Chapter 5

Linear Algebra II, Algorithms

5.1 Introduction

As we say earlier, many algorithms of numerical linear algebra may be formulated as ways to calculate matrix factorizations. This point of view gives conceptual insight. Since computing the factorization is usually much more expensive than using it, storing the factors makes it possible, for example, to solve many systems of equations, $Ax = b$, with the same the same A but different b (and therefore different x), faster than if we had started over each time. Finally, when we seek high performance, we might take advantage of alternative ways to organize computations of the factors.

This chapter does not cover the many factorization algorithms in great detail. This material is available, for example, in the book of Golub and van Loan and many other places. My aim is to make the reader aware of what the computer does (roughly), and how long it should take. First I explain how the classical Gaussian elimination algorithm may be viewed as a matrix factorization, the LU factorization. The algorithm presented is not the practical one because it does not include "pivoting". Next, I discuss the Choleski (LL^*) decomposition, which is a natural version of LU for symmetric positive definite matrices. Understanding the details of the Choleski decomposition will be useful later when we study optimization methods and still later when we discuss sampling multivariate normal random variables with correlations. Finally, we show how to compute matrix factorizations, such as the QR decomposition, that lead to orthogonal matrices.

5.2 Gauss elimination and the LU decomposition

Gauss elimination is a simple systematic way to solve systems of linear equations. For example, suppose we have the system of equations

$$\begin{aligned} e_1 : &\qquad 2x + y + z = 4 \ , \\ e_2 : &\qquad x + 2y + z = 3 \ , \\ e_3 : &\qquad x + y + 2z = 4 \ . \end{aligned}$$

To find the values of x, y, and z, we first try to write equations that contain fewer variables, and eventually just one. We can "eliminate" x from the equation e_2 by subtracting $\frac{1}{2}$ of both sides of e_1 from e_2.

$$e_2' : \qquad x + 2y + z - \frac{1}{2}(2x + y + z) = 3 - \frac{1}{2} \cdot 4 \ ,$$

Which involves just y and z:

$$e_2' : \qquad \frac{3}{2}y + \frac{1}{2}z = 2 \ .$$

We can do the same to eliminate x from e_3, subtracting $\frac{1}{2}$ of each side of e_1 from the corresponding side of e_3:

$$e_3' : \qquad x + y + 2z - \frac{1}{2}(2x + y + z) = 4 - \frac{1}{2} \cdot 4 \, ,$$

which gives

$$e_3' : \qquad \frac{1}{2}y + \frac{3}{2}z = 2 \, .$$

We now have a pair of equations, e_2', and e_3' that involve only y and z. We can use e_2' to eliminate y from e_3; we subtract $\frac{1}{3}$ of each side of e_2' from the corresponding side of e_3' to get:

$$e_3'' : \qquad \frac{1}{2}y + \frac{3}{2}z - \frac{1}{3}(\frac{3}{2}y + \frac{1}{2}z) = 2 - \frac{1}{3} \cdot 2 \ \ ,$$

which simplifies to:

$$e_3'' : \qquad \frac{4}{3}z = \frac{5}{3} \, .$$

This completes the elimination phase. In the "back substitution" phase we successively find the values of z, y, and x. First, from e_3'' we immediately find

$$z = \frac{5}{4} \, .$$

Then we use e_2' (not e_3') to get y:

$$\frac{3}{2}y + \frac{1}{2} \cdot \frac{5}{4} = 1 \ \ \Longrightarrow \ \ y = \frac{1}{4} \, .$$

Lastly, e_1 yields x:

$$2x + \frac{1}{4} + \frac{5}{4} = 4 \ \ \Longrightarrow \ \ x = \frac{9}{4} \, .$$

The reader can (and should) check that $x = \frac{9}{4}$, $y = \frac{1}{4}$, and $z = \frac{5}{4}$ satisfies the original equations e_1, e_2, and e_3.

The above steps may be formulated in matrix terms. The equations, e_1, e_2, and e_3, may be assembled into a single equation involving a matrix and two vectors:

$$\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} \ \ .$$

The operation of eliminating x from the second equation may be carried out by multiplying this equation from the left on both sides by the *elementary* matrix

$$E_{21} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \ \ . \tag{5.1}$$

The result is

$$\begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} \quad .$$

Doing the matrix multiplication gives

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 4 \end{pmatrix} \quad .$$

Note that the middle row of the matrix contains the coefficients from e_2'. Similarly, the effect of eliminating x from e_3 comes from multiplying both sides from the left by the elementary matrix

$$E_{31} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \quad .$$

This gives

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 1 \\ 4 \end{pmatrix} \quad ,$$

which multiplies out to become

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{3}{2} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 2 \end{pmatrix} \quad .$$

This is the matrix form of three equations e_1, e_2', and e_3'. The last elimination step removes y from the last equation using

$$E_{32} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{3} & 1 \end{pmatrix} \quad .$$

This gives

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{3} & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{3}{2} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{3} & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 1 \\ 2 \end{pmatrix} \quad ,$$

which multiplies out to be

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & 0 & \frac{4}{3} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ \frac{5}{3} \end{pmatrix} \quad . \tag{5.2}$$

Because the matrix in (5.2) is upper triangular, we may solve for z, then y, then x, as before. The matrix equation (5.2) is equivalent to the system e_1, e_2', and e_3''.

We can summarize this sequence of multiplications with elementary matrices by saying that we multiplied the original matrix,

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

first by E_{21}, then by E_{31}, then by E_{32} to get the upper triangular matrix

$$U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & 0 & \frac{4}{3} \end{pmatrix} \ .$$

This may be written formally as

$$E_{32}E_{31}E_{21}A = U \ .$$

We turn this into a factorization of A by multiplying successively by the inverses of the elementary matrices:

$$A = E_{21}^{-1}E_{31}^{-1}E_{32}^{-1}U \ .$$

It is easy to check that we get the inverse of an elementary matrix, E_{jk} simply by reversing the sign of the number below the diagonal. For example,

$$E_{31}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix}$$

since

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \ .$$

Also, the product of the elementary matrices just has the nonzero subdiagonal elements of all of them in their respective positions (check this):

$$\begin{aligned} L &= E_{21}^{-1}E_{31}^{-1}E_{32}^{-1} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{3} & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & \frac{1}{3} & 1 \end{pmatrix} \end{aligned}$$

Finally, the reader should verify that we actually have $A = LU$:

$$\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & \frac{1}{3} & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & 0 & \frac{4}{3} \end{pmatrix} \ .$$

Now we know that performing Gauss elimination on the three equations e_1, e_2, and e_3 is equivalent to finding an LU factorization of A where the lower triangular factor has ones on its diagonal.

Finally, we can turn this process around and seek the elements of L and U directly from the structure of L and U. In terms of the entries of L and U, the matrix factorization becomes

$$\begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \ . \tag{5.3}$$

We may find the entries l_{jk} and u_{jk} one by one by multiplying out the product on the left and comparing to the known element on the right. For the $(1,1)$ element, we get

$$1 \cdot u_{11} = 2 \ ,$$

Which gives $u_{11} = 2$, as we had before. With this, we may calculate either l_{21} from matching the $(2,1)$ entries, or u_{12} from the $(1,2)$ entries. The former gives

$$l_{21} \cdot u_{11} = 1 \ ,$$

which, given $u_{11} = 2$, gives $l_{21} = \frac{1}{2}$. The latter gives

$$1 \cdot u_{12} = 1 \ .$$

and then $u_{12} = 1$. These calculations show that the LU factorization, if it exists, is unique (remembering to put ones on the diagonal of L). They also show that there is some freedom in the order in which we compute the l_{jk} and u_{jk}.

We may compute the LU factors of A without knowing the right hand side

$$\begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} \ .$$

If we know L and U and then learn the right hand side, we may find x, y, and z in a two stage process. First, *forward substitution* finds x', y', and z' so that

$$L \cdot \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} \ . \tag{5.4}$$

Then *back substitution* finds x, y, and z so that

$$U \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \ . \tag{5.5}$$

This vector solves the original equations because

$$A \begin{pmatrix} x \\ y \\ z \end{pmatrix} = L \cdot U \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = L \cdot \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} \quad .$$

Since L is lower triangular, we may solve (5.4) in the forward order, first find x', then y', then z'. The first equation is $1 \cdot x\prime = 4$ (because $l_{11} = 1$). The next is $l_{21}x' + 1 \cdot y' = 3$, which gives $y' = 1$, since $l_{21} = \frac{1}{2}$. Finally, $l_{13} = \frac{1}{2}$ and $l_{23} = \frac{1}{3}$ give $z' = \frac{5}{3}$.

Knowing x', then y', then z', we may find x, y, and z from (5.5) in the backward order (U being upper triangular), first z, then y, then x. First $u_{33}z = z'$, or $\frac{4}{3}z = \frac{5}{3}$ gives $z = \frac{5}{4}$. Then $u_{22}y + u_{23}z = y'$, or $\frac{3}{2}y + \frac{1}{2}\frac{5}{4} = 1$ gives $y = \frac{1}{4}$. Then $u_{11}x + u_{12}y + u_{23}z = x'$, or $2 \cdot x + 1 \cdot \frac{1}{4} + 1 \cdot \frac{5}{4} = 4$, gives $x = \frac{9}{4}$. The reader will recognize that these calculations are essentially the same as the ones at the beginning of the section, as solving a system using LU factorization is essentially the same as using elementary Gauss elimination.

The general LU algorithm for solving linear systems should be clear from this example. We have an $n \times n$ matrix A, and a column vector $b \in R^n$, and we wish to find another column vector x so that $Ax = b$. This is equivalent to n linear equations in the n unknowns $x_1, \ldots, x_n$. We first compute the LU decomposition of A, in one of several related ways. Then we solve a lower triangular system of equations $Ly = b$ using forward elimination. The intermediate vector entries, $y_1, \ldots, y_n$, are what we would have gotten had we applied Gauss elimination to the right hand side and A at the same time. Finally, we perform back substitution, finding x with $Ux = y$. Multiplying this by L and using $LU = a$ and $Ly = b$, we see that this x solves our problem.

Dense linear algebra computations (computations with dense matrices) like these can take lots of computer time for large n. We make an abstract count of the number of floating point operations involved in Gauss elimination. The elimination process goes from $k = 1$ to $k = n-1$, at each stage removing (setting to zero through elimination) the elements a_{jk} for $j > k$ by subtracting a multiple of row k from row j. Since rows k and j at this stage have each have $n-k$ nonzero entries, the work for row j is $n - k$ additions and multiplications. The number of rows eliminated at stage k is $n - 1$, so the work for stage k is about $(n - k)^2$. The total work for all the stages is about $\sum_{k=1}^{n}(n - k)^2 = \sum_{k=1}^{n} k^2 \approx \frac{1}{3}n^3$. Once we have L and U, the forward and backward substitutions take $O(n^2)$ operations each, far fewer than $\frac{1}{3}n^3$. For this reason, if we need to solve more than one problem $Ax = b$ with the same A, we should compute and save the LU factors once, then use them with the different b vectors.

We can appreciate the $\frac{1}{3}n^3$ work estimate in several ways. If the computer time to factor an $n \times n$ matrix is T, the time for a $2n \times 2n$ matrix will be roughly $8T$. Ten seconds would become a minute and twenty seconds. If a one gigahertz computer performed one addition and multiplication per cycle, that would be 10^9 adds and multiplies per second. Factoring a hypothetical $50,000 \times 50,000$ matrix requires about $\frac{1}{3}(50,000)^3 \approx 40 \cdot 10^{12}$ operations, which would take about

$4 \cdot 10^4$ seconds, which is about eleven hours. I doubt any desktop computer could do it nearly that fast.

The elimination and factorization algorithms just described may fail or be numerically unstable even when A is well conditioned. To get a stable algorithm, we need to introduce "pivoting". In the present context[1] this means adaptively reordering the equations or the unknowns so that the elements of L do not grow. Details are in the references.

5.3 Choleski factorization

Many applications call for solving linear systems of equations with a symmetric and positive definite A. An $n \times n$ matrix is *positive definite* if $x^* A x > 0$ whenever $x \neq 0$. Symmetric positive definite (SPD) matrices arise in many applications. If B is an $m \times n$ matrix with $m \geq n$ and $\text{rank}(B) = n$, then the product $A = B^* B$ is SPD. This is what happens when we solve a linear least squares problem using the normal equations, see Section 4.2.8). If $f(x)$ is a scalar function of $x \in R^n$, the *Hessian* matrix of second partials has entries $h_{jk}(x) = \partial^2 f(x)/\partial x_j \partial x_k$. This is symmetric because $\partial^2 f/\partial x_j \partial x_k = \partial^2 f/\partial x_k \partial x_j$. The minimum of f probably is taken at an x_* with $H(x_*)$ positive definite, see Chapter 6. Solving elliptic and parabolic partial differential equations often leads to large sparse SPD linear systems. The variance/covariance matrix of a multivariate random variable is symmetric, and positive definite except in degenerate cases.

We will see that A is SPD if and only if A has an LU factorization with $U = L^*$, i.e. $A = LL^*$ for a lower triangular matrix, L. This is the *Choleski factorization*, or *Choleski decomposition* of A. As with the LU factorization, we can find the entries of L from the equations for the entries of $LL^* = A$ one at a time, in a certain order. We write it out:

$$\begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & \vdots \\ l_{31} & l_{32} & l_{33} & \ddots & \\ \vdots & \vdots & & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & & l_{nn} \end{pmatrix} \cdot \begin{pmatrix} l_{11} & l_{21} & l_{31} & \cdots & l_{n1} \\ 0 & l_{22} & l_{32} & \cdots & l_{n2} \\ 0 & 0 & l_{33} & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & \cdots & & l_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} & a_{21} & a_{31} & \cdots & a_{n1} \\ a_{21} & a_{22} & a_{32} & \cdots & a_{n2} \\ a_{31} & a_{32} & a_{33} & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \\ a_{n1} & a_{n2} & \cdots & & a_{nn} \end{pmatrix} .$$

Notice that we have written, for example, a_{32} for the $(2,3)$ entry because A is symmetric. We start with the top left corner. Doing the matrix multiplication

[1] The term "pivot" means something different, for example, in linear programming.

gives

$$l_{11}^2 = a_{11} \implies l_{11} = \sqrt{a_{11}}\ .$$

The square root is real because $a_{11} > 0$ because A is positive definite and[2] $a_{11} = e_1^* A e_1$. Next we match the $(2,1)$ entry of A. The matrix multiplication gives:

$$l_{21} l_{11} = a_{21} \implies l_{21} = \frac{1}{l_{11}} a_{21}\ .$$

The denominator is not zero because $l_{11} > 0$ because $a_{11} > 0$. We could continue in this way, to get the whole first column of L. Alternatively, we could match $(2,2)$ entries to get l_{22}:

$$l_{21}^2 + l_{22}^2 = a_{22} \implies l_{22} = \sqrt{a_{22} - l_{21}^2}\ .$$

It is possible to show (see below) that if the square root on the right is not real, then A was not positive definite. Given l_{22}, we can now compute the rest of the second column of L. For example, matching $(3,2)$ entries gives:

$$l_{31} \cdot l_{21} + l_{32} \cdot l_{22} = a_{32} \implies l_{32} = \frac{1}{l_{22}} \left(a_{32} - l_{31} \cdot l_{21} \right)\ .$$

Continuing in this way, we can find all the entries of L. It is clear that if L exists and if we always use the positive square root, then all the entries of L are uniquely determined.

A slightly different discussion of the Choleski decomposition process makes it clear that the Choleski factorization exists whenever A is positive definite. The algorithm above assumed the existence of a factorization and showed that the entries of L are uniquely determined by $LL^* = A$. Once we know the factorization exists, we know the equations are solvable, in particular, that we never try to take the square root of a negative number. This discussion represents L as a product of elementary lower triangular matrices, a point of view that will be useful in constructing the QR decomposition (Section 5.4).

Suppose we want to apply Gauss elimination to A and find an elementary matrix of the type (5.1) to set $a_{21} = a_{12}$ to zero. The matrix would be

$$E_{21} = \begin{pmatrix} 1 & 0 & \cdots & \\ \frac{-a_{12}}{a_{11}} & 1 & 0 & \cdots \\ 0 & 0 & \ddots & \\ \vdots & & & \end{pmatrix}\ .$$

Multiplying out gives:

$$E_{21} A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots \\ 0 & a'_{22} & a'_{23} & \cdots \\ a_{13} & a_{23} & a_{33} & \\ \vdots & & & \end{pmatrix}\ .$$

[2]Here e_1 is the vector with one as its first component and all the rest zero. Similarly $a_{kk} = e_k^* A e_k$.

Only the entries in row two have changed, with the new values indicated by primes. Note that $E_{21}A$ has lost the symmetry of A. We can restore this symmetry by multiplying from the right by E_{21}^* This has the effect of subtracting $\frac{a_{12}}{a_{11}}$ times the first column of $E_{21}A$ from the second column. Since the top row of A has not changed, this has the effect of setting the $(1,2)$ entry to zero:

$$E_{21}AE_{21}^* = \begin{pmatrix} a_{11} & 0 & a_{13} & \cdots \\ 0 & a'_{22} & a'_{23} & \cdots \\ a_{13} & a'_{23} & a_{33} & \\ \vdots & & & \end{pmatrix} .$$

Continuing in this way, elementary matrices E_{31}, etc. will set to zero all the elements in the first row and top column except a_{11}. Finally, let D_1 be the diagonal matrix which is equal to the identity except that $d_{11} = 1/\sqrt{a_{11}}$. All in all, this gives ($D_1^* = D_1$):

$$D_1E_{n1}\cdots E_{31}E_{21}AE_{21}^*E_{31}^*\cdots E_{n1}^*D_1^* = \begin{pmatrix} 1 & 0 & 0 & \cdots \\ 0 & a'_{22} & a'_{23} & \cdots \\ 0 & a'_{23} & a'_{33} & \\ \vdots & & & \end{pmatrix} . \qquad (5.6)$$

We define L_1 to be the lower triangular matrix

$$L_1 = D_1E_{n1}\cdots E_{31}E_{21} ,$$

so the right side of (5.6) is $A_1 = L_1AL_1^*$ (check this). It is nonsingular since D_1 and each of the elementary matrices are nonsingular. To see that A_1 is positive definite, simply define $y = L^*x$, and note that $y \neq 0$ if $x \neq 0$ (L_1 being nonsingular), so $x^*A_1x = x^*LAL^*x = y^*Ay > 0$ since A is positive definite. In particular, this implies that $a'_{22} > 0$ and we may find an L_2 that sets a'_{22} to one and all the a_{2k} to zero.

Eventually, this gives $L_{n-1}\cdots L_1AL_1^*\cdots L_{n-1}^* = I$. Solving for A by reversing the order of the operations leads to the desired factorization:

$$A = L_1^{-1}\cdots L_{n-1}^{-1}L_{n-1}^{-*}\cdots L_1^{-*} ,$$

where we use the common convention of writing M^{-*} for the inverse of the transpose of B, which is the same as the transpose of the inverse. Clearly, L is given by $L = L_1^{-1}\cdots L_{n-1}^{-1}$.

All this may seem too abstract, but as with Gauss elimination from Section 5.2, the products of inverses of elementary matrices are easy to figure out explicitly.

Once we have the Choleski decomposition of A, we can solve systems of equations $Ax = b$ using forward and back substitution, as we did for the LU factorization.

5.4 Orthogonal matrices, least squares, and the QR factorization

Many problems in linear algebra call for linear transformations that do not change the l^2 norm:

$$\|Qx\|_{l^2} = \|x\|_{l^2} \qquad \text{for all } x \in R^n. \tag{5.7}$$

A real matrix satisfying (5.7) is orthogonal, because[3]

$$\|Qx\|_{l^2}^2 = (Qx)^* Qx = x^* Q^* Qx = x^* x = \|x\|_{l^2}^2 \ .$$

(Recall that $Q^*Q = I$ is the definition of orthogonality for square matrices.)

There is a version of Gauss elimination for solving linear least squares problems that uses orthogonal matrices Q_{jk} rather than elementary lower triangular matrices E_{jk} to make A upper triangular. A related problem is constructing an orthonormal basis for a subspace. If $v_1, \ldots, v_m$ span $V \subseteq R^n$, we desire an orthonormal basis for V or an orthonormal basis for the orthogonal complement of V. Finally, reductions using the Q_{jk} are a first step in many algorithms for finding eigenvalues and eigenvectors of symmetric matrices.

The elementary matrix E_{21} operates only on the top two rows of a matrix and modifies only the second row. The *Givens rotation*, $Q_{21}(\theta)$, is an orthogonal matrix that modifies both rows one and two. As with E_{21}, the parameter may be chosen so that $Q_{21}A$ has a zero in the $(2,1)$ position. Suppose we have a 2×2 matrix $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$, and we want an orthogonal matrix $Q = \begin{pmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{pmatrix}$ so that $QA = A' = \begin{pmatrix} a'_{11} & a'_{12} \\ a'_{21} & a'_{22} \end{pmatrix}$ has $a'_{21} = 0$. An orthogonal matrix preserves angles between vectors,[4] so an orthogonal matrix operating on the plane (R^2) must be a simple rotation possibly followed by a reflection about some line (reflect on this). The 2×2 matrix that represents rotation through angle θ is $Q = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$, so $a'_{21} = -\sin(\theta)a_{11} + \cos(\theta)a_{21}$. We achieve $a'_{21} = 0$ with the choice

$$\begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} = \frac{1}{\sqrt{a_{11}^2 + a_{21}^2}} \begin{pmatrix} a_{11} \\ a_{21} \end{pmatrix} .$$

A Givens rotation, Q_{jk}, in dimension $n > 2$ acts only on rows j and k of A.

[3] This shows an orthogonal matrix satisfies (5.7). Exercise 6 shows that a matrix satisfying (5.7) must be orthogonal.

[4] The angle between vectors x and y is given by $x^*y = \|x\| \cdot \|y\| \cdot \cos(\theta)$. This shows that if Q preserves lengths of vectors and inner products, it also preserves angles.

For example

$$Q_{42} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & \\ 0 & \cos(\theta) & 0 & \sin(\theta) & 0 & \cdots \\ 0 & 0 & 1 & 0 & \cdots & \\ 0 & -\sin(\theta) & 0 & \cos(\theta) & 0 & \\ \vdots & 0 & 0 & 0 & 1 & \ddots \\ & & \vdots & & \ddots & \ddots \end{pmatrix} .$$

This is the identity matrix except for the $\cos(\theta)$ and $\sin(\theta)$ entries shown. The matrix $A' = Q_{42}A$ is the same as A except in rows 2 and 4. We could choose θ to set $a'_{42} = 0$, as above.

The Givens' elimination strategy for the least squares problem (4.21) is an orthogonal matrix version of Gauss elimination. It is based on the observation that orthogonal Q does not change the l^2 norm of the residual: $\|Qr\|_{l^2} = \|r\|_{l^2}$. That means that solving (4.21) is equivalent to solving

$$\min_x \|QAx - Qb\|_{l^2} \ .$$

If we choose $Q = Q_{jk}$ with $j > k$ to eliminate (set to zero) all the entries a_{jk} below the diagonal ($j < k$). Of course, we must apply the Q_{jk} in the same order to b. In the end, we have an equivalent problem

$$\min_x \|Rx - b'\|_{l^2} \ . \tag{5.8}$$

where R is upper triangular.[5]

$$\left(\begin{array}{cccc} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & \vdots \\ \vdots & \ddots & \ddots & \\ 0 & \cdots & 0 & r_{nn} \\ \hline 0 & \cdots & & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & & 0 \end{array}\right) \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \left(\begin{array}{c} b'_1 \\ b'_2 \\ \vdots \\ b'_n \\ \hline b'_{n+1} \\ \vdots \\ b'_m \end{array}\right)$$

5.5 Software: Performance and cache management

In scientific computing, *performance* refers to the time it takes to run the program that does the computation. Faster computers give more performance, but so do better programs. To write high performance software, we should know

[5] The upper triangular part of the LU decomposition is called U while the upper triangluar part of the QR decomposition is called R.

what happens inside the computer, something about the compiler and about the hardware. This and several later Software sections explore performance-related issues.

Memory hierarchy is one aspect of computing hardware that influences performance. How long it takes to execute an instruction such as `a = b * c;` depends on where `a`, `b`, and `c` are stored. The time needed to *fetch* the *operands*, `b` and `c`, and to *store* the result, `a`, can be far greater than the time needed to do the operation $a = b * c$. This delay depends on where in the memory hierarchy the variables are stored when the instruction is executed.

A computer processor such as the Pentium 4 (P4) operates in time slots called *cycles*. We will give numbers for a specific but typical P4 that we call Paul. The speed of a processor is given in Hertz (cycles per second). Paul runs at 2 GHz, or 2×10^9 cycles per second. At this rate, each cycle takes $.5 \times 10^{-9}$ seconds, which is half a nanosecond. If the *pipeline* is full (see Section **??**), the processor can perform one floating point operation per cycle, if it can get the data.

The structure and performance of the memory hierarchy differs from processor to processor. On the top of the hierarchy are a small number of *registers* (about 20 for Paul). Next is *level one cache*. For Paul, transfer between a register and level one cache takes one or two cycles. Paul has two separate level one caches of 32 KBytes (32×2^{10} bytes) each, one for instructions (see Section **??**), and the other for data, such as the values of `a`, `b`, and `c`. Next is *level two* cache with a 3 or 4 cycle delay, which has 2 MBytes in Paul's case. Below the cache is main memory. One high end office desktop in 2006 had main memory with a 10 nanosecond *latency* and 6.4 GByte/sec *bandwidth*.[6] Latency is the time delay between asking for a set of numbers (see below) and the arrival of the first one. For Paul, 10 nanoseconds is about 20 cycles. Once data starts flowing, consecutive bytes are delivered at the rate of 6.4×10^9 bytes/sec, or about one double precision word per nanosecond.

Logically, computer memory is a long list of bytes. For each integer *address*, k, in the *address space* $0 \leq k < MS$ (*MS* for memory size), there is one byte of information. In a *fetch* operation, the processor specifies an address and the memory provides the contents of that address from somewhere in the hierarchy. In a *store* operation, the processor specifies the contents and the address the memory should write it into. Except for performance, the caches do not change the way the processor views data.

A cache consists of a set of cache *lines*, each one consisting of a short sequence of consecutive memory bytes. Paul's cache lines hold 128 bytes, or 16 double precision floating point words. Each of his level one caches has 256 such lines. The address space is divided into neighboring disjoint lines, any one of which may occupy a level one or level two cache line. At any given moment, some of these lines are in level one or level two caches.

[6] The *band* in *bandwidth* is the range of frequencies, the *frequency band* that can be realized on a given connection. A book on signal processing will show that the number of bits that a connection can transmit per second is proportional to the range of frequencies it can support, its *bandwidth*.

When the processor requests a byte that is not in a register, the memory will get it from the nearest place, level one cache, level two cache, or main memory.[7] For example, if it asks for byte 129 and the line of bytes from 128 to 255 is in cache, the memory will supply the cached value with the delay for the cache it was in. If byte 129 is not in cache, a *cache miss*, the whole cache line of bytes 128 to 255 will be copied from main memory to cache. This takes about 60 cycles for Paul: A 10 nsec latency and 128 byte/(6.4 GByte/sec) = 20 nsec transfer time give 30 nsec, or 60 cycles. Whatever data was in that cache line would be *flushed*, lost from cache but saved in main memory.

Many details of *cahce management* are constantly changing as hardware improves and are beyond the programmer's control. For example, Paul may have to decide which cache line to flush when there is a cache miss. If he stores a byte that is in cache (changing its value), does he *write back* that value to main memory at the same time or only when the line is flushed? How does he decide which lines to put in level one and level two? Processor manufacturers (Intel, AMD, etc.) make available detailed descriptions of the cache strategies for each of their processors.

Variable reuse and *data locality* are two factors under the programmer's control that effect cache and ultimately software performance. If the variable `b` needs to be used many times, the programmer can avoid cache misses by making these consecutive. If a program uses byte k, it is likely that fetching byte $k+1$ immediately after will not cause a cache miss because k and $k+1$ are likely to be in the same cache line.

To see this concretely, consider the product of $n \times n$ matrices $A = BC$ using the vanilla formula $a_{jk} = \sum_{l=1}^{n} b_{jl} c_{lk}$. If n is known at compile time,[8] the code could be

```
double a[N][N], b[N][N], c[N][N];
   .
   .   (define B and C)
   .
for ( j = 0; j < N; j++ ) { // j from 0 to N-1 in C.
   for ( k = 0; k < N; k++ ) {
      sum = 0;
      for ( l = 0; l < N; l++ )   // The inner loop.
         sum += b[j][l]*c[l][k]; //    one line l loop
      a[j][k] = sum;
     }  // end k loop
  } // end of j loop
```

Let us examine the pattern of memory references in this code. In C/C++ (and not in FORTRAN) the second index of the array "moves fastest". In memory, the entries of A are stored in the order `a[0][0], a[0][1], ...  ,`

[7] We also could consider the hard drive as a level in the memory hierarchy below main memory. Transfer between main memory and the hard drive is very slow on the nanosecond time scale.

[8] It is a tragedy that C/C++ is so clumsy in handling arrays of more than one index.

`a[0][N-1],a[1][0], a[1][1], ....` The *inner loop*, `for ( l=0; l<N; l++) ..`, references consecutive entries of b, which is good because each cache miss loads 16 consecutive elements of b, so only one in 16 references to b leads to a cache miss. References to c are offset by a *stride* of N. If $N > 15$, each c reference in the inner loop references a different cache line and therefore probably causes a cache miss.

A simple solution, if memory is not a problem, would be to compute the transpose of `c` before the matrix multiplication:

```
double ct[N][N];
   .
   .   (as before)
   .
for ( j = 0; j < N; j++ )     // Compute the transpose of c.
  for ( k = 0; k < N; k++ )
     ct[j][k] = c[k][j];
for ( j = 0; j < N; j++ ) { // Martix multiply loops as before
            .
            .   (as before)
            .
          sum += b[j][l]*ct[k][l];  // Use ct for data locality.
            .
            .   (as before)
            .
```

The new inner loop accesses both arrays with a unit stride to improve data locality. For large n, the loop that computes C^* uses $2n^2$ memory references, one fetch and one store for each (j, k), while the loop that computes $A = BC$ uses n^3 references to C. Thus, the time computing C^* should be tiny compared to the time saved in computing the matrix product.

5.6 References and resources

The algorithms of numerical linear algebra for dense matrices are described in great detail in the book by Charles vanLoan and Gene Golub and in the book by James Demmel. The book ??? describes computational methods for matrices stored in sparse matrix format. Still larger problems are solved by *iterative methods*. Generally speaking, iterative methods are not very effective unless the user can concoct a good *preconditioner*, which is an approximation to the inverse. Effective preconditioners usually depend in physical understanding of the problem and are problem specific.

Despite its importance for scientific computing, there is a small literature on high performance computing. The best book available in 2006 seems to be *High Performance Computing* by Kevin Down and Charles Severance. Truly high performance computing is done on computers with more than one processor,

which is called *parallel computing*. There are many specialized algorithms and programming techniques for parallel computing.

The LAPack software package is designed to make the most of memory hierarchies. In 2006, it is the best way to do high performance computational linear algebra, at least on dense matrices. It comes with a package called *Atlas* that chooses good parameters (block sizes, etc.) for LAPack depending on the cache performance of your particular processor. The LAPack manual is published by SIAM, the *Society for Industrial and Applied Mathematics*.

5.7 Exercises

1. Write a program to compute the LL^* decomposition of an SPD matrix A. Your procedure should have as arguments the dimension, n, and the matrix A. The output should be the Choleski factor, L. Your procedure must detect and report a matrix that is not positive definite and should not perform the operation `sqrtc` if $c < 0$. Write another procedure that has n and L as arguments and returns the product LL^*. Hand in: (*i*) printouts of the two procedures and the driving program, (*ii*) a printout of results showing that the testing routine reports failure when $LL^* \neq A$, (*iii*) a printout showing that the Choleski factoring procedure reports failure when A is not positive definite, (*iv*) a printout showing that the Choleski factoring procedure works correctly when applied to a SPD matrix, proven by checking that $LL^* = A$.

2. A square matrix A has *bandwidth* $2k + 1$ if $a_{jk} = 0$ whenever $|j - k| > k$. A *subdiagonal* or *superdiagonal* is a set of matrix elements on one side of the main diagonal (below for sub, above for super) with $j-k$, the distance to the diagonal, fixed. The bandwidth is the number of nonzero bands. A bandwidth 3 matrix is *tridiagonal*, bandwidth 5 makes *pentadiagonal*, etc.

   (a) Show that a SPD matrix with bandwidth $2k+1$ has a Choleski factor with nonzeros only on the diagonal and up to k bands below.

   (b) Show that the Choleski decomposition algorithm computes this L in work proportional to $k^2 n$ (if we skip operations on entries of A outside its nonzero bands).

   (c) Write a procedure for Choleski factorization of tridiagonal SPD matrices, apply it to the matrix of Exercise 11, compare the running time with this dense matrix factorizer and the one from Exercise 1. Of course, check that the answer is the same, up to roundoff.

3. Suppose $v_1, \ldots, v_m$ is an orthonormal basis for a vector space $V \subseteq R^n$. Let L be a linear transofrmation from V to V. Let A be the matrix that represents L in this basis. Show that the entries of A are given by

$$a_{jk} = v_j^* L v_k \ . \tag{5.9}$$

Hint: Show that if $y \in V$, the representation of y is this basis is $y = \sum_j y_j v_j$, where $y_j = v_j^* y$. In physics and theoretical chemistry, inner products of the form (5.9) are called *matrix elements*. For example, the eigenvalue perturbation formula (4.40) (in physicist terminology) simply says that the perturbation in an eigenvalue is (nearly) equal to to the appropriate matrix element of the perturbation in the matrix.

4. Suppose A is an $n \times n$ symmetric matrix and $V \subset R^n$ is an *invariant subspace* for A (i.e. $Ax \in V$ if $x \in V$). Show that A defines a linear transformation from V to V. Show that there is a basis for V in which this linear transformation (called A *restricted to* V) is represented be a symmmetric matrix. Hint: construct an orthonormal basis for V.

5. If Q is an $n \times n$ matrix, and $(Qx)^* Qy = x^* y$ for all x and y, show that Q is an orthogonal matrix. Hint: If $(Qx)^* Qy = x^*(Q^* Q)y = x^* y$, we can explore the entries of $Q^* Q$ by choosing particular vectors x and y.

6. If $\|Qx\|_{l^2} = \|x\|_{l^2}$ for all x, show that $(Qx)^* Qy = x^* y$ for all x and y. Hint (*polarization*): If $\|Q(x + sy)\|_{l^2}^2 = \|x + sy\|_{l^2}^2$ for all s, then $(Qx)^* Qy = x^* y$.