n}$, *then there exist unitary* $Q, Z \in \mathbb{C}^{n\times n}$ *such that* $Q^H A Z = T$ *and* $Q^H B Z = S$ *are upper triangular.*

Mathematical notes

The λ which make $A - \lambda B$ singular are called the *generalized eigenvalues* of the *pencil* $A - \lambda B$. Since

$$\det(A - \lambda B) = \det(Q^H Z)\det(T - \lambda S) = \det(Q^H Z) \prod_{k=1}^{n} (t_{kk} - \lambda s_{kk}),$$

it follows that the quotients t_{kk}/s_{kk} are generalized eigenvalues. If $s_{kk} = 0$ then $A - \lambda B$ has an infinite eigenvalue. If for some k, $t_{kk} = s_{kk} = 0$, then $\lambda(A, B) = \mathbb{C}$.

If B is nonsingular then $Q^H A B^{-1} Q = T S^{-1}$ is the Schur decomposition of $C = AB^{-1}$.

The columns of $Z_1 = Z(:, 1{:}p)$ define what is called a *deflating subspace* for the matrix pencil $A - \lambda B$. In particular, $\operatorname{ran}(AZ_1) = \operatorname{ran}(BZ_1)$.

Algorithmic and numerical notes

A generalization of the QR algorithm called the QZ algorithm is the standard approach to this problem. The algorithm is backwards stable and is *not* effected by singularity in A or B.

Applications

• The main application of the QZ algorithm is the computation to generalized eigenvalues and their associated eigenvectors. Indeed, the eigenvectors of the triangular pencil $T - \lambda S$ can be found via back substitution analogous to how one proceeds with the standard Schur decomposition.

Further reading

Many theoretical and practical aspects of the $A - \lambda B$ problem are covered by Kågström & Ruhe [1983], and Stewart [1972]. Various aspects of the QZ algorithm are detailed by Moler & Stewart [1973], Ward [1975], and Golub & Van Loan [1989].

2.17. The symmetric Schur decomposition

Theorem. *If $A \in \mathbb{R}^{n\times n}$ is symmetric, then there exists an orthogonal $Q \in \mathbb{R}^{n\times n}$ such that $Q^{\mathrm{T}}AQ = D = \mathrm{diag}(\lambda_i)$.*

Mathematical notes

Because symmetric matrices have a diagonal Schur form, the columns of Q are eigenvectors: $AQ(:, j) = Q(:, j)\lambda_j$.

The stationary values of the *Rayleigh quotient*

$$r(x) = \frac{x^{\mathrm{T}}Ax}{x^{\mathrm{T}}x}$$

are the eigenvalues of A. Moreover, if $\lambda_1 \geqslant \lambda_2 \geqslant \cdots \geqslant \lambda_n$, then

$$\lambda_k(A) = \max_{\dim(S)=k} \min_{0\neq y\in S} r(y)\,.$$

This result can be used to show that if $\lambda \in \lambda(A)$, then there exists an eigenvalue $\tilde{\lambda} \in \lambda(A + E)$ such that $|\lambda - \tilde{\lambda}| \leqslant \|E\|_2$. Unlike the unsymmetric eigenvalue problem, $O(\varepsilon)$ perturbations of A induce $O(\varepsilon)$ perturbations of the eigenvalues.

Algorithmic and numerical notes

If A is tridiagonal (see Section 2.18), then a symmetrized version of the QR iteration described in Section 2.13 is an effective method for computing the decomposition. A divide-and-conquer technique based on tearing is an alternative method of interest.

If A is full, then the Jacobi iteration can be used. Jacobi's idea is to compute

the real Schur decomposition via a sequence of 2-by-2 problems. In particular, updates of the form

$$A \leftarrow J_{pq}^{T} A J_{pq}$$

are repeatedly performed, where J_{pq} is a Givens rotation in the (p, q) plane. With each update the Frobenius norm of the off-diagonal portion of A is strictly reduced. By choosing the rotation indices (p, q) carefully it is possible to introduce significant amounts of parallelism.

Applications

- In certain optimization algorithms one would like to translate the system $Ax = b$ so that the solution is suitably bounded. In particular, we are given $A = A^{T} \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^{n}$, and $\delta > 0$ and we seek $\sigma \geq 0$ such that the solution $x(\sigma)$ to $(A + \sigma I)x = b$ satisfies $\|x(\sigma)\|_2 \leq \delta$. This problem transforms to an equivalent diagonal problem via the symmetric Schur decomposition: $(D + \sigma I)y(\sigma) = \tilde{b}$ subject to $\|y(\sigma)\|_2 \leq \delta$. Since $y_i(\sigma) = \tilde{b}_i/(\lambda_i + \sigma)$ the optimum y is readily found.
- The maximization of $r(x)$ over a subspace is the largest eigenvalue of $U^{T}AU$ where the columns of U are an orthonormal basis for the subspace.

Further reading

The book by Parlett [1980] covers many aspects of the symmetric eigenvalue problem. See also Golub & Van Loan [1989] for algorithmic details.

2.18. *Orthogonal tridiagonalization*

Theorem. *If $A \in \mathbb{R}^{n \times n}$ is symmetric, then there exists an orthogonal $Q \in \mathbb{R}^{n \times n}$ such that $Q^{T}AQ = T$ is tridiagonal, i.e., $t_{ij} = 0$ if $|i - j| > 1$.*

Mathematical notes

If T has no zero subdiagonal elements, then A has no repeated eigenvalues. Let $T_r = T(1{:}r, 1{:}r)$ denote the leading r-by-r principal submatrix of

$$T = \begin{bmatrix} \alpha_1 & \beta_1 & \cdots & 0 \\ \beta_1 & \alpha_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \beta_{n-1} \\ 0 & \cdots & \beta_{n-1} & \alpha_n \end{bmatrix},$$

and define the polynomials $p_r(x) = \det(T_r - xI)$, $r = 1{:}n$. A simple determinan-

tal expansion can be used to show that

$$p_r(x) = (\alpha_r - x)p_{r-1}(x) - \beta_{r-1}^2 p_{r-2}(x)$$

for $r = 2{:}n$ if we define $p_0(x) = 1$.

Algorithmic and numerical notes

For dense problems, Q is usually computed as a product of Householder transformations: $Q = Q_1 \cdots Q_{n-2}$. The resulting algorithm is backwards stable.

The *method of Lanczos* is a technique for computing T without resorting to updates of A, see Section 2.19.

Because the polynomial $p_n(x)$ can be evaluated in $O(n)$ flops, it is feasible to find its roots by using the method of bisection.

Applications

- Given a general symmetric matrix A, it is customary to tridiagonalize it first before applying the QR iteration for eigenvalues.
- Some of the applications of the Hessenberg decomposition (Section 2.14) can be solved via tridiagonalization if A is symmetric. For example, if we must solve symmetric systems of the form $(A - \sigma I)x = b$ for many different σ and b.

Further reading

The tridiagonalization of a symmetric matrix is detailed by Golub & Van Loan [1989].

2.19. Special topic: The Lanczos and conjugate gradient algorithms

A different method for computing the tridiagonalization can be derived by comparing columns in the equation $AQ = QT$. With the notation of the previous section and the column partitioning $Q = [q_1, \ldots, q_n]$ we obtain

$$Aq_j = \beta_{j-1}q_{j-1} + \alpha_j q_j + \beta_j q_{j+1} \quad (\beta_0 q_0 \equiv 0)$$

for $j = 1{:}n - 1$. The orthonormality of the q_i implies $\alpha_j = q_j^{\mathrm{T}} A q_j$. Moreover, if $r_j = (A - \alpha_j I)q_j - \beta_{j-1}q_{j-1}$ is nonzero, then $q_{j+1} = r_j/\beta_j$, where $\beta_j = \pm\|r_j\|_2$. Collecting these observations we obtain the *Lanczos iteration*:

$r_0 = q_1;\ \beta_0 = 1;\ q_0 = 0;\ j = 0$
while $\beta_j \neq 0$
  $q_{j+1} = r_j/\beta_j;\ j = j + 1;\ \alpha_j = q_j^{\mathrm{T}} A q_j$
  $r_j = (A - \alpha_j I)q_j - \beta_{j-1}q_{j-1};\ \beta_j = \|r_j\|_2$
end

See Golub & Van Loan [1989, p. 478]. The q_j are called *Lanczos vectors*.

Notice that the matrix A is involved only through matrix–vector multiplication. This is important in large sparse problems because the Householder approach can lead to an unacceptable level of fill-in. If $\beta_j = 0$ then $q_1, \ldots, q_j$ define an invariant subspace for A associated with the eigenvalues of $T(1{:}j, 1{:}j)$.

Unfortunately, the computed Lanczos vectors are usually far from orthogonal and a great deal of research has been devoted to handle this problem. On the positive side, the extreme eigenvalues of $T(1{:}j, 1{:}j)$ tend to be very good approximations of A's extreme eigenvalues even for relatively small j making the Lanczos procedure very attractive in certain settings.

The Lanczos procedure can also be used to solve sparse symmetric positive definite systems $Ax = b$. Indeed the sequence of vectors x_j produced by the iteration

$r_0 = b$; $\beta_0 = \|b\|_2$; $q_0 = 0$; $j = 0$; $x_0 = 0$
while $\beta_j \neq 0$
  $q_{j+1} = r_j/\beta_j$; $j = j + 1$; $\alpha_j = q_j^{\mathrm{T}} A q_j$
  $r_j = (A - \alpha_j I)q_j - \beta_{j-1} q_{j-1}$; $\beta_j = \|r_j\|_2$
  if $j = 1$
    $d_1 = \alpha_1$; $c_1 = q_1$; $\rho_1 = \beta_0/\alpha_1$; $x_1 = \rho_1 q_1$
  else
    $\mu_{j-1} = \beta_{j-1}/d_{j-1}$; $d_j = \alpha_j - \beta_{j-1}u_{j-1}$
    $c_j = q_j - \mu_{j-1}c_{j-1}$; $\rho_j = -\mu_{j-1}d_j - {}_1\rho_{j-1}/d_j$
    $x_j = x_{j-1} + \rho_j c_j$
  end
end
$x = x_j$

have the property that they minimize $\phi(x) = \frac{1}{2}x^{\mathrm{T}}Ax - x^{\mathrm{T}}b$ over $\mathrm{span}\{q_1, \ldots, q_j\}$. It follows that $Ax_n = b$ since $x = A^{-1}b$ is the global minimizer of ϕ. However, it turns out that x_j tends to be a very good approximate solution for relatively small j. This method, known as the *method of conjugate gradients* has found wide applicability. (The above specification is taken from Golub & Van Loan [1989, p. 497]).

A technique known as *preconditioning* is often crucial to the process. The preconditioned conjugate gradient algorithm is very similar to the above iteration only during each step, a vector z_j defined by $Mz_j = r_j$ must be computed. M is the preconditioner and it must satisfy two constraints:

- Linear systems involving M must be easily solved.
- M must approximate A either in the norm sense or in the sense that $M - A$ has low rank.

Further reading

Good 'global' references for the Lanczos algorithm include Parlett [1980], Cullum & Willoughby [1985a,b], and Golub & Van Loan [1989]. The rapid

convergence of the Lanczos process is discussed by Kaniel [1966], Paige [1971, 1980], and Saad [1980]. Practical details associated with the implementation of the Lanczos procedure are discussed by Parlett & Nour-Omid [1985]; Parlett, Simon & Stringer [1982], and Scott [1979].

Interesting papers that pertain to the method of conjugate gradients include Axelsson [1985], Concus, Golub & Meurant [1985], Concus, Golub & O'Leary [1976], and Golub & Meurant [1983]. Representative papers that are concerned with preconditioning include Elman [1986], Manteuffel [1979], Meijerink & Van der Vorst [1977], and Rodrigue & Wolitzer [1984]. The idea of adapting the method to unsymmetric problems is explored by Saad [1981], and Faber & Manteuffel [1984]. Implementation strategies for high performance computers are set forth in Meurant [1984], Poole & Ortega [1987], Seager [1986], and Van der Vorst [1982a,b]. The GMRES scheme due to Saad & Schultz [1986] is one of the most widely used sparse unsymmetric system solvers. The survey by Saad [1989] is recommended for further insight into GMRES and related Ktylov subspace methods.

2.20. The singular value decomposition

Theorem. *If $A \in \mathbb{R}^{m\times n}$ then there exists an orthogonal $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ such that $U^{\mathrm{T}}AV = \Sigma$ is diagonal. The diagonal elements of Σ are the singular values and the columns of U and V are left and right singular vectors.*

Mathematical notes

If $\sigma_r > \sigma_{r+1} = 0$, then $r = \operatorname{rank}(A)$. If $U = [u_1, \ldots, u_m]$ and $V = [v_1, \ldots, v_n]$ are column partitionings, then $\{v_{r+1}, \ldots, v_n\}$ is an orthonormal basis for $\operatorname{null}(A)$ and $\{u_1, \ldots, u_r\}$ is an orthonormal basis for $\operatorname{ran}(A)$. Moreover, if

$$A_{\tilde{r}} = \sum_{j=1}^{\tilde{r}} \frac{u_j v_j^{\mathrm{t}}}{\sigma_j},$$

then $A = A_r$ and $B_{\mathrm{opt}} = A_{\tilde{r}}$ minimizes $\|A - B\|_2$ subject to $\operatorname{rank}(B) = \tilde{r} \leqslant r$.

Algorithmic and numerical notes

A variant of the symmetric QR algorithm is the standard means for computing the SVD of a dense matrix. A is first reduced to bidiagonal form (see Section 2.21) and then the symmetric QR iteration is *implicitly* applied to the tridiagonal matrix $A^{\mathrm{T}}A$.

Jacobi procedures also exist. These involve solving a sequence of 2-by-2 SVD problems which make A progressively more diagonal.

Applications

- The rank deficient least squares problem is handled nicely through the SVD. If $u_i = U(:, i)$ and $v_i = V(:, i)$, then

$$x_{\mathrm{LS}} = \sum_{i=1}^{r} \frac{u_i^{\mathrm{T}} b}{\alpha_i} v_i , \quad r = \mathrm{rank}(A) .$$

• In exact arithmetic, if $r = \mathrm{rank}(A)$, then $\sigma_r > \sigma_{r+1} = \cdots = \sigma_n = 0$. However, this is an unworkable criterion in the face of fuzzy data and inexact arithmetic. So a more realistic approach to numerical rank determination is to choose a parameter ε and then say A has ε-rank $\hat{r}$ if

$$\hat{\sigma}_{\hat{r}} > \hat{\sigma}_1 \varepsilon \geqslant \hat{\sigma}_{\hat{r}+1} ,$$

where the 'hats' designate computed quantities.

• Given A, $B \in \mathbb{R}^{m \times p}$ the problem

$$\text{minimize} \quad \|A - BQ\|_F$$

$$\text{subject to} \quad Q^{\mathrm{T}} Q = I_p$$

is solved by $Q = UV^{\mathrm{T}}$ where $U^{\mathrm{T}} CV$ is the SVD of $C = B^{\mathrm{T}} A$.

• Suppose $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{p \times n}$ are given and that we want to compute an orthonormal basis for $\mathrm{null}(A) \cap \mathrm{null}(B)$. Using the SVD, first compute a matrix V_A whose columns are an orthonormal basis for $\mathrm{null}(A)$. Then use the SVD again to compute a matrix V_C whose orthonormal columns span the null space of $C = BV_A$. The columns of V_C can be shown to span the required null space.

• Suppose the columns of $Q_S \in \mathbb{R}^{n \times p}$ and $Q_T \in \mathbb{R}^{n \times p}$ are orthonormal and define a pair of subspaces S and T. The singular values of $C = Q_S^{\mathrm{T}} Q_T$ are situated in $[0, 1]$ and so we may write $\sigma_i = \cos(\theta_i)$ for $i = 1{:}p$ with $0 \leqslant \theta_1 \leqslant \cdots \leqslant \theta_p \leqslant \frac{1}{2}\pi$. The θ_i are called the *principal angles* between the subspaces S and T. If $0 = \theta_1 = \cdots = \theta_p < \theta_{p+1}$ and $U^{\mathrm{T}} CV = \Sigma$ is the SVD, then $\mathrm{ran}(V(:, 1{:}p)) = \mathrm{ran}(U(:, 1{:}P)) = S \cap T$.

• If error is also present in the 'data matrix' A, then instead of solving the ordinary least squares problem it may be more natural to consider the problem

$$\min_{b+r \in \mathrm{range}(A+E)} \|D[E \quad r]T\|_F , \quad E \in \mathbb{R}^{m \times n}, r \in \mathbb{R}^m , \tag{5}$$

where $D = \mathrm{diag}(d_1, \ldots, d_m)$ and $T = \mathrm{diag}(t_1, \ldots, t_{n+1})$ are nonsingular. This problem is referred to as the *total least squares* (TLS) problem. If $U^{\mathrm{T}} CV = \Sigma$ is the SVD of $C = D[A \quad b]T$, then $[E_{\mathrm{opt}} \quad r_{\mathrm{opt}}] = -\sigma_{n+1} u_{n+1} v_{n+1}^{\mathrm{T}}$ and $(A + E_{\mathrm{opt}}) x_{\mathrm{TLS}} = b + r_{\mathrm{opt}}$, where

$$x_{\mathrm{TLS}} = -V(1{:}n, n+1)/V(n+1, n+1)$$

is the total least squares fit.

Further reading

All aspects of the SVD are discussed by Golub & Van Loan [1989]. The standard means for computing the SVD is a modification of the symmetric QR algorithm for eigenvalues. Appropriate references include Chan [1982], Golub & Kahan [1965], and Golub & Reinsch [1970].

A modification of the symmetric Jacobi algorithm can also be used. See Brent, Luk & Van Loan [1985], Forsythe & Henrici [1960], Kogbetliantz [1955], and Van Dooren & Paige [1986].

Using the SVD to solve the canonical correlation problem was originally proposed by Björck & Golub [1973].

The TLS problem is discussed by Golub & Van Loan [1980], and Van Huffel [1987, 1988]. If some of the columns of A are known exactly, then it is sensible to force the TLS perturbation matrix E to be zero in the same columns. Aspects of this constrained TLS problem are discussed by Van Huffel & Vandewalle [1988].

2.21. Orthogonal bidiagonalization

Theorem. *If $A \in \mathbb{R}^{m\times n}$ then there exist orthogonal $U \in \mathbb{R}^{m\times m}$ and orthogonal $V \in \mathbb{R}^{n\times n}$ such that $U^{T}AV = B$ is upper bidiagonal.*

Mathematical notes

A and B have the same singular values. If A is rank deficient, then B has at least one zero on its diagonal. Note that V tridiagonalizes $A^{T}A$ and U tridiagonalizes AA^{T}.

Algorithmic and numerical notes

The standard method for bidiagonalization uses Householder transformations. The reduction is backwards stable and requires $2mn^2 + n^3$ flops.

2.22. The CS decomposition

Theorem. *If*

$$Q = \begin{matrix} \\ k \\ m-k \end{matrix} \overset{\begin{matrix} k & j \end{matrix}}{\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}}$$

is orthogonal with $k \geq j$, then there exist orthogonal matrices $U_1, V_1 \in \mathbb{R}^{k\times k}$ and orthogonal matrices $U_2, V_2 \in \mathbb{R}^{j\times j}$ such that

$$\begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix}^{T} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} V_1 & 0 \\ 0 & V_2 \end{bmatrix} = \begin{bmatrix} I_{k-j} & 0 & 0 \\ 0 & C & S \\ 0 & -S & C \end{bmatrix},$$

where

$$C = \operatorname{diag}(c_1, \ldots, c_j) \in \mathbb{R}^{j \times j}, \quad c_i = \cos(\theta_i),$$
$$S = \operatorname{diag}(s_1, \ldots, s_j) \in \mathbb{R}^{j \times j}, \quad s_i = \sin(\theta_i),$$

and $0 \leqslant \theta_1 \leqslant \theta_2 \leqslant \cdots \leqslant \theta_j \leqslant \frac{1}{2}\pi$.

Mathematical notes

Paige & Saunders [1981] have explored a variation of the above concerned with the SVDs of the blocks of

$$Q = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix},$$

where Q has orthonormal columns. No assumption that Q_1 be square is required.

Algorithmic and numerical notes

Algorithms for the CS decomposition algorithm may be found in Stewart [1983], and Van Loan [1985a]. SVDs and QR factorizations are involved. They are all backwards stable.

Applications

- The CS decomposition is useful in the analysis of many problems that involve distances between subspaces.
- Some generalized SVD problems can be solved using the Paige–Saunders variant, see Section 2.24.

Further reading

The CS decomposition has many applications associated with subspace nearness. See Davis & Kahan [1970].

2.23. Generalized SVD

Theorem. *If we have* $A \in \mathbb{R}^{m \times n}$ *with* $m \geqslant n$ *and* $B \in \mathbb{R}^{p \times n}$, *then there exist orthogonal* $U \in \mathbb{R}^{m \times m}$ *and* $V \in \mathbb{R}^{p \times p}$ *and an invertible* $X \in \mathbb{R}^{n \times n}$ *such that*

$$U^{\mathrm{T}} A X = C = \operatorname{diag}(c_1, \ldots, c_n), \quad 1 \geqslant c_1 \geqslant \cdots \geqslant c_n \geqslant 0$$

and

$$V^{\mathrm{T}} B X = S = \operatorname{diag}(s_1, \ldots, s_q), \quad s_i \geqslant 0, s_i^2 + c_i^2 = 1,$$

where $q = \min(p, n)$.

Algorithmic and numerical notes

A sequence of carefully chosen QR factorizations and SVDs can be used to compute the generalized SVD. For stability it is sometimes better to compute X^{-1} rather than X, see Section 2.24.

Applications

• Consider the least squares problem with quadratic inequality constraint (LSQI):

$$\begin{aligned} &\text{minimize} && \|Ax - b\|_2 \\ &\text{subject to} && \|Bx - d\|_2 \leq \alpha , \end{aligned}$$

where $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$, $b \in \mathbb{R}^m$, $B \in \mathbb{R}^{p\times n}$, $d \in \mathbb{R}^p$, and $\alpha \geq 0$. The generalized singular value decomposition sheds light on the solvability of the LSQI problem. Indeed, if

$$\begin{aligned} U^{\mathrm{T}}AX &= \operatorname{diag}(\alpha_1, \ldots, \alpha_n) , \qquad U^{\mathrm{T}}U = I_m , \\ V^{\mathrm{T}}BX &= \operatorname{diag}(\beta_1, \ldots, \beta_q) , \qquad V^{\mathrm{T}}V = I_p , \; q = \min\{p, n\} \end{aligned}$$

is the generalized singular value decomposition of A and B, then the original LSQI problem transforms to

$$\begin{aligned} &\text{minimize} && \|D_A y - \tilde{b}\|_2 \\ &\text{subject to} && \|D_B y - \tilde{d}\|_2 \leq \alpha , \end{aligned}$$

where $\tilde{b} = U^{\mathrm{T}}b$, $\tilde{d} = V^{\mathrm{T}}d$, and $y = X^{-1}x$. The simple diagonal form of the objective function and the constraint equation facilitate the analysis of the LSQI problem. Here, $r = \operatorname{rank}(B)$.

• The problem of finding nontrivial solutions to the generalized eigenvalue problem $A^{\mathrm{T}}A - \lambda B^{\mathrm{T}}B$ is called the *generalized singular value problem*. It is easy to show that if we have the above generalized SVD, then the λ we seek are the zeros of $\det(C^{\mathrm{T}}C - \lambda S^{\mathrm{T}}S) = \Pi_{k=0}^{n}(c_k^2 - \lambda s_k^2)$.

Further reading

Various aspects of the generalized singular value decomposition are discussed by Golub & Van Loan [1989], Kågström [1985], Paige [1986], Stewart [1983], Paige & Saunders [1981], and Van Loan [1976].

2.24. *Special topic: Avoiding inverses and cross-products*

We have seen several instances where the avoidance of explicit inverse computation results in methods with superior numerical properties. We would like to expand upon this critical point as it is a symbol of intelligent matrix

computations. We consider a problem that arises in signal processing, see Speiser & Van Loan [1984].

Suppose $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times n}$ with $m, n \geq p$ and that

$$\text{null}(A) \cap \text{null}(B) = \{0\} .$$

This condition can be relaxed but with a loss of clarity. Suppose

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$$

are the generalized eigenvalues of $A^TA - \lambda B^TB$. We wish to find an orthonormal basis for the subspace

$$S_{\min} = \{x\colon A^TAx = \lambda_n B^TBx\} .$$

This is the subspace associated with smallest generalized singular value of the pair (A, B). If

$$U^TAX = C = \text{diag}(c_1, \ldots, c_n) , \quad 0 \leq c_1 \leq \cdots \leq c_n ,$$
$$V^TBX = S = \text{diag}(s_1, \ldots, s_n) , \quad s_1 \geq \cdots \geq s_n \geq 0$$

is the generalized SVD with $c_i^2 + s_i^2 = 1$ and

$$c_1 = \cdots = c_q < c_{q+1} ,$$

then

$$S_{\min} = \text{span}\{x_1, \ldots, x_q\} ,$$

where $X = [x_1, \ldots, x_n]$ is a column partitioning of X. Thus, our problem is solved once we obtain the QR factorization of $X(:, 1{:}q)$.

We proceed as follows. First we compute the QR factorization

$$\begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R ,$$

where Q_1 and Q_2 have the same size as A and B, respectively. Because A and B have trivially intersecting nullspaces, we know that R is nonsingular.

We then compute the Paige–Saunders CS decomposition

$$\begin{bmatrix} U & 0 \\ 0 & V \end{bmatrix}^T \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} Z = \begin{bmatrix} C \\ S \end{bmatrix} ,$$

where $C = \text{diag}(c_i)$ and $S = \text{diag}(s_i)$. Assume that the c_i and s_i are ordered as

above. It follows that

$$\begin{bmatrix} U & 0 \\ 0 & V \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} C \\ S \end{bmatrix} Z^{\mathrm{T}} R ,$$

and so $X = R^{-1}Z$.

Now one approach is to compute the QR factorization of an explicitly formed $X(:, 1{:}q)$. This involves solving $RX(:, 1{:}q) = Z(:, 1{:}q)$ and is thus subject to great error if R is ill-conditioned, i.e., if the nullspaces of A and B nearly intersect. However, we can avoid the inversion of R as follows:

Form $W = Z^{\mathrm{T}}R$. Note that $W = X^{-1}$ and that all we have here is a stable orthogonal matrix times matrix multiplication.

Compute an orthogonal Y such that $WY = T$ is upper triangular. This can be accomplished by a minor modification of the Householder QR process in which the rows of W are zeroed bottom-up via Householder matrices.

Since $X = W^{-1} = (TY^{\mathrm{T}})^{-1} = YT^{-1}$ we see that the first q columns of Y define the required orthonormal basis.

This example is typical of many applications in control theory and signal processing where calculations are put on a much sounder footing through the use of stable matrix factorizations.

3. High performance matrix computations

New computer architectures have brought about a change in how efficient matrix algorithms are designed. *We are now compelled to pay as much attention to the flow of data as to the amount of arithmetic.* A ramification of this is that flop counting is no longer adequate as a mechanism for anticipating performance. A careless implementation of Gaussian elimination (for example) can be ten times slower than one is organized around the careful control of memory traffic.

Our goal is to delineate the scope of the memory management problem in high performance matrix computations. By 'high performance' we mean any computer whose speed when performing an operation is a strong function of data locality. This applies to a multiprocessor where one processor may be forced into idleness as it waits for data that happens to reside somewhere else in the network. But it is also an issue in a uniprocessor with hierarchical memory. The schematic in Figure 1 depicts such a memory system. Movement of data between two levels in the hierarchy represents a communication overhead. A submatrix that climbs its way to the top of the hierarchy should be involved in as much constructive computation as possible before it returns to its niche further down in the memory system.

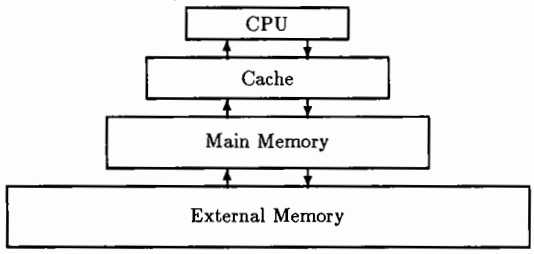

Fig. 1. A hierarchical memory system.

From the standpoint of clarifying these issues, matrix–matrix multiplication and the fast Fourier transform are great teaching algorithms. Both procedures can be arranged in numerous ways and this enables one to resolve many of the computational dilemmas that typify high performance matrix manipulation. However, in this section we have chosen to frame the discussion in terms of the Cholesky factorization, which is simple mathematically and can be organized in several ways like matrix multiplication and the FFT.

Much of what we say carries over to other matrix factorizations and to general scientific computing as well. By the end of the discussion the reader should acquire a practical intuition about high performance computing. We start by deriving several versions of the Cholesky procedure. These highlight the issues of stride and data re-use and motivate four different parallel implementations. The parallel versions of Cholesky that we discuss are chosen for what they reveal about algorithmic thinking in multiprocessor environments.

Before we begin it should be stressed that memory traffic has been a central concern in matrix computations for many years. For example, one can find papers written in the early 1950s that are concerned with the 'out of core' solution of linear systems. However, a hallmark of the current situation is that we must pay attention to memory traffic even for relatively small, dense problems.

3.1. A 'point' derivation of Cholesky

One way to derive the Cholesky algorithm is to compare entries in the equation $A = GG^{\mathrm{T}}$. If $i \geq j$, then

$$a_{ij} = \sum_{k=1}^{j} g_{ik} g_{jk} ,$$

and so

$$g_{ij} g_{jj} = a_{ij} - \sum_{k=1}^{j-1} g_{ik} g_{jk} \equiv s_{ij} .$$

Thus, $g_{ij} = s_{ij}/g_{jj}$ if $i > j$ and $g_{jj} = \sqrt{s_{jj}}$ if $i = j$. By computing the lower triangular matrix G row-by-row we have:

Algorithm 1.
```
for i = 1:n
  for j = 1:i
    s ← A(i, j)
    for k = 1:j − 1
      s ← s − G(i, k)G(j, k)
    end
    if j < i
      G(i, j) ← s/A(j, j)
    else
      G(j, j) ← √s
    end
  end
end
```

A flop count reveals that this implementation of Cholesky involves $\frac{1}{3}n^3 + O(n^2)$ flops.

3.2. Level-1 operations

Notice that the k-loop in Algorithm 1 oversees an inner product between subrows of G. To highlight this and the fact that the lower triangular portion of A can be overwritten with the lower triangular portion of G, we rewrite the algorithm as follows.

Algorithm 1'.
```
for i = 1:n
  for j = 1:i
    s ← A(i, j) − A(i, 1:j − 1)A(j, 1:j − 1)^T
    if j < i
      A(i, j) ← s/A(j, j)
    else
      A(j, j) ← √s
    end
  end
end
```

An inner product is an example of a *level-1* linear algebra operation. Level-1 operations involve $O(n)$ work and $O(n)$ data. Inner products, vector scaling, vector addition, and saxpy's are level-1 operations. (A *saxpy* is a vector operation of the form $y \leftarrow ax + y$ where x and y are vectors and α is a scalar.)

It is important to be able to identify level-1 operations. A class of processors

known as *vector processors* are often able to execute level-1 operations faster than what would be expected from consideration of individual, 'free-standing' scalar operations. For example, the successful vectorization of a length n vector addition $z \leftarrow x + y$ would require much less time than an n-fold increase in the time for a single $z_i \leftarrow x_i + y_i$ operation.

The philosophy of vector processing is identical with the philosophy of an automobile assembly line whose mission is the production of car(1), ..., car(n). It is much more efficient to pipeline the assembly than to assign *all* the workers (functional units) to car(1), then to car(2), etc.

3.3. Stride

Arrays are stored in column major order in Fortran. This means that the entries that define an array column are contiguous in memory while those within a row are not. For example, if A is a matrix stored in a 100-by-50 array, then $A(4,2)$ is 100 memory units 'beyond' $A(4,1)$.

Now observe that the vectors which define the inner product in Algorithm 1' are not contiguous in memory because both of the vectors involved in the operation, i.e., $A(i, 1{:}j-1)$ and $A(j, 1{:}j-1)$, are matrix subrows. If A is stored in an array with row dimension *lda*, then we say that *lda* is the *stride* of these vectors. Unit stride code organization can enhance performance because the cost of accessing n contiguous memory locations that house a vector may be much less than the cost of accessing n individual scalars. To make an analogy, suppose a file cabinet contains a thousand folders. It is easy for a human being to extract adjacent folders 101 through 200 than non-adjacent folders 2, 12, 22, ..., 982, 992.

3.4. A vector derivation

Notice that the nonunit stride problem in Algorithm 1' would be solved if either i or j was the inner loop variable. To derive such a procedure we compare jth columns in the equation $A = GG^{\mathrm{T}}$ and get

$$A(:, j) = \sum_{k=1}^{j} G(:, k)G(j, k) .$$

Focussing on components j through n in this vector equation we obtain

$$G(j{:}n, j)G(j, j) = A(j{:}n, j) - \sum_{k=1}^{j-1} G(j{:}n, k)G(j, k) \equiv v(j{:}n) .$$

Since $G(j, j) = \sqrt{v(j)}$ it follows that $G(j{:}n, j) = v(j{:}n)/\sqrt{v(j)}$ and so with overwriting we have the following specification of the Cholesky process.

Algorithm 2.
for $j = 1{:}n$
$v(j{:}n) \leftarrow A(j{:}n, j)$
for $k = 1{:}j-1$
for $i = j{:}n$
$v(i) \leftarrow v(i) - G(i,k)G(j,k)$
end
end
$G(j{:}n, j) \leftarrow v(j{:}n)/\sqrt{v(j)}$
end

Notice that as i ranges from j to n, the kth column of A is accessed in the inner loop. Thus, Algorithm 2 is a unit stride Cholesky procedure. From the flop point of view, it is identical to Algorithm 1.

3.5. Level-2 operations

Recognize that the inner two loops in Algorithm 2 oversee a matrix–vector product. Indeed, from the derivation of the algorithm we see that

$$v(j{:}n) = A(j{:}n, j) - \begin{bmatrix} G(j,1)\cdots G(j,j-1) \\ \vdots \quad\quad \vdots \\ G(n,1)\cdots G(n,j-1) \end{bmatrix} \begin{bmatrix} G(j,1) \\ \vdots \\ G(j,j-1) \end{bmatrix}$$

$$= A(j{;}n, j) - G(j{:}n, 1{:}j-1)G(j, 1{:}j-1)^{\mathrm{T}}.$$

Substituting this observation into Algorithm 2 gives:

Algorithm 2'.
for $j = 1{:}n$
$v(j{:}n) \leftarrow A(j{:}n, j) - A(j{:}n, 1{:}j-1)A(j, 1{:}j-1)^{\mathrm{T}}$
$A(j{:}n, j) \leftarrow v(j{:}n)/\sqrt{v(j)}$
end

Matrix–vector multiplication is a *level-2* operation. Such operations are characterized by quadratic work and quadratic data, e.g., for m-by-n matrix–vector multiplication, O(mn) data and O(mn) flops are involved.

From a certain standpoint, a matrix–vector product is 'just a bunch' of level-1 saxpy operations. Indeed, the computation of

$$u \leftarrow u + Av = u + \sum_{j=1}^{n} v(j)A(:, j)$$

has the form:

for $j = 1{:}n$
  $u \leftarrow u + v(j)A(:, j)$
end

However, one should think of u as a *vector accumulator*. It is a running vector sum which can be built up in a vector register without any writing to lower level memory until the summation is complete. Thus, there are $n + 1$ vector reads and only one vector write. In contrast, if each saxpy is executed without an awareness of the overall mission of the loop, then $2n$ vector reads and $2n$ vector writes are involved.

A matrix–vector product is referred to as a *gaxpy* operation. Use of the term tacitly implies vector accumulation as discussed above.

3.6. *An inductive derivation*

Consider the following blocking of the matrix equation $A = GG^{\mathrm{T}}$:

$$\begin{bmatrix} \alpha & w^{\mathrm{T}} \\ w & B \end{bmatrix} = \begin{bmatrix} \beta & 0 \\ v & G_1 \end{bmatrix} \begin{bmatrix} \beta & 0 \\ v & G_1 \end{bmatrix}^{\mathrm{T}}.$$

Here $\alpha, \beta \in \mathbb{R}$, $w, v \in \mathbb{R}^{n-1}$, and $B, G_1 \in \mathbb{R}^{(n-1)\times(n-1)}$. We illustrate a third 'methodology' for developing a matrix factorization algorithm by equating entries in the above:

$$\begin{aligned} \alpha &= \beta^2 && \Rightarrow \beta = \sqrt{\alpha}\,, \\ w &= \beta v && \Rightarrow v = w/\beta\,, \\ B &= vv^{\mathrm{T}} + G_1 G_1^{\mathrm{T}} && \Rightarrow G_1 G_1^{\mathrm{T}} = B - vv^{\mathrm{T}}\,. \end{aligned}$$

It can be shown that the symmetric matrix $B - vv^{\mathrm{T}}$ is positive definite and so by induction on n we can find its Cholesky factor G_1 as required above.

Note that $G(2{:}n, 2) = G_1(1{:}n-1, 1)$. Thus, repitition of the square root, the scaling, and the update of B leads to yet another formulation of Cholesky:

Algorithm 3.
for $k = 1{:}n$
  $G(k{:}n, k) \leftarrow A(k{:}n, k)/\sqrt{A(k, k)}$
  for $j = k + 1{:}n$
    for $i = j{:}n$
      $A(i, j) \leftarrow A(i, j) - G(i, k)G(j, k)$
    end
  end
end

The inner loop oversees a unit stride saxpy. The overall procedure involves exactly the same amount of floating point arithmetic as Algorithms 1 and 2.

3.7. Outer product

The inner two loops of Algorithm 3 feature a level-2 operation referred to as an *outer product*. If $A \in \mathbb{R}^{m \times n}$, $u \in \mathbb{R}^m$, and $v \in \mathbb{R}^n$, then an update of the form $A \leftarrow A \pm uv^T$ is an outer product update. Notice that this involves $2n$ vector reads and $2n$ writes. Each column is an outer product update is a saxpy $[A(:, j) \leftarrow A(:, j) \pm v(j)u]$ and there is *no* opportunity for vector accumulation in contrast to the gaxpy operation.

Algorithm 3 features a *symmetric outer product update*, i.e., an update of the form $B \leftarrow B \pm uu^T$ where $B = B^T$. To emphasize this we rewrite Algorithm 3 and incorporate overwriting:

Algorithm 3'.
for $k = 1{:}n$
$u(k{:}n) \leftarrow A(k{:}n, k)/\sqrt{A(k, k)}$
$A(k{:}n, k) \leftarrow u(k{:}n)$
$A(k+1{:}n, k+1{:}n) \leftarrow A(k+1{:}n, k+1{:}n) - u(k+1{:}n)u(k+1{:}n)^T$
end

We assume that symmetry is exploited during the symmetric outer product update of the submatrix $A(k+1{:}n, k+1{:}n)$.

3.8. Loop reorderings

Algorithms 1, 2, and 3 are identical with respect to arithmetic but different with respect to the ordering of the three loops. For example, Algorithm 1 can be through of as *ijk* Cholesky and Algorithms 2 and 3 are the *jki* and *kji* variants, respectively. There are actually three other versions: *jik* (via permutation of the first two loops in Algorithm 1), *ijk* (the row-by-row analog of Algorithm 2), and *kij* (which computes the outer product updates by row in Algorithm 3). Each loop ordering features certain linear algebraic operations and has distinct memory reference patterns. In general, the *jki* version (Algorithm 2) has the best attributes in terms of stride and the potential for vector accumulation.

3.9. Recursion

The notion of a *recursive procedure* is increasingly important in matrix computations. Roughly speaking, a recursive procedure is able to call itself.

Recursion makes the specification of some matrix algorithms particularly concise. Here is a recursive version of Algorithm 3 stated in quasi-Matlab style:

function $G = \mathbf{chol}(A, n)$
  $G(1,1) \leftarrow \sqrt{(A(1,1)}$
  if $n > 1$
    $G(2{:}n, 1) \leftarrow A(2{:}n, 1)/G(1,1)$
    $G(2{:}n, 2{:}n) \leftarrow \mathbf{chol}(A(2{:}n, 2{:}n) - G(2{:}n, 1)G(2{:}n, 1)^{\mathrm{T}}, n-1)$
  end
end chol

Leading examples of recursion in matrix computations include Strassen matrix multiply, cyclic reduction, and the Cuppen–Dongarra–Sorensen divide and conquer algorithm for the symmetric tridiagonal eigenproblem. These procedures are all discussed by Golub & Van Loan [1989].

3.10. Level-3 operations

For many architectures, matrix–matrix multiplication is the operation of choice. There are two reasons for this: it is a highly parallelizable computation and it has a favorable 'computation-to-communication' ratio. The latter aspect needs additional comment. Consider the n-by-n matrix multiplication $C = AB$. This computation involves the reading and writing of $O(n^2)$ data. However, $2n^3$ flops are expended. Thus, in a loose manner of speaking, the ratio of arithmetic to data movement is $O(n)$. As n grows the overhead associated with the accessing of data diminishes. This explains the current interest in *block matrix algorithms* by which we mean algorithms that are rich in matrix multiplication. Matrix operations that involve $O(n^2)$ data and $O(n^3)$ work are referred to as *level-3* operations. Matrix multiplication updates of the form $C \leftarrow \alpha AB + \beta C$ and the multiple triangular system solve $C \leftarrow \alpha T^{-1}C$ are examples of frequently occurring level-3 operations. (Here, α and β are scalars and T is triangular.) See Golub & Van Loan [1989] for a discussion of the latter problem and why it is rich in matrix multiplication.

In many computing environments, level-3 operations can be executed at near peak speed in contrast to matrix/vector computations at the first and second level. Thus, the goal of designing good block algorithms is the extraction of *level-3 performance* from the underlying architecture.

3.11. Block Cholesky

We show how to organize the Cholesky computation so that all but a small fraction of the arithmetic occurs in the context of matrix multiplication. This rather surprising result can be accomplished in several ways. For simplicity, we have chosen to develop a block version of Algorithm 1. Assume for clarity that

$n = Nr$ and partition A and G into r-by-r blocks as follows:

$$A = \begin{bmatrix} A_{11} \cdots A_{1N} \\ \vdots \quad \vdots \\ A_{N1} \cdots A_{NN} \end{bmatrix}, \qquad G = \begin{bmatrix} G_{11} \cdots 0 \\ \vdots \quad \ddots \quad \vdots \\ G_{N1} \cdots G_{NN} \end{bmatrix}.$$

Comparing (i, j) blocks in the equation $A = GG^{\mathrm{T}}$ with $i \geqslant j$ gives

$$A_{ij} = \sum_{k=1}^{j} G_{ik} G_{jk}^{\mathrm{T}},$$

and so

$$G_{ij} G_{jj}^{\mathrm{T}} = A_{ij} - \sum_{k=1}^{j-1} G_{ik} G_{jk}^{\mathrm{T}} \equiv S_{ij}.$$

Corresponding to the derivation of Algorithm 1 we see that G_{ij} is the solution of $XG_{jj}^{\mathrm{T}} = S_{ij}$ if $i > j$ and G_{jj} is the Cholesky factor of S_{jj}. We therefore obtain:

Algorithm 4.

```
for i = 1:N
  for j = 1:i
    S ← A_ij
    for k = 1:j − 1
      S ← S − G_ik G_jk^T
    end
    if j < i
      Solve G_ij G_jj^T = S for G_ij
    else
      Compute the Cholesky factorization S = G_jj G_jj^T
    end
  end
end
```

Notice again that each subdiagonal block is the solution of a multiple right-hand side triangular system while the diagonal blocks are obtained as r-by-r Cholesky factorizations. Most importantly, the matrix S is the consequence of a matrix–matrix product

$$S_{ij} = A_{ij} - [G_{i1}, \ldots, G_{i,j-1}][G_{j1}, \ldots, G_{j,j-1}]^{\mathrm{T}}.$$

Algorithm 4 involves the same number of flops as Algorithms 1, 2, and 3. The only flops that are *not* level-3 flops are those associated with the computation of the r-by-r Cholesky factors $G_{11}, \ldots, G_{NN}$. This accounts for approxi-

mately $Nr^3/3$ flops. Thus, the fraction of flops that are associated with level-3 computation is given by:

$$L_3 \approx 1 - \frac{Nr^3/3}{n^3/3} = 1 - \frac{1}{N^2} .$$

We say that L_3 is the *level-3 fraction* associated with Algorithm 4.

A tacit assumption in all this is that the block size r is large enough so that true level-3 performance is extracted during the computation of S. Intelligent block size determination is a function of algorithm and architecture and typically involves careful experimentation.

3.12. Block algorithm development: The LAPACK Project

The search for good block algorithms is an active area of current research. Block algorithms that are rich in matrix multiplication like the above Cholesky procedure have been found for

- The LU factorization with or without partial pivoting.
- The Bunch–Kaufman and Aasen factorizations.
- The QR factorization with and without pivoting. [Bischof & Van Loan, 1987; Bischof, 1988.]

The best block algorithm for the Hessenberg reduction has a level-3 fraction equal to one-half. However, with some regrouping of the Householder transformations significant improvements to the standard level-2 procedure can be realized. To date, no effective block algorithms (in the level-3 sense) have been found for the QR family of iterations (QR, SVD, QZ, etc.) although again, it is possible to improve upon level-2 performance via some clustering of the matrix–vector operations.

The mission of the LAPACK Project is to develop a library of level-3 algorithms for the LINPACK and EISPACK codes. These packages handle linear systems and the eigenproblem, respectively. The efficient implementation of this block library on a particular machine requires an optimized *level-3 BLAS library*. 'BLAS' stands for Basic Linear Algebra Subprograms. Level-1 and 2 BLAS libraries exist as well. Details may be found in Dongarra, du Croz, Hammarling & Hanson [1988a,b], Dongarra, du Croz, Duff & Hammarling [1988], Lawson, Hanson, Kincaid & Krogh [1979a,b], and Kågström, Ling & Van Loan [1991].

Further elaboration of the notion of 'level' and what it means for the design of efficient matrix code is discussed by Dongarra, Gustavson & Karp [1984], Gallivan, Jalby & Meier [1987], Gallivan, Jalby, Meier & Sameh [1988], and Kågström & Ling [1988].

The high-level LAPACK codes for things like Cholesky are written in terms of these BLAS and 'automatically' perform well if the underlying level-3 fraction is close to 1 *and* the level-3 BLA routines are fine-tuned to the underlying architecture.

3.13. Parallel computation

We next discuss the parallel computation of the Cholesky factorization. Two types of multiprocessors are considered: those with *shared memory* and those with *distributed memory*. In either case, an individual processor (sometimes called a *node*) comes complete with processing units and its own *local memory*. In a shared memory machine each individual processor is able to read and write to a typically large *global memory*. This enables one processor to communicate with another. A distributed memory machine has no global memory. Instead, each node is connected to some subset of the remaining nodes. The overall *distributed network* supports communication via the routing of messages between nodes.

Throughout the remainder of this section, p will stand for the number of processors in the system. For us, the act of designing a parallel algorithm is the act of designing a *node algorithm* for each of the participating processors. We suppress very important details such as (a) the downloading of data and programs into the nodes, (b) the subscript computations associated with local array access, and (c) the formatting of messages. We designate the μth processor by Proc(μ). The μth node program is usually a function of μ. For example, in the n-by-n matrix–vector product problem $y = Ax$, Proc(μ) might be assigned the computation of $y(\mu:p:n) = A(\mu:p:n, :)x$.

To illustrate the differences and similarities between shared and distributed memory computing and the crucial role of block matrix notation, we consider the two-processor calculation of the n-by-n matrix multiply update $C \leftarrow C + AB$. Assume $n = 2m$ and consider the block matrix equation

$$[C_1 \quad C_2] \leftarrow [C_1 \quad C_2] + [A_1 \quad A_2] \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

where we assume $A_1, A_2, C_1, C_2 \in \mathbb{R}^{n\times m}$ and $B_{11}, B_{12}, B_{21}, B_{22} \in \mathbb{R}^{m\times m}$ are situated in global memory. A shared memory implementation can be organized as follows:

Proc(1):	**Proc(2):**
$C_{\text{loc}} \leftarrow C_1$	$C_{\text{loc}} \leftarrow C_2$
$A_{\text{loc}} \leftarrow A_1$	$A_{\text{loc}} \leftarrow A_2$
$B_{\text{loc}} \leftarrow B_{11}$	$B_{\text{loc}} \leftarrow B_{22}$
$C_{\text{loc}} \leftarrow C_{\text{loc}} + A_{\text{loc}}B_{\text{loc}}$	$C_{\text{loc}} \leftarrow C_{\text{loc}} + A_{\text{loc}}B_{\text{loc}}$
$A_{\text{loc}} \leftarrow A_2$	$A_{\text{loc}} \leftarrow A_1$
$B_{\text{loc}} \leftarrow B_{12}$	$B_{\text{loc}} \leftarrow B_{21}$
$C_{\text{loc}} \leftarrow C_{\text{loc}} + A_{\text{loc}}B_{\text{loc}}$	$C_{\text{loc}} \leftarrow C_{\text{loc}} + A_{\text{loc}}B_{\text{loc}}$
$C_1 \leftarrow C_{\text{loc}}$	$C_2 \leftarrow C_{\text{loc}}$

The 'loc' subscript is used to indicate local arrays. Notice that each processor

has 5 global-to-local matrix reads, two matrix multiplies, and a single local-to-global matrix write.

A distributed memory procedure is similar. However, instead of data flowing back and forth between the local memories and the global memory, it moves along the channels that connect the processors themselves. The primitives **send** and **recv** are used for this purpose and they have the following syntax:

$$\textbf{send}(\{matrix\}, \{destination\ node\}) \qquad \textbf{recv}(\{matrix\}, \{source\ node\})$$

If a processor invokes a **send**, then we assume that execution resumes immediately after the message is sent. If a **recv** is encountered, then we assume that execution of the node program is suspended until the requested messages arrives. We also assume that messages arrive in the same order that they are sent.

Now in our $p = 2$ problem let us assume that for $j = 1{:}2$, Proc(j) houses C_j, A_j, and

$$\begin{bmatrix} B_{1j} \\ B_{2j} \end{bmatrix}$$

in the local arrays C_{loc}, A_{loc}, and B_{loc}. We then have:

Proc(1):	**Proc(2):**
$C_{\text{loc}} \leftarrow C_{\text{loc}} + A_{\text{loc}}B_{\text{loc}}(1{:}m, :)$	$C_{\text{loc}} \leftarrow C_{\text{loc}} + A_{\text{loc}}B_{\text{loc}}(m+1{:}n, :)$
send(A_{loc}, 2)	**send**(A_{loc}, 1)
recv(A_{loc}, 2)	**recv**(A_{loc}, 1)
$C_{\text{loc}} \leftarrow C_{\text{loc}} + A_{\text{loc}}B_{\text{loc}}(m+1{:}n, :)$	$C_{\text{loc}} \leftarrow C_{\text{loc}} + A_{\text{loc}}B_{\text{loc}}(1{:}m, :)$
send(A_{loc}, 2)	**send**(A_{loc}, 1)
recv(A_{loc}, 2)	**recv**(A_{loc}, 1)

The last **send**/**recv** is executed so that upon completion, A is distributed in its original manner. This could be avoided with the introduction of additional local workspaces. However, one tends to be stingy with work spaces in distributed environments because local memory is often limited.

Notice that in the distributed memory case, each processor has two **send**s, two **recv**s, and two matrix multiplies.

We now make a few comments about parallel computations in general using the above parallel algorithms for illustration. We say that a parallel procedure is *load balanced* if each processor has roughly the same amount of arithmetic and communication. The above procedures have this property. However, this would not be the case if $B_{12} = 0$ for although both processors would be equally loaded, the B_{12} computations are superfluous. Hence, a revision of the node programs would be required so that each node had equal amounts of *meaningful* work and communication.

The *speed-up* associated with a parallel program is a quotient:

$$\text{speed-up} = \frac{\text{time required by the best single-processor program}}{\text{time required for the } p\text{-processor implementation}}.$$

In this definition we do not just set the numerator to be the $p = 1$ version of the parallel code because the best uniprocessor algorithm may not parallelize. Ideally, one would like the speed-up for an algorithm to equal p. In the above 2-processor example, speed-up will approach 2 as the communication overhead is reduced compared to the time required for the actual matrix multiplication. In particular, if the time for communication is proportional to $O(n^2)$, then speed-up approaches 2 as n gets large.

At times it is sufficient to quantify communication overheads in an order of magnitude sense. On other occasions we need a little more precision. To that end we assume that the communication of a length r floating point vector requires $\alpha + \beta r$ seconds. Here, α represents a start-up overhead and β reflects the rate of transmission. The communication may be between local and global memory, between main memory and a disk, or between two neighbor processors in a distributed network. The values of α and β relative to computation speed have a great effect on the design of a parallel matrix code.

Before we go on to illustrate Cholesky in shared and distributed memory environments, we mention that in practice one often has to work with a system that embodies a mixture of these multiprocessor designs. However, our goal is to paint the parallel matrix computation picture with broad strokes capturing just the central ideas. Nothing much is lost by considering the purely shared and purely distributed systems.

For additional overviews of the parallel matrix computation area, see Dongarra & Sorensen [1986], Gallivan, Plemmons & Sameh [1990], Heller [1978], Hockney & Jesshope [1988], and Ortega & Voigt [1985]. See also Golub & Van Loan [1989, Chapter 6].

3.14. Shared memory computation

A schematic of a shared memory multiprocessor is given in Figure 2. During execution, data flows back and forth between the nodes and the global memory. All computation takes place on data situated in the node.

Two shared memory implementations of the Cholesky factorization are

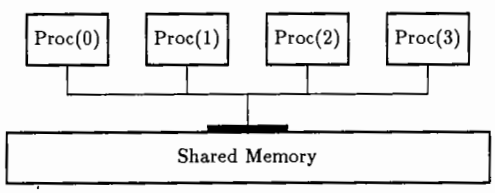

Fig. 2. A 4-processor shared memory system.

given. In the first each processor's portion of the overall computation is determined in advance. This is an example of *static scheduling*. The strategy in the second implementation is different in that the overall computation is broken down into a sequence of tasks with *no* a priori assignment of tasks to processors. A queue of remaining tasks is then maintained during the computation. When a processor finishes a task it goes to the queue and obtains (if possible) the next available task. This is an example of the *pool-of-tasks* approach to *dynamic scheduling*.

3.15. Shared memory Cholesky with static scheduling

Suppose $B = B^T \in \mathbb{R}^{m\times m}$ and $v \in \mathbb{R}^m$ reside in a global memory that is accessible to p processors and that we wish to overwrite B with $B - vv^T$. One way to approach the parallel computation of this symmetric outer product update is to have Proc(μ) update the columns designated by the integer vector $\mu{:}p{:}m$:

$v_{\text{loc}} \leftarrow v$
for $j = \mu{:}p{:}m$
  $b_{\text{loc}}(j{:}m) \leftarrow B(j{:}m, j)$
  $b_{\text{loc}}(j{:}m) \leftarrow b_{\text{loc}}(j{:}m) - v_{\text{loc}}(j{:}m)v_{\text{loc}}(j)$
  $B(j{:}m, j) \leftarrow b_{\text{loc}}(j{:}m)$
end

A few comments are in order. Each processor needs a copy of v and this is accomplished outside the loop with the global to local assignment $v_{\text{loc}} \leftarrow v$. Within the j-loop we see repetition of the following activity:

- The *global to local reading* of the original $B(j{:}m, j)$ into $b_{\text{loc}}(j{:}m)$.
- The *local computation* of the saxpy $b_{\text{loc}}(j{:}m) \leftarrow b_{\text{loc}}(j{:}m) - v_{\text{loc}}(j{:}m)v_{\text{loc}}(j)$.
- The *local to global writing* of $b_{\text{loc}}(j{:}m)$ into $B(j{:}m, j)$.

This 3-step pattern of reading from global memory, local computation, and writing to global memory is typical. The reading and the writing represent communication overheads and hence, one tries to design shared memory procedures so that the amount of local computation is significant compared to the amount of data that goes back and forth between the local memories and shared memory.

Our parallel outer product update is load balanced. To see this, assume $m = pM$ for clarity and note that jth saxpy involves $2(m - j + 1)$ flops. Thus, Proc(μ) must perform

$$\sum_{j=\mu:p:m} 2(m-j+1) = \frac{m^2}{p} + m\left(3 + \frac{2(1-\mu)}{p}\right) \approx \frac{m^2}{p}$$

flops, i.e., one pth of the arithmetic.

Proc(μ)'s communication overhead is proportional to

$$\alpha \frac{2m}{p} + \beta\left(\frac{m^2}{p} + m\left(\frac{2(1-\mu)}{p} + 1\right)\right) \approx \alpha \frac{2m}{p} + \beta\left(\frac{m^2}{p}\right),$$

where α and β capture the nature of communication between local and global memory. Because this is independent of μ we see that the overall procedure is load balanced from the communication point of view.

Note that we could not make these loading balancing claims if Proc(μ) is assigned the update of contiguous columns, i.e., $B(j{:}m, j)$ for $j = (\mu - 1)M + 1{:}\mu M$. This is because the work and communication associated with the jth saxpy is a decreasing function of j.

We now apply these ideas to the design of a shared memory version of outer product Cholesky (Algorithm 3). Recall that this algorithm is structured as follows:

for $k = 1{:}n$
  Perform a square root and a vector scale
  Perform a symmetric outer product update of $A(k+1{:}n, k+1{:}n)$
end

Our strategy is to let a single designated processor handle the square root and the scaling and to let all the processors participate in the outer product updates as discussed above. However, some synchronization is necessary so that the correct Cholesky factor emerges:

- A processor cannot begin its share of the kth outer product update until the kth column of G is available.
- The computation of $G(k+1{:}n, k+1)$ should not begin until the kth outer product update is completed.

To ensure that these 'rules' are enforced we use the **barrier** construct in the following algorithm.

Algorithm 5.

```
for k = 1:n
  if μ = 1
    g_loc(k:n) ← A(k:n, k)
    g_loc(k:n) ← g_loc(k:n)/√g_loc(k)
    A(k:n, k) ← g_loc(k:n)
  end
  barrier
  v_loc(k+μ:n) ← A(k+μ:n, k)
  for j = (k+μ):p:n
    a_loc(j:n) ← A(j:n, j)
    a_loc(j:n) ← a_loc(j:n) − v_loc(j:n)v_loc(j)
    A(j:n, j) ← a_loc(j:n)
  end
  barrier
end
```

This procedure overwrites the lower triangular portion of A (in global memory) with the lower triangular portion of G.

Here is how the **barrier** construct works. When a processor encounters a **barrier** during the execution of its node program, computation is suspended. It resumes as soon as *every* other processor reaches its **barrier**. In that sense the **barrier** is like a stream to be traversed by p hikers. For safety, no one proceeds across the stream until *all* p hikers arrive at its edge. In Algorithm 5, the first **barrier** ensures that the kth outer product update does not begin until $G(k+1{:}n, k)$ is ready. The second **barrier** forces all the processors to wait until the kth update is completely finished before work commences with the next update.

Algorithm 5 is load balanced for the same reasons that the above outer product update algorithm is load balanced. (The square root and scaling assigned to Proc(1) do not appreciably overload this processor so long as $n \gg p$, a reasonable assumption.)

3.16. Shared memory Cholesky with dynamic scheduling

For some simple computations static scheduling is adequate. However, it is important to recognize that a **barrier** forces global idleness and therefore fosters a certain inefficiency. For this reason, dynamic scheduling using the pool-of-task paradigm is often attractive. This style of shared memory programming begins with an identification of computational tasks. The tasks are then executed in a prescribed order by the individual processors. Various synchronization problems must typically be solved along the way.

We illustrate this by implementing Algorithm 4, block Cholesky. We refer to the computation of the Cholesky block G_{ij} as task(i, j). Following the order of G_{ij} resolution in Algorithm 4, we perform the tasks in the following order:

$$(1,1),(2,1),(2,2),(3,1),(3,2),(3,3),\ldots,(N,1),$$
$$\ldots,(N,N-1),(N,N)\,.$$

We assume the existence of a function **next.task** that is invoked by a processor whenever it wants to acquire a new task. A pair of special pointers **row** and **col** is updated with each call, i.e.,

```
function (i, j) = next.task(·)
  col ← col + 1
  if col = N + 1
    row ← row + 1; col ← 1
  end
  i ← row; j ← col
end next.task
```

If we assume that **row** = 1 and **col** = 0 at the start and that at any instant only one processor can be executing **next.task**, then we can structure each node program as follows:

```
(i, j) ← next.task(·)
while i ≤ N
  Perform task(i, j)
  (i, j) ← next.task(·)
end
```

The assumption that only one processor at a time can be executing **next.task** ensures that each task is assigned to exactly one processor. A node program terminates as soon as **row** has value $N+1$. This happens when a task is requested after task(N, N) is assigned. Thus, all node programs eventually terminate.

Now returning to the development of a parallel version of Algorithm 4, recall that task(i, j) requires the computation of

$$S_{ij} = A_{ij} - \sum_{k=1}^{j-1} G_{ik} G_{jk}^{\mathrm{T}} .$$

Note that a simple summation of the form

```
S_loc ← A_ij
for k = 1:j − 1
  B_loc ← A_ik
  C_loc ← A_jk
  S_loc ← S_loc − B_loc C_loc^T
end
```

is not guaranteed to work since we have no way of knowing that A_{ik} and A_{jk} actually house G_{ik} and G_{jk} at the time they are requested from global memory. For this reason we assume the existence of a function **get.block**(q, r) which returns the contents of A_{qr} *only* if it houses G_{qr}. If G_{qr} is not available, then the invoking processor is placed in a state of idle waiting to be broken only when the requested block is available. When this happens the requested G-block is read into local memory and execution resumes with the next statement in the node program.

One way to implement this scheme is for **get.block** to maintain a binary scoreboard *done*$(1{:}N, 1{:}N)$ with the convention that *done*(q, r) is zero or one depending upon the availability of G_{qr}.

We are not quite ready to specify the overall procedure in that we have yet to indicate how the scoreboard is updated. Once a processor has computed the matrix sum S_{ij} it proceeds with the computation of G_{ij}. If $i = j$ then G_{jj} is the Cholesky factor of S_{ij} and a local computation ensues. If $j < i$ then G_{jj} is

required for the resolution of $G_{ij}G_{jj}^{T} = S_{ij}$ and **get.block** must be invoked. In either case once G_{ij} is computed it can be written to global memory. For this process we need another function **put.block**$(i, j, \{local\ matrix\})$ which writes the specified local matrix to a shared memory block A_{ij}. In addition, **put.block** updates the scoreboard by changing the value of $done(i, j)$ from a zero to one. Combining all these ideas and assuming that A is to be overwritten by G in global memory, we obtain the following node program:

Algorithm 6.

$(i, j) \leftarrow$ **next.task**$(\cdot)$
while $i \leqslant N$
  $S_{loc} \leftarrow A_{ij}$
  for $k = 1{:}j - 1$
    $B_{loc} \leftarrow$ **get.block**(i, k)
    $C_{loc} \leftarrow$ **get.block**(j, k)
    $S_{loc} \leftarrow S_{loc} - B_{loc}C_{loc}^{T}$
  end
  if $j < i$
    $B_{loc} \leftarrow$ **get.block**(j, j)
    Solve $XB_{loc} = S_{loc}$ for X; $B_{loc} \leftarrow X$
  else
    Compute the Cholesky factorization $XX^{T} = S_{loc}$; $B_{loc} \leftarrow X$
  end
  $A_{ij} \leftarrow$ **put.block**(i, j, B_{loc})
  $(i, j) \leftarrow$ **next.task**$(\cdot)$
end

As with the updating of the task indices **row** and **col** in shared memory, the updating of the scoreboard must be carefully controlled. In particular, we must never allow the scoreboard to be manipulated by more than one processor at a time. These restrictions coupled with the ability of **get.block** and **put.block** to purposely delay an invoking processor imply that these functions are more than just functions in the sense of $\sin(\theta)$. The 'computer science' aspects associated with the design of these synchronization tools (and the **barrier** as well) are discussed by Andrews & Schneider [1983], and Boyle, Butler, Disz, Glickfield, Lusk, Overbeek, Patterson & Stevens [1987].

3.17. *Comments*

We have stepped through the design of two shared memory Cholesky procedures. While we have depicted the kind of logic associated with shared memory computing, we are a long way from being able to predict performance. Although we can model the rate of floating point arithmetic and the time required for local–global communication, we suppressed a number of important hardware factors such as the bandwidth between local and global

memory. Without knowledge of the local/global memory interface we cannot anticipate what happens when two nodes wish to access global memory at the same time.

We also have made no attempt to quantify the synchronization overheads, i.e., the cost of invoking the **barrier**'s in Algorithm 5 and the 'synchronization functions' **next.task**, **get.block**, and **put.block** in Algorithm 6. An understanding of these factors would most likely be acquired through intelligent experimentation.

Further references for the shared memory computation of matrix factorizations include Chen, Dongarra & Hsuing [1984], Dongarra & Hewitt [1986], Dongarra & Hiromoto [1984], Dongarra, Kaufman & Hammarling [1986], Dongarra, Sameh & Sorensen [1986], and George, Heath & Liu [1986]. See also Golub & Van Loan [1989].

3.18. Distributed memory computing

We now turn out attention to the design of matrix algorithms in a distributed memory environment. Of central importance is the *topology* of the underlying processor network, i.e., how the processors are interconnected. Popular schemes include the ring, the mesh, the tree, the torus, and the hypercube. In this section we consider the implementation of Cholesky on a ring. In this situation, each processor has a left and a right neighbor (see Figure 3). For

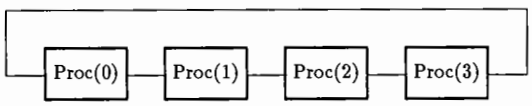

Fig. 3. A 4-processor ring.

notational convenience we assume the existence of local integer variables *left* and *right* so that commands of the form

send({local matrix}, *left*)

send({local matrix}, *right*)

instruct the executing node to send the named local matrix to the left and right neighbors, respectively. (Similarly for **recv**.) Thus, if $p = 4$ and Proc(2) executes **send**(A_{loc}, *left*), then a copy of A_{loc} is send to Proc(1). (Proc(2) retains a copy of A_{loc}.)

The distributed Cholesky procedures that we are about to develop involve only nearest-neighbor communication meaning that whenever a message is sent, it is always to a neighbor in the network. This minimizes reliance on the *routing algorithms* that oversee the message passing and generally helps to control communication overheads.

For any network topology there are usually several natural *distributed data*

structures for matrices and vectors. Consider the storage of

$$A = [A_1 \quad \cdots \quad A_N], \quad A_i \in \mathbb{R}^{n \times n_i}, n_i + \cdots + n_N = n,$$

on a p-processor ring. With a *wrap block column* data structure Proc(μ) houses block columns A_i, where $i = \mu{:}p{:}N$. Note that if $N = p$, then each processor houses a contiguous set of columns, namely, the columns of A_μ. If $N = n$, then A has a *wrap column* distribution.

3.19. *Distributed memory Cholesky on ring*

We now develop a ring implementation of Algorithm 2. Recall that in that version of Cholesky, the generation of $G(j{:}n, j)$ requires the computation of

$$s_j = A(j{:}n, j) - \sum_{k=1}^{j-1} G(j{:}n, k)G(j, k),$$

followed by the scaling $G(j{:}n, j) \leftarrow s_j/\sqrt{s_j(1)}$.

Assume for clarity that $n = pN$ and that Proc(μ) initially houses $A(i{:}n, i)$ for $i = \mu{:}p{:}n$ in a local array A_{loc}. The mission of Proc(μ) in our ring Cholesky procedure is to overwrite this array with the nonzero portion of $G(:, \mu{:}p{:}n)$. This involves using the columns of A_{loc} as running vector sums for building up the s_j that are to be used locally.

Notice that a G-column generated by one processor is generally needed by all the other processors. So suppose Proc(μ) generates $G(j{:}n, j)$. This vector is then circulated around the ring in merry-go-round fashion 'stopping' at Proc($\mu + 1$), ..., Proc(p), Proc(1), ..., Proc($\mu - 1$) in turn. At each stop the visiting G-column is incorporated into all the local s_k for which $k \geq j$.

By counting the number of received G-columns a processor can determine whether or not it is ready to take its turn in G-column generation. For example, if Proc(μ) has received $k - 1$ G-columns and $k \in \{\mu, \mu + p, \ldots, \mu + (N-1)p\}$, then it 'knows' that it is time to generate $G(k{:}n, k)$.

Here is the program to be executed by Proc(μ). It assumes that the local variables *left* and *right* contain the indices of the left and right neighbor and that $n = pN$.

Algorithm

$k \leftarrow 0$; $j \leftarrow 0$
last $\leftarrow \mu + (N-1)p$
$\tilde{j} \leftarrow 1$
while $j \neq$ *last*
  if $k + 1 \in \{\mu, \mu + p, \ldots, last\}$
    $j \leftarrow k + 1$
    Generate $G(j{:}n, j)$ and copy into $g_{\text{loc}}(j{:}n)$ and $A_{\text{loc}}(j{:}n, \tilde{j})$

```
      if j < n
        send(g_loc(j:n), right)
        Update A_loc(:, j̃ + 1:N)
        k ← k + 1
      end
      j̃ ← j̃ + 1
    else
      recv(g_loc(k:n), left)
      if k ∉ {right, right + p, ..., right + (N − 1)p}
        send(g_loc(k:n), right)
      end
      k ← k + 1
      Update A_loc(:, j̃:N)
    end
  end
```

We make a few observations about this node program:

- The number of received G-column updates is maintained in the variable k. Note that k is incremented when a G-column is received and when a G-column is locally generated.
- The index of the last locally generated G-column is maintained in j. If $j = last$, then there is nothing let for Proc(μ) to do.
- $\tilde{j}$ points to the next column of A_{loc} that will produce a G-column.
- A circulating G-column is not sent to Proc(*right*) if Proc(*right*) is the generator of the G-column.

Notice that when early g-columns circulate, the local computation associated with each message is of order $O(n^2/p)$. However, as the algorithm proceeds the circulating g-columns get shorter and there are fewer local columns to update. Thus, towards the end of the algorithm, $O(k)$ flops are performed per received g-column of length $O(k)$. The unfavorable computation/communication balance can be improved by implementing a *block column* version of the above. The messages are now block columns of G and local computation is more nearly level-3.

Additional details on ring factorization procedures may be found in Geist & M.T. Heath [1986], Ipsen, Saad & Schultz [1986], Bischof [1988], Golub & Van Loan [1989, Chapter 6], and Geist & Heath [1985]. An important related topic is the parallel solution of triangular systems. See Romine & Ortega [1988], Heath & Romine [1988], Li & Coleman [1988], and Eisenstat, Heath, Henkel & Romine [1988].

Acknowledgement

This research was partially supported by the U.S. Army Research Office through the Mathematical Sciences Institute, Cornell University.

References

Aasen, J.O. (1971). On the reduction of a symmetric matrix to tridiagonal form. *BIT* 11, 233–242.

Anderson, N., I. Karasalo (1975). On computing bounds for the least singular value of a triangular matrix. *BIT* 15, 1–4.

Andrews, G., F.B. Schneider (1983). Concepts and notations for concurrent programming. *Comput. Surv.* 15, 1–43.

Arioli, M., J.W. Demmel, I.S. Duff (1989). Solving sparse linear systems with sparse backward error, Report CSS 214, Computer Science and Systems Division, AERE Harwell, Didcot, England.

Ashby, S., T.A. Manteuffel, P.E. Saylor (1988). A taxonomy for conjugate gradient methods, Report UCRL-98508, Lawrence Livermore National Laboratory, Livermore, CA.

Axelsson, O. (1985). A survey of preconditioned iterative methods for linear systems of equations. *BIT* 25, 166–187.

Barlow, J.L., N.K. Nichols, R.J. Plemmons (1988). Iterative methods for equality constrained least squares problems. *SIAM J. Sci. Statist. Comput.* 9, 892–906.

Bartels, R.H. (1971). A stabilization of the simplex method. *Numer. Math.* 16, 414–434.

Barwell, V., J.A. George (1976). A comparison of algorithms for solving symmetric indefinite systems of linear equations. *ACM Trans. Math. Software* 2, 242–251.

Bavely, C., G.W. Stewart (1979). An algorithm for computing reducing subspaces by block diagonalization. *SIAM J. Numer. Anal.* 16, 359–367.

Bischof, C.H. (1988). A parallel QR factorization algorithm with local pivoting, Report ANL/MCS-P21-1088, Argonne National Laboratory, Argonne, IL.

Bischof, C.H., C. Van Loan (1986). Computing the SVD on a ring of array processors, in: J. Cullum, R. Willoughby (eds.), *Large Scale Eigenvalue Problems*, North-Holland, Amsterdam, pp. 51–66.

Bischof, C.H., C. Van Loan (1987). The WY representation for products of householder matrices. *SIAM J. Sci. Statist. Comput*, 8, s2–s13.

Björck, Å. (1984). A general updating algorithm for constrained linear least squares problems. *SIAM J. Sci. and Statist. Comput.* 5, 394–402.

Björck, Å. (1987). Stability analysis of the method of seminormal equations. *Linear Algebra Appl.* 88/89, 31–48.

Björck, Å. (1988). *Least Squares Methods: Handbook of Numerical Analysis, Vol. 1: Solution of Equations in R^N*, North-Holland, Amsterdam.

Björck, Å., G.H. Golub (1973). Numerical methods for computing angles between linear subspaces. *Math. Comp.* 27, 579–594.

Björck, Å., R.J. Plemmons, H. Schneider (1981). *Large-Scale Matrix Problems*, North-Holland, New York.

Bojanczyk, A.W., R.P. Brent, P. Van Dooren, F.R. de Hoog (1987). A note on downdating the Cholesky factorization. *SIAM J. Sci. Statist. Comput.* 8, 210–221.

Boyle, J., R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, R. Stevens (1987). *Portable Programs for Parallel Processors*, Holt, Rinehart & Winston, New York.

Brent, R.P., F.T. Luk, C. Van Loan (1985). Computation of the singular value decomposition using mesh connected processors. *J. VLSI Computer Systems* 1, 242–270.

Buckley, A. (1974). A note on matrices $A = 1 + H$, H skew-symmetric. *Z. Angew. Math. Mech.* 54, 125–126.

Buckley, A. (1977). On the solution of certain skew-symmetric linear systems. *SIAM J. Numer. Anal.* 14, 566–570.

Bunch, J.R. (1971). Analysis of the diagonal pivoting method. *SIAM J. Numer. Anal.* 8, 656–680.

Bunch, J.R., L. Kaufman (1977). Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.* 31, 162–179.

Bunch, J.R., L. Kaufman, B.N. Parlett (1976). Decomposition of a symmetric matrix. *Numer. Math.* 27, 95–109.

Bunch, J.R., C.P. Nielsen, D.C. Sorensen (1978). Rank-one modification of the symmetric eigenproblem. *Numer. Math.* 31, 31–48.

Bunch, J.R., B.N. Parlett (971). Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J. Numer. Anal.* 8, 639–655.

Businger, P.A. (1969). Reducing a matrix to Hessenberg form. *Math. Comp.* 23, 819–821.

Businger, P.A. (1971). Numerically stable deflation of Hessenberg and symmetric tridiagonal matrices, *BIT* 11, 262–270.

Businger, P.A., G.H. Golub (1965). Linear least squares solutions by Householder transformations. *Numer. Math.* 7, 269–276.

Chan, S.P., B.N. Parlett (1977). Algorithm 517: A program for computing the condition numbers of matrix eigenvalues without computing eigenvectors. *ACM Trans. Math. Software* 3, 186–203.

Chan, T.F. (1982). An improved algorithm for computing the singular value decomposition. *ACM Trans. Math. Software* 8, 72–83.

Chan, T.F. (1987). Rank-revealing QR factorizations. *Linear Algebra Appl.* 88/89, 67–82.

Chen, S., J. Dongarra, C. Hsuing (1984). Multiprocessing linear algebra algorithms on the Cray X-MP-2; Experiences with small granularity. *J. Parallel and Distributed Computing* 1, 22–31.

Cline, A.K., A.R. Conn, C. Van Loan (1982). Generalizing the LINPACK condition estimator, in: J.P. Hennart (ed.), *Numerical Analysis*, Lecture Notes in Mathematics, no. 909, Springer–Verlag, New York.

Cline, A.K., C.B. Moler, G.W. Stewart, J.H. Wilkinson (1979). An estimate for the condition number of a matrix. *SIAM J. Numer. Anal.* 16, 368–375.

Coleman, T., C. Van Loan (1988). *Handbook for Matrix Computations*, SIAM Philadelphia, PA.

Concus, P., G.H. Golub, G. Meurant (1985). Block preconditioning for the conjugate gradient method. *SIAM J. Sci. Statist. Comput.* 6, 220–252.

Concus, P., G.H. Golub, D.P. O'Leary (1976). A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations, in: J.R. Bunch, D.J. Rose (eds.), *Sparse Matrix Computations*, Academic Press, New York.

Cox, M.G. (1981). The least squares solution of overdetermined linear equations having band or augmented band structure. *IMA J. Numer. Anal.* 1, 3–22.

Cullum, J., R.A. Willoughby (1985a). *Lanczos Algorithms for Large Symmetric Eigenvalue Computations. Vol. I: Theory*, Birkhauser, Boston.

Cullum, J., R.A. Willoughby (1985b). *Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Vol. II: Programs*, Birkhauser, Boston.

Daniel, J., W.B. Gragg, L. Kaufman, G.W. Stewart (1976). Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization. *Math. Comp.* 30, 772–795.

Davis, C., W.M. Kahan (1970). The rotation of eigenvectors by a perturbation III. *SIAM J. Numer. Anal.* 7, 1–46.

De Boor, C., A. Pinkus (1977). A backward error analysis for totally positive linear systems. *Numer. Math.* 27, 485–490.

Demmel, J.W. (1983). A numerical analyst's Jordan canonical form, Ph.D. Thesis, Univ. of California, Berkeley.

Dongarra, J.J., J.R. Bunch, C.B. Moler, G.W. Stewart (1978). *LINPACK Users Guide*, SIAM, Philadelphia, PA.

Dongarra, J.J., J. Du Croz, I.S. Duff, S. Hammarling (1988). A set of level 3 basic linear algebra subprograms, Report ANL-MCS-TM-88, Argonne National Laboratory, Argonne, IL.

Dongarra, J.J., J. Du Croz, S. Hammarling, R.J. Hanson (1988a). An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Software* 14, 1–17.

Dongarra, J.J., J. Du Croz, S. Hammarling, R.J. Hanson (1988b). Algorithm 656: An extended set of Fortran basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Software* 14, 18–32.

Dongarra, J.J., F.G. Gustavson, A. Karp (1984). Implementation linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Rev.* 26, 91–112.

Dongarra, J., T. Hewitt (1986). Implementing dense linear algebra algorithms using multi-tasking on the Cray X-MP-4 (or approaching the gigaflop). *SIAM J. Sci. Statist. Comput.* 7, 347–350.

Dongarra, J.J., R.E. Hiromoto (1984). A collection of parallel linear equation routines for the Denelcor HEP. *Parallel Comput.* 1, 133–142.

Dongarra, J.J., L. Kaufman, S. Hammarling (1986). Squeezing the most our of eigenvalue solvers on high performance computers. *Linear Algebra Appl.* 77, 113–136.

Dongarra, J.J., A. Sameh, D. Sorensen (1986). Implementation of some concurrent algorithms for matrix factorization. *Parallel Comput.* 3, 25–34.

Dongarra, J.J., D.C. Sorensen (1986). Linear algebra on high performance computers. *Appl. Math. and Comput.* 20, 57–88.

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

Duff, I.S., J.K. Reid (1976). A comparison of some methods for the solution of sparse over-determined systems of lienar equations. *J. Inst. Math. Its Appl.* 17, 267–280.

Eisenstat, S.C., M.T. Heath, C.S. Henkel, C.H. Romine (1988). Modified cyclic algorithms for solving triangular systems on distributed memory multiprocessors. *SIAM J. Sci. Statist. Comput.* 9, 589–600.

Eldèn, L. (1980). Perturbation theory for the least squares problem with linear equality constraints. *SIAM J. Numer. Anal.* 17, 338–350.

Elman, H. (1986). A stability analysis of incomplete LU factorization. *Math. Comp.* 47, 191–218.

Erisman, A.M., J.K. Reid (1974). Monitoring the stability of the triangular factorization of a sparse matrix. *Numer. Math.* 22, 183–186.

Faber, V., T. Manteuffel (1984). Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Anal.* 21, 352–362.

Forsythe, G.E., P. Henrici (1960). The cyclic Jacobi method for computing the principal values of a complex matrix. *Trans. Amer. Math. Soc.* 94, 1–23.

Forsythe, G.E., C.B. Moler (1967). *Computer Solution of Linear Algebraic Systems*. Prentice-Hall, Englewood Cliffs, NJ.

Funderlic, R.E., A. Geist (1986). Torus data flow for parallel computation of missized matrix problems. *Linear Algebra Appl.* 7, 149–164.

Gallivan, K., W. Jalby, U. Meier (1987). The use of BLAS3 in linar algebra on a parallel processor with a hierarchical memory. *SIAM J. Sci. Statist. Comput.* 8, 1079–1084.

Gallivan, K., W. Jalby, U. Meier, A.H. Sameh (1988). Impact of hierarchical memory systems on linear algebra algorithm design. *Int. J. Supercomputer Appl.* 2, 12–48.

Gallivan, K., R.J. Plemmons, A.H. Sameh (1990). Parallel algorithms for dense linear algebra computations. *SIAM Rev.* 32, 54–135.

Garbow, B.S., J.M. Boyle, J.J. Dongarra, C.B. Moler (1972). *Matrix Eigensystem Routines: EISPACK Guide Extension*, Springer, New York.

Geist, G.A., M.T. Heath (1985). Parallel Cholesky factorization on a hypercube multiprocessor, Report ORNL 6190, Oak Ridge Laboratory, Oak Ridge, TN.

Geist, G.A., M.T. Heath (1986). Matrix factorization on a hypercube, in: M.T. Heath (ed.), *Hypercube Multiprocessor*, SIAM, Philadelphia, PA, 161–180.

Gentleman, W.M. (1973). Least squares computations by Givens transformations without square roots. *J. Inst. Math. Appl.* 12, 329–336.

Gentleman, W. M., H.T. Kung (1981). Matrix triangularization by systolic arrays. *SPIE Proceedings* 298, 19–26.

George, J.A., M.T. Heath, J. Liu (1986). Parallel Cholesky factorization on a shared memory multiprocessor. *Linear Algebra Appl.* 77, 165–187.

George, J.A., J.W. Liu (1981). *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Gill, P.E., G.H. Golub, W. Murray, M.A. Saunders (1974). Methods for modifying matrix factorizations. *Math. Comp.* 28, 505–535.

Gill, P.E., W. Murray (1976). The orthogonal factorization of a large sparse matrix, in: J.R. Bunch, D.J. Rose (eds.), *Sparse Matrix Computations*, Academic Press, New York, pp. 177–200.

Gill, P.E., W. Murray, M.A. Saunders (1975). Methods for computing and modifying the LDV factors of a matrix. *Math. Comp.* 29, 1051–1077.

Goldfarb, D. (1976). Factorized variable metric methods for unconstrained optimization. *Math. Comp.* 30, 796–811.

Golub, G.H. (1965). Numerical methods for solving linear least squares problems. *Numer. Math.* 7, 206–216.

Golub, G.H. (1969). Matrix decompositions and statistical computation, in: R.C. Milton, J.A. Nelder (eds.), *Statistical Computation*, Academic Press, New York, pp. 365–397.

Golub, G.H. (1973). Some modified matrix eigenvalue problems. *SIAM Rev.* 15, 318–344.

Golub, G.H., W. Kahan (1965). Calculating the singular values and pseudo-inverse of a matrix. *SIAM J. Num. Anal. Ser. B* 2, 205–224.

Golub, G.H., G. Meurant (1983). *Résolution Numérique des Grandes Systèmes Linéaires*, Collection de la Direction des Etudes et Recherches de l'Electricité de France, Vol. 49, Eyolles, Paris.

Golub, G.H., S. Nash, C. Van Loan (1979). A Hessenberg–Schur method for the matrix problem $AX + XB = C$. *IEEE Trans. Auto. Cont. AC*-24, 909–913.

Golub, G.H., C. Reinsch (1970). Singular value decomposition and least squares solutions. *Numer. Math.* 14, 403–420. See also Wilkinson & Reinsch [1971, pp. 134–151].

Golub, G.H., C.F. Van Loan (1979). Unsymmetric positive definite linear systems. *Linear Algebra Appl.* 28, 85–98.

Golub, G.H., C.F. Van Loan (1980). An analysis of the total least squares problem. *SIAM J. Number. Anal.* 17, 883–893.

Golub, G.H., C.F. Van Loan (1989). *Matrix Computations, 2nd edition*, Johns Hopkins Uinversity Press, Baltimore, MD.

Golub, G.H., J.H. Wilkinson (1976). Ill-conditioned eigensystems and the computation of the Jordan canonical form. *SIAM Rev.* 18, 578–619.

Grimes, R.G., J.G. Lewis (1981). Condition number estimation for sparse matrices. *SIAM J. Sci. Statist. Comput.* 2, 384–388.

Hager, W. (1984). Condition Estimates. *SIAM J. Sci. Statist. Comput.* 5, 311–316.

Hager, W. (1988). *Applied Numerical Linear Algebra*, Prentice-Hall, Englewood Cliffs, NJ.

Halmos, P. (1958). *Finite Dimensional Vector Spaces*, Van Nostrant Reinhold, New York.

Hammarling, S. (1974). A note on modifications to the Givens plane roation. *J. Inst. Math. Appl.* 13, 215–218.

Heath, M.T. (ed.) (1986). *Proceedings of First SIAM Conference on Hypercube Multiprocessors*, SIAM, Philadelphia, PA.

Heath, M.T. (ed.) (1987). *Hypercube Multiprocessors*, SIAM, Philadelphia, PA.

Heath, M.T., C.H. Romine (1988). Parallel solution of triangular systems on distributed memory multiprocessors. *SIAM J. Sci. Statist. Comput.* 9, 558–588.

Heath, M.T., D.C. Sorensen (1986). A pipelined method for computing the QR factorization of a sparse matrix. *Linear Algebra Appl.* 77, 189–203.

Heller, D. (1978). A survey of parallel algorithms in numerical linear algebra. SIAM Rev. 20, 740–777.

Heller, D.E., I.C.F. Ipsen (1983). Systolic networks for orthogonal decompositions, *SIAM J. Sci. Statist. Comput.* 4, 261–269.

Higham, N.J. (1987). A survey of condition number estimation for triangular matrices. *SIAM Rev.* 29, 575–596.

Higham, N.J. (1988). Computing a nearest symmetric positive semidefinite matrix. *Linear Algebra Appl.* 103, 103–118.

Higham, N.J. (1989). Analysis of the Cholesky decomposition of a semi-definite matrix, in: M.G. Cox, S.J. Hammarling (eds.), *Reliable Numerical Computation*, Oxford University Press, Oxford.

Higham, N.J., D.J. Higham (1989). Large growth factors in Gaussian elimination with pivoting. *SIAM J. Matrix Anal. Appl.* 10, 155–164.

Hockney, R.W., C.R. Jesshope (1989). *Parallel Computers 2*, Adam Hilger, Bristol and Philadelphia.

Ipsen, I.C.F., Y. Saad, M. Schultz (1986). Dense linear systems on a ring of processors. *Linear Algebra Appl.* 77, 205–239.

Kågström, B. (1985). The generalized singular value decomposition and the general $A - \lambda B$ problem. *BIT* 24, 568–583.

Kågström, B., P. Ling (1988). Level 2 and 3 BLAS routines for the IBM 3090 VF/400: Implementation and experiences, Report UMINF-154.88, Univesity of Umeå, Inst. of Inf. Proc., S-901 87 Umeå, Sweden.

Kågström, B., P. Ling, C. Van Loan (1991). High performance GEMM-based level-3 BLAS: Sample routines for double precision real data, Report UMINF-91.09, University of Umeå, Inst. of Inf. Proc., S-901 87 Umeå, Sweden.

Kågström, B., A. Ruhe (1980a). An algorithm for numerical computation of the Jordan normal form of a complex matrix. *ACM Trans. Math. Software* 6, 398–419.

Kågström, B., A. Ruhe (1980b). Algorithm 560 JNF: An algorithm for numerical computation of the Jordan normal form of a complex matrix. *ACM Trans. Math. Software* 6, 437–443.

Kågström, B., A. Ruhe (1983). *Matrix Pencils*, Proc. Pite Havsbad, 1982, Lecture Notes in Mathematics 973, Springer, New York and Berlin.

Kaniel, S. (1966). Estimates for some computational techniques in linear algebra. *Math. Comp.* 20, 369–378.

Kielbasinski, A. (1987). A note on rounding error analysis of Cholesky factorization. *Linear Algebra Appl.* 88/89, 487–494.

Kogbetlianztz, E.G. (1955). Solution of linear equations by diagonalization of coefficient matrix. *Quart. Appl. Math.* 13, 123–132.

Laub, A.J. (1981). Efficient multivariable frequency response computations. *IEEE Trans. Auto. Contr. AC*-26, 407–408.

Lawson, C.L., R.J. Hanson (1974). *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, NJ.

Lawson, C.L., R.J. Hanson, D.R. Kincaid, F.T. Krogh (1979a). Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Software* 5, 308–323.

Lawson, C.L., R.J. Hanson, D.R. Kincaid, F.T. Krogh (1979b). Algorithm 539: Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Software* 5, 324–325.

Leon, S. (1980). *Linear Algebra with Applications*, MacMillan, New York.

Li, G., T. Coleman (1988). A parallel triangular solver for a distributed-memory multiprocessor. *SIAM J. Sci. Statist. Comput.* 9, 485–502.

Lusk, E., R. Overbeek (1983). Implementation of monitors with macros: A programming aid for the HEP and other parallel processors, Argonne Report 83–97.

Manteuffel, T.A. (1977). The Tchebychev iteration for nonsymmetric linear systems. *Numer. Math.* 28, 307–327.

Manteuffel, T.A. (1979). Shifted incomplete Cholesky factorization, in: I.S. Duff, G.W. Stewart (eds.), *Sparse Matrix Proceedings, 1978*, SIAM, Philadelphia, PA.

Meijerink, J.A., H.A. Van der Vorst (1977). An iterative solution method for linear equations systems of which the coefficeint matrix is a symmetric M-matrix. *Math. Comp.* 31, 148–462.

Meinguet, J. (1983). Refined error analyses of Cholesky factorization. *SIAM J. Numer. Anal.* 20, 1243–1250.

Meurant, G. (1984). The block preconditioned conjugate gradient method on vector computers. *BIT*, 24, 623–633.

Moler, C.B., G.W. Stewart (1973). An algorithm for generalized matrix eigenvalue problems. *SIAM J. Numer. Anal.* 10, 241–256.

O'Leary, D.P., G.W. Stewart (1985). Data flow algorithms for parallel matrix computations. *Comm. ACM* 28, 841–853.

Ortega, J.M., R.G. Voigt (1985). Solution of partial differential equations on vector and parallel computers. *SIAM Rev.* 27, 149–240.

Paige, C.C. (1971). The computation of eigenvalues and eigenvectors of very large sparse matrices, Ph.D. thesis, London University, London, England.

Paige, C.C. (1980). Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem. *Linear Algebra Appl.* 34, 235–258.

Paige, C.C. (1986). Computing the generalized singular value decomposition. *SIAM J. Sci. Statist. Comput.* 7, 1126–1146.

Paige, C.C., M. Saunders (1981). Towards a generalized singular value decomposition. *SIAM J. Numer. Anal.* 18, 398–405.

Parlett, B.N. (1971). Analysis of algorithms for reflections in bisectors. *SIAM Rev.* 13, 197–208.

Parlett, B.N. (1980). *The Symmetric Eigenvalue Problem*, Prentice-Hall, Englewood Cliffs, NJ.

Parlett, B.N., B. Nour-Omid (1985). The use of a refined error bound when updating eigenvalues of tridiagonals. *Linear Algebra Appl.* 68, 179–220.

Parlett, B.N., W.G. Poole (1973). A geometric theory for the QR, LU, and power iterations. *SIAM J. Numer. Anal.* 10, 389–412.

Parlett, B.N., H. Simon, L.M. Stringer (1982). On estimating the largest eigenvalue with the Lanczos algorithm. *Math. Comp.* 38, 153–166.

Poole, E.L., J.M. Ortega (1987). Multicolor ICCG methods for vector computers. *SIAM J. Number. Anal.* 24, 1394–1418.

Rodrigue, G. (ed.) (1982). *Parallel Computations*, Academic Press, New York.

Rodrigue, G., D. Wolitzer (1984). Preconditions by incomplete block cyclic reduction. *Math. Comp.* 42, 549–566.

Romine, C.H., J.M. Ortega (1988). Parallel solution of triangular systems of equations. *Parallel Comput* 6, 109–114.

Ruhe, A. (1969). An algorithm for numerical determination of the structure of a general matrix. *BIT* 10, 196–216.

Saad, Y. (1980). On the rates of convergence of the Lanczos and the block Lanczos methods. *SIAM J. Numer. Anal.* 17, 687–706.

Saad, Y. (1981). Krylov subspace methods for solving large unsymmetric linear systems. *Math. Comp.* 37, 105–126.

Saad, Y. (1982). The Lanczos biorthogonalization algorithm and other oblique projection methods for solving large unsymmetric systems. *SIAM J. Numer. Anal.* 19, 485–506.

Saad, Y. (1984). Practical use of some Krylov subspace methods for solving indefinite and nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.* 5, 203–228.

Saad, Y. (1989). Krylov subspace methods on supercomputers. *SIAM J. Sci. Statist. Comput.* 10, 1200–1232.

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

Schreiber, R., B.N. Parlett (1987). Block reflectors: Theory and computation. *SIAM J. Numer. Anal.* 25, 189–205.

Schreiber, R., C. Van Loan (1989). A storage efficient WY representation for products of Householder transformations. *SIAM J. Sci. Statist. Comput.* 10, 53–57.

Scott, D.S. (1979). Block Lanczos software for symmetric eigenvalue problems, Report ORNL/CSD-48, Oak Ridge National Laboratory, Union Carbide Corporation, Oak Ridge, TN.

Seager, M.K. (1986). Parallelizing conjugate gradient for the Cray X-MP. *Parallel Comput.* 3, 35–47.

Serbin, S. (1980). On factoring a class of complex symmetric matrices without pivoting. *Math. Comp.* 35, 1231–1234.

Skeel, R.D. (1979). Scaling for numerical stability in Gaussian elimination. *J. ACM* 26, 494–526.

Skeel, R.D. (1980). Iterative refinement implies numerical stability for Gaussian elimination. *Math. Comp.* 35, 817–832.

Skeel, R.D. (1981). Effect of equilibration on residual size for partial pivoting. *SIAM J. Numer. Anal.* 18, 449–455.

Smith, B.T., J.M. Boyle, Y. Ikebe, V.C. Klema, C.B. Moler (1970). *Matrix Eigensystem Routines: EISPACK Guide, 2nd edition*, Springer, New York.

Speiser, J., C. Van Loan (1984). Signal processing computations using the generalized singular value decomposition, *Proc. SPIE Vol. 495*, SPIE Int'l Conf., San Diego, CA, 1984.

Stewart, G.W. (1971). Error bounds for approximate invariant subspaces of closed linear operators. *SIAM J. Numer. Anal.* 8, 796–808.

Stewart, G.W. (1972). On the sensitivity of the eigenvalue problem $Ax = \lambda Bx$. *SIAM J. Numer. Anal.* 9, 669–686.

Stewart, G.W. (1973). *Introduction to Matrix Computations*, Academic Press, New York.

Stewart, G.W. (1975). Methods of simultaneous iteration for calculating eigenvectors of matrices, in: J.H. Miller (ed.), *Topics in Numerical Analysis II*, Academic Press, New York, pp. 185–196.

Stewart, G.W. (1976a). The economical storage of plane rotations. *Numer. Math.* 25, 137–138.

Stewart, G.W. (1976b). Simultaneous iteration for computing invariant subspaces of non-Hermitian matrices. *Numer. Math.* 25, 12–36.

Stewart, G.W. (1977). On the perturbation of pseudo-inverses, projections, and linear least squares problems. *SIAM Rev.* 19, 634–662.

Stewart, G.W. (1979). The effects of rounding error on an algorithm for downdating a Cholesky factorization. *J. Inst. Math. Its Appl.* 23, 203–213.

Stewart, G.W. (1983). A method for computing the generalized singular value decomposition, in: B. Kågström, A. Ruhe (eds.), *Matrix Pencils*, Springer, New York, pp. 207–220.

Stewart, G.W. (1984). Rank degeneracy. *SIAM J. Sci. Statist. Comput.* 5, 403–413.

Stewart, G.W. (1987). Collinearity and least squares regression. *Statist. Sci.* 2, 68–100.

Strang, G. (1988). *Linear Algebra and Its Applications, 3rd edition*, Harcourt, Brace and Jovanovich, San Diego.

Symm, H.J., J.H. Wilkinson (1980). Realistic error bounds for a simple eigenvalue and its associated eigenvector. *Numer. Math.* 35, 113–126.

Trefethen, L.N., R.S. Schrieber (1987). Average case stability of Gaussian elimination, Numer. Anal. Report 88-3, Department of Mathematics, MIT.

Tsao, N.K. (1975). A note on implementating the Householder transformation. *SIAM J. Numer. Anal.* 12, 53–58.

Van der Sluis, A. (1969). Condition numbers and equilibration matrices. *Number. Math.* 14, 14–23.

Van der Sluis, A. (1970). Condition, equilibration, and pivoting in linear algebraic systems. *Numer. Math.* 15, 74–86.

Van der Vorst, H.A. (1982a). A vectorizable variant of some ICCG methods. *SIAM J. Sci. Statist. Comput.* 3, 350–356.

Van der Vorst, H.A. (1982b). A generalized Lanczos scheme. *Math. Comp.* 39, 559–562.

Van Dooren, P., C.C. Paige (1986). On the quadratic convergence of Kogbetliantz's algorithm for computing the singular value decomposition. *Linear Algebra Appl.* 77, 301–313.

Van Huffel, S. (1987). Analysis of the total least squares problem and its use in parameter estimation, Doctoral Thesis, Department of Electrical Engineering, K.U. Leuven.

Van Huffel, S., J. Vandewalle (1988). The partial total least squares algorithm. *J. Comp. and App. Math.* 21, 333–342.

Van Loan, C.F. (1976). Generalizing the singular value decomposition. *SIAM J. Numer. Anal.* 13, 76–83.

Van Loan, C.F. (1982a). A generalized SVD analysis of some weighting methods for equality-constrained least squares, in: B. Kagstrom, A. Ruhe (eds.), *Proceedings of the Conference on Matrix Pencils*, Springer, New York.

Van Loan, C.F. (1982b). Using the Hessenberg decomposition in control theory, in: D.C. Sorensen, R.J. Wets (eds.), *Algorithms and Theory in Filtering and Control*, Mathematical Programming Study no. 18, North-Holland, Amsterdam, pp. 102–111.

Van Loan, C.F. (1985a). Computing the CS and generalized singular value decomposition. *Numer. Math.* 46, 479–492.

Van Loan, C.F. (1985b). On the method of weighting for equality constrained least squares problems. *SIAM J. Numer. Anal.* 22, 851–864.

Van Loan, C.F. (1987). On estimating the condition of eigenvalues and eigenvectors. *Linear Algebra Appl.* 88/89, 715–732.

Varga, R.S. (1962). *Matrix Iterative Analysis*, Prentice-Hall, Englewood Cliffs, NJ.

Walker, H.F. (1988). Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Statist. Comput.* 9, 152–163.

Ward, R.C. (1975). The combination shift QZ algorithm. *SIAM J. Numer. Anal.* 12, 835–853.

Watkins, D.S. (1991). *Fundamentals of Matrix Computations*, Wiley, New York.

Wedin, P.Å. (1972). Perturbation bounds in connection with the singular value decomposition. *BIT* 12, 99–111.

Wedin, P.Å. (1973). On the almost rank-deficient case of the least squares problem. *BIT* 13, 344–354.

Wilkinson, J.H. (1961). Error analysis of direct methods of matrix inversion. *J. ACM.* 10, 281–330.

Wilkinson, J.H. (1965). *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford.

Wilkinson, J.H. (1971). Modern error analysis. *SIAM Rev.* 14, 548–568.

Wilkinson, J.H. (1972). Note on matrices with a very ill-conditioned eigenproblem. *Numer. Math.* 19, 176–178.

Wilkinson, J.H., C. Reinsch (eds.) (1971). *Handbook for Automatic Computation, Vol. 2: Linear Algebra*, Springer, New York. (Here abbrevated as *HACLA*.)