COMPACT SCHEMES

We know that if A is nonsingular, and if pivoting is not needed, then $A = LU$ is produced by Gaussian elimination. The idea by *compact schemes* is to produce these explicitly without going thru the elimination process. What is involved? Recall

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_{2,1} & 1 & 0 & & \\ m_{3,1} & m_{3,2} & 1 & & \vdots \\ \vdots & & & \ddots & \\ m_{n,1} & m_{n,2} & \cdots & m_{n,n-1} & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ 0 & u_{2,2} & u_{2,3} & & u_{2,n} \\ \vdots & 0 & \ddots & & \vdots \\ & \vdots & 0 & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & \cdots & 0 & u_{n,n} \end{bmatrix}$$

We multiply these and match the results to the corresponding elements of A.

Step 1a. Immediately,

$$u_{1,j} = a_{1,j}, \quad j = 1, ..., n$$

Step 1b. Solve for the first column of L, beginning by multiplying rows 2 thru n of L times column 1 of U. Solve

$$m_{i,1}u_{1,1} = a_{i,1}, \quad i = 2, ..., n$$

for the coefficients $m_{i,1}$.

Step 2a. Solve for the second row of U, beginning by multiplying row 2 of L times columns 2 thru n of U. This yields

$$m_{2,1}u_{1,j} + u_{2,j} = a_{2,j}, \quad j = 2, ..., n$$

which we solve for $u_{2,j}$ for $j = 2, ..., n$.
Step 2b. Solve for the second column of L, beginning by multiplying rows 3 thru n of L times column 2 of U. Solve

$$m_{i,1}u_{1,2} + m_{i,2}u_{2,2} = a_{i,2}, \quad i = 3, ..., n$$

for the coefficients $m_{i,2}$ for $i = 3, ..., n$.

We continue in this manner, obtaining all elements of L and U. Rather than giving the details, I will give the analogous results for some other cases. This algorithm for factoring $A = LU$ is called Doolittle's method; and when the algorithm requires the diagonal elements of U to equal 1, rather than those of L, it is called Crout's method. We have assumed that no pivoting is needed in the factorization process. In fact, it is possible to combine this factorization algorithm with row pivoting. In computer software for Gaussian elimination, it is generally one of the compact schemes that is usually implemented rather than our original definition of Gaussian elimination.

Once we have L and U, how do we solve $Ax = b$? Let $z = Ux$. Then solve the pair of systems

$$Lz = b, \quad Ux = z$$

The first is a lower triangular system (solved by forward substitution), and the second is the upper triangular system which we solved earlier by back substitution.

Historically, these algorithms were invented independent of Gaussian elimination, and it was realized only later that they were different aspects of the same method.

CHOLESKY'S METHOD

Let A be a symmetric, positive definite matrix. Then it can be shown that A can be factored in the form

$$A = LL^T$$

for some lower triangular matrix

$$L = \begin{bmatrix} \ell_{1,1} & 0 & 0 & \cdots & 0 \\ \ell_{2,1} & \ell_{2,2} & 0 & & \\ \ell_{3,1} & \ell_{3,2} & \ell_{3,3} & & \vdots \\ \vdots & & & \ddots & \\ \ell_{n,1} & \ell_{n,2} & \cdots & \ell_{n,n-1} & \ell n, n \end{bmatrix}$$

with the diagonal elements $\ell_{i,i} > 0$. Again, multiply LL^T and match to the corresponding elements of A.

Step 1. Row 1 times column 1 yields

$$\ell_{1,1}^2 = a_{1,1}$$

It can be proven that $a_{1,1} > 0$. Thus we can find $\ell_{1,1}$.

Step 2 . Multiply row 1 of L times columns 2 thru n of L^T do get

$$\ell_{1,1}\ell_{i,1} = a_{i,1}, \quad i = 2, ..., n$$

Solve for all $\ell_{i,1}$.

Continuing, for $i = 2, ..., n$, we have

$$\ell_{i,j} = \frac{a_{i,j} - \sum_{k=1}^{j-1} \ell_{i,k}\ell_{j,k}}{\ell_{j,j}}, \quad j = 1, ..., i-1$$

$$\ell_{i,i} = \left[a_{i,i} - \sum_{k=1}^{j-1} \ell_{i,k}^2 \right]^{\frac{1}{2}}$$

If we do an operations count, we will find that the number of arithmetic operations is approximately

$$\frac{1}{3}n^3$$

which is around one half of that needed with a general matrix A. This is due to the symmetry of A.

EXAMPLE

From the text, let

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

Then

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{\text{sqrt}(12)} & 0 \\ \frac{1}{3} & \frac{1}{\text{sqrt}(12)} & \frac{1}{\text{sqrt}(180)} \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{\text{sqrt}(12)} & \frac{1}{\text{sqrt}(12)} \\ 0 & 0 & \frac{1}{\text{sqrt}(180)} \end{bmatrix}$$

What is the advantage of using a compact method? Looking at the above formulas for $\ell_{i,j}$ and $\ell_{i,i}$, we see we are calculating inner products of vectors. Going back to Chapter 1, we discussed how to minimize errors in the calculation of inner products

$$\sum_{k=1}^{m} \alpha_k \beta_k$$

If we accumulated these in the arithmetic precision of the numbers themselves, then we had around $2m$ rounding errors. But if we performed the multiplications in extended precision, rounding only at the very end, then we had essentially only one rounding error. This is the advantage of compact methods. We can greatly reduce the number of rounding errors, and therefore we can obtain a more accurate answer. We can reduce the number of rounding errors from $O(n^3)$ to $O(n^2)$.

TRIDIAGONAL MATRICES

$$A = \begin{bmatrix} a_1 & c_1 & 0 & 0 & \cdots & 0 \\ b_2 & a_2 & c_2 & 0 & & \\ 0 & b_3 & a_3 & c_3 & & \vdots \\ & & & \ddots & & \\ \vdots & & & b_{n-1} & a_{n-1} & c_{n-1} \\ 0 & & \cdots & & b_n & a_n \end{bmatrix}$$

These occur very commonly in the numerical solution of partial differential equations, as well as in other applications (e.g. computing interpolating cubic spline functions).

We factor $A = LU$, as before. But now L and U take very simple forms. Before proceeding, we note with an example that the same may not be true of the matrix inverse.

EXAMPLE

Define an $n \times n$ tridiagonal matrix

$$A = \begin{bmatrix} -1 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & & \\ 0 & 1 & -2 & 1 & & \vdots \\ & & & \ddots & & \\ \vdots & & & 1 & -2 & 1 \\ 0 & & \cdots & & 1 & -\frac{n-1}{n} \end{bmatrix}$$

Then A^{-1} is given by

$$\left(A^{-1}\right)_{i,j} = \max\{i, j\}$$

Thus the sparse matrix A can (and usually does) have a dense inverse.

We factor $A = LU$, with

$$L = \begin{bmatrix} \alpha_1 & 0 & 0 & 0 & \cdots & 0 \\ b_2 & \alpha_2 & 0 & 0 & & \\ 0 & b_3 & \alpha_3 & 0 & & \vdots \\ & & & \ddots & & \\ \vdots & & & b_{n-1} & \alpha_{n-1} & 0 \\ 0 & & \cdots & & b_n & \alpha_n \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & \gamma_1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \gamma_2 & 0 & & \\ 0 & 0 & 1 & \gamma_3 & & \vdots \\ & & & \ddots & & \\ \vdots & & & 0 & 1 & \gamma_{n-1} \\ 0 & & \cdots & & 0 & 1 \end{bmatrix}$$

Multiply these and match coefficients with A to find $\{\alpha_i, \gamma_i\}$.

By doing a few multiplications of rows of L times columns of U, we obtain the general pattern as follows.

$$a_1 = \alpha_1, \quad \alpha_1\gamma_1 = c_1$$

$$a_i = \alpha_i + b_i\gamma_{i-1}, \quad i = 2, ..., n$$

$$\alpha_i\gamma_i = c_i, \quad i = 2, 3, ..., n-1$$

These are straightforward to solve.

$$\alpha_1 = a_1, \quad \gamma_1 = \frac{c_1}{\alpha_1}$$

$$\alpha_i = a_i - b_i\gamma_{i-1}, \quad \gamma_i = \frac{c_i}{\alpha_i}, \quad i = 2, ..., n-1$$

$$\alpha_n = a_n - b_n\gamma_{n-1}$$

To solve the linear system

$$Ax = f$$

or

$$LUx = f$$

instead solve the two triangular systems

$$Lz = f, \qquad Ux = z$$

Solving $Lz = f$:

$$z_1 = \frac{f_1}{\alpha_1}, \quad z_i = \frac{f_i - b_i z_{i-1}}{\alpha_i}, \quad i = 2, ..., n$$

Solving $Ux = z$:

$$x_n = z_n, \quad x_i = z_i - \gamma_i x_{i+1}, \quad i = n-1, ..., 1$$

OPERATIONS COUNT

Factoring $A = LU$.

Additions	$n-1$
Multiplications	$n-1$
Divisions	$n-1$

Solving $Lz = f$ and $Ux = z$:

Additions	$2n-2$
Multiplications	$2n-2$
Divisions	n

Thus the total number of arithmetic operations is approximately $3n$ to factor A; and it takes about $5n$ to solve the linear system using the factorization of A.

If we had A^{-1} at no cost, what would it cost to compute $x = A^{-1}f$?

$$x_i = \sum_{j=1}^{n} \left(A^{-1}\right)_{i,j} f_j, \quad i = 1, ..., n$$

THEOREM

Assume $\{a_i, b_i, c_i\}$ satisfy:

$$|a_1| > |c_1| > 0$$
$$|a_i| \geq |b_i| + |c_i|\,, \quad b_i, c_i \neq 0 \quad i = 2, ..., n-1$$
$$|a_n| > |b_n| > 0$$

Then A is nonsingular,

$$|\gamma_i| < 1, \quad i = 1, ..., n-1$$

$$|a_i| - |b_i| < |\alpha_i| < |a_i| + |b_i|\,, \quad i = 2, ..., n$$

ALGORITHMS

In the text on pages 520-521, two algorithms are given. The first, *Factor,* is to perform the LU factorization

$$LU = PA$$

The linear system $Ax = b$ is then solved using *Solve* and the results from *Factor*. This is typical of most linear equations packages, since it is very common to solve a linear system for many different right sides b.

A well-known package for solving linear systems is called *LINPACK*, and it is available from *NETLIB*. There are four versions of the package:
single precision real
double precision real
single precision complex
double precision complex

The first letters of the name of a routine are respectively, S, D, C, Z to indicate which version is being used. The analogues to *Factor* and *Solve* are called *DGEFA* and *DGESL*, respectively.

BLAS

The routines in *LINPACK* are written using smaller and more basic routines called *Basic Linear Algebra Subroutines*, or for short, *BLAS*. The idea is to use small routines to do the basic and time consuming aspects of the solution process; and then the programs can be made to run more efficiently by just implementing the *BLAS* efficiently.

Examples: *DAXPY*
This does the operation

$$a \cdot x + y \rightarrow y$$

with a a constant and both x and y vectors.

DDOT
Form the dot product of two vectors x and y.

DSCAL
Multiply a vector x by a scalar a.

DASUM

Form the sum of the absolute values of the components of a vector x.

IDAMAX

Find the index of the element in the vector x of largest magnitude.

DCOPY

Overwrite a vector y with a vector x.

DSWAP

Swap two vectors x and y.

These are given in *LINPACK* in Fortran. However, by machine coding these and taking advantage of the special features of a machine, the running time of a *LINPACK* program can be made much smaller in many cases.

For example, if you are using a vector pipeline machine such as a CRAY, then you should take advantage of the vector operations and vector registers to do *DAXPY* more efficiently. Most machine manufacturers ensure that the *BLAS* will run efficiently on their machines; and this leads to much faster *LINPACK* programs.

The examples given above are called "Level 1 *BLAS*", being vector-vector or vector-scalar in nature. We also have "Level 2 *BLAS*" and "Level 3 *BLAS*".

The "Level 2 *BLAS*" are matrix-vector operations; and the "Level 3 *BLAS*" are matrix-matrix operations. For example, *MMULT* multiplies a matrix A times a vector x. As an introduction to these ideas, see

T. Coleman and C. Van Loan, *Handbook for Matrix Computations*, SIAM, 1988.

LAPACK

The packages LINPACK and EISPACK (for solving eigenvalue problems) have been updated and replace by LAPACK. As with LINPACK and EISPACK, LAPACK can be obtained from *NETLIB*. LINPACK and EISPACK were written principally in the 1970's. New techniques and results were developed subsequently, which needed to be incorporated into these packages. In addition, many of the LINPACK algorithms would often not perform well on some of the supercomputer architectures of the 1980s, and newer algorithms needed to be developed.

The general purpose code for solving standard linear systems (in double precision) is named *dgesvx.f*, and it is available in the class account. To use LAPACK from our HP workstation network, use the compilation statement

```
f90 filename.f  -L/usr/pkg/lapack/LAPACK -llapack
                -L/opt/fortran90/lib -lblas
```

MATLAB FOR MATRICES

The name "MATLAB" is an acronym for "matrix laboratory". It was created to help in the teaching of linear algebra, by Cleve Moler at the University of New Mexico. As such, it was meant to manipulate matrices and vectors in an easy fashion. You should read the manual on the matrix operations in Matlab, and you can also run the MATLAB demo programs on this topic. I will put some Matlab programs for creating and working with matrices in the class account.

If A is a square matrix and if b is a column vector of the same order, then to solve $Ax = b$, simply write

$$x = A\backslash b$$

This is more efficient and safer than writing

$$x = \mathrm{inv}(A) * b$$

RECENT REFERENCES

J. Demmel, *Applied Numerical Linear Algebra*, SIAM Pub., 1997.

G. Golub & C. Van Loan, *Matrix Computations*, 3^{rd} ed., John Hopkins Press, 1996.

N. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM Pub., 1996.

L. Trefethen & D. Bau, *Numerical Linear Algebra*, SIAM Pub., 1997.

E. Anderson, et al, *LAPACK Users' Guide,* SIAM Pub., 1992