Chapter 7

The QR and Cholesky Factorizations

§7.1 Least Squares Fitting

§7.2 The QR Factorization

§7.3 The Cholesky Factorization

§7.4 High-Performance Cholesky

The solution of overdetermined systems of linear equations is central to computational science. If there are more equations than unknowns in $Ax = b$, then we must lower our aim and be content to make Ax close to b. Least squares fitting results when the 2-norm of $Ax - b$ is used to quantify success. In §7.1 we introduce the least squares problem and solve a simple fitting problem using built-in MATLAB features.

In §7.2 we present the QR factorization and show how it can be used to solve the least squares problem. Orthogonal rotation matrices are at the heart of the method and represent a new class of transformations that can be used to introduce zeros into a matrix.

The solution of systems of linear equations with symmetric positive definite coefficient matrices is discussed in §7.3 and a special "symmetric version" of the LU factorization called the Cholesky factorization is introduced. Several different implementations are presented that stress the importance of being able to think at the matrix-vector level. In the last section we look at two Cholesky implementations that have appeal in advanced computing environments.

7.1 Least Squares Fitting

It is not surprising that a square nonsingular system $Ax = b$ has a unique solution, since there are the same number of unknowns as equations. On the other hand, if we have more equations than unknowns, then it may not be possible to satisfy $Ax = b$. Consider the 3-by-2 case:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

For this *overdetermined* $Ax = b$ problem to have a solution, it is necessary for b to be in the span of A's two columns. This is not a forgone conclusion since this span is a proper subspace of $\mathbb{R}^3$. For example, if we try to find x_1 and x_2 so that

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix},$$

then the first equation says that $x_1 = -2x_2$. Substituting this into the second equation implies that $1 = 3x_1 + 4x_2 = 3(-2x_2) + 4x_2 = -2x_2$, while substituting it into equation 3 says that $1 = 5x_1 + 6x_2 = 5(-2x_2) + 6x_2 = -4x_2$. Since the requirements $x_2 = -1/2$ and $x_2 = -1/4$ conflict, the system has no solution.

So with more equations than unknowns, we need to adjust our aims. Instead of trying to "reach" b with Ax, we try to get as close as possible. Vector norms can be used to quantify the degree of success. If we work with the 2-norm, then we obtain this formulation:

> Given $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$, find $x \in \mathbb{R}^n$ to minimize $\| Ax - b \|_2$.

This is the *least squares* (LS) problem, apt terminology because the 2-norm involves a sum of squares:

$$\| Ax - b \|_2 = \sqrt{\sum_{i=1}^{m} \left(A(i,:)x - b(i)\right)^2}.$$

The goal is to minimize the discrepancies in each equation:

$$\left(A(i,:)x - b(i)\right)^2 = \left(a_{i1}x_1 + \cdots + a_{in}x_n - b_i\right)^2.$$

From the column point of view, the goal is to find a linear combination of A's columns that gets as close as possible to b in the 2-norm sense.

7.1.1 Setting Up Least Squares Problems

LS fitting problems often arise when a scientist attempts to fit a model to experimentally obtained data. Suppose a biologist conjectures that plant height h is a function of four soil nutrient concentrations a_1, a_2, a_3, and a_4:

$$h = a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4.$$

This is a *linear model*, and x_1, x_2, x_3, and x_4 are *model parameters* whose value must be determined. To that end, the biologist performs m (a large number) experiments. The ith experiment consists of establishing the four nutrient values a_{i1}, a_{i2}, a_{i3} and a_{i4} in the soil, planting the seed, and observing the resulting height h_i. If the model is perfect, then for $i = 1{:}m$ we have

$$h_i = a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + a_{i4}x_4.$$

That is,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ \vdots \\ h_m \end{bmatrix}.$$

Of course, the model will not be perfect, making it impossible to find such an x. The aims of the biologist are lowered and the minimizer of $\| Ax - h \|_2$ is sought. If the minimum sum of squares is small, then the biologist has reason to be happy with the chosen model. Otherwise, additional factors may be brought into play (e.g., a fifth nutrient or the amount of sunlight). The linear model may be exchanged for a nonlinear one with twice the number of parameters: $h = c_1 e^{-x_1 a_1} + \cdots + c_4 e^{-x_1 a_4}$. The treatment of such problems is briefly discussed in the next chapter.

LS fitting also arises in the approximation of known functions. Suppose the designer of a built-in square root function needs to develop an approximation to the function $f(x) = \sqrt{x}$ on the interval $[.25, 1]$. A linear approximation of the form $\ell(x) = \alpha + \beta x$ is sought. [Think of $\ell(x)$ as a two-parameter model with parameters α and β.] We could set this function to be just the linear interpolant of f at two well-chosen points. Alternatively, if a partition

$$.25 = x_1 < \cdots < x_m = 1$$

is given, then the parameters α and β can be chosen so that the quantity

$$\phi_m(\alpha, \beta) = \sum_{i=1}^{m} [(\alpha + \beta x_i) - \sqrt{x_i}]^2$$

is minimized. Note that if we set $f_i = \sqrt{x_i}$, then in the language of matrices, vectors, and norms we have

$$\phi_m(\alpha, \beta) = \left\| \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} - \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix} \right\|_2^2 .$$

Thus, an m-by-2 least squares problem needs to be solved in order to resolve α and β.

It is important to recognize that any norm could be used to quantify the error in the fit of a model. However, the 2-norm is particularly important for two reasons: (1) In many experimental settings the fitting errors are normally distributed. The underlying statistics can then be used to make a rigorous case for 2-norm minimization. (2) The mathematics of LS fitting is rich and supportive of interesting and powerful algorithms.

7.1.2 MATLAB's Least Squares Tools

The backslash operator can be used to solve the LS problem in MATLAB once it is cast in the matrix/vector terms, i.e., min $\| Ax - b \|_2$. Here is a script that solves the square root fitting problem mentioned above:

```
% Script File: ShowLSFit
% Displays two LS fits to the function f(x) = sqrt(x) on [.25,1]
close all
z = linspace(.25,1);
fz = sqrt(z);
for m = [2 100 ]
   x = linspace(.25,1,m)';
   A = [ones(m,1) x];
   b = sqrt(x);
   xLS = A\b;
   alpha = xLS(1);
   beta  = xLS(2);
   figure
   plot(z,fz,z,alpha+beta*z,'--')
   title(sprintf('m = %2.0f,  alpha = %10.6f,  beta = %10.6f',m,alpha,beta))
end
```

The two fits are displayed in Figure 7.1 on page 244. Note that if $m = 2$, then we just obtain the interpolant to the square root function at $x = .25$ and $x = 1.00$. For large m it follows from the rectangle rule that

$$\frac{.75}{m}\sum_{i=1}^{m}[(\alpha+\beta x_i)-\sqrt{x_i}]^2 \;\approx\; \int_{.25}^{1}[(\alpha+\beta x)-\sqrt{x}]^2dx \;\equiv\; \phi_\infty(\alpha,\beta)$$

where $x_i = .25 + .75(i-1)/(m-1)$ for $i = 1{:}m$. Thus, as $m \to \infty$ the minimizer of $\phi_m(\alpha,\beta)$ converges to the minimizer of $\phi_\infty(\alpha,\beta)$. From the equations

$$\frac{\partial\phi_\infty}{\partial\alpha} = 0 \qquad \text{and} \qquad \frac{\partial\phi_\infty}{\partial\beta} = 0$$

we are led to a 2-by-2 linear system that specifies the α_* and β_* that minimize $\phi_\infty(\alpha,\beta)$:

$$\begin{bmatrix} 3/4 & 15/32 \\ 15/32 & 21/64 \end{bmatrix}\begin{bmatrix} \alpha_* \\ \beta_* \end{bmatrix} = \begin{bmatrix} 7/12 \\ 31/80 \end{bmatrix}.$$

The solution is given by

$$\begin{bmatrix} \alpha_* \\ \beta_* \end{bmatrix} = \begin{bmatrix} 0.370370370 \\ 0.651851851 \end{bmatrix}.$$

In general, if we try to fit data points $(x_1, f_1), \ldots, (x_m, f_m)$ in the least squares sense with a polynomial of degree d, then an m-by-$(d+1)$ least squares problem arises. The MATLAB function `polyfit` can be used to solve this problem and `polyval` can be used to evaluate the approximant. Suppose the m-vectors `x` and `y` house the data and `d` is the required degree of the fitting polynomial. It follows that the script

```
c = polyfit(x,y,d);
xvals = linspace(min(x),max(x));
yvals = polyval(c,xvals);
plot(xvals,yvals,x,y,'o')
```

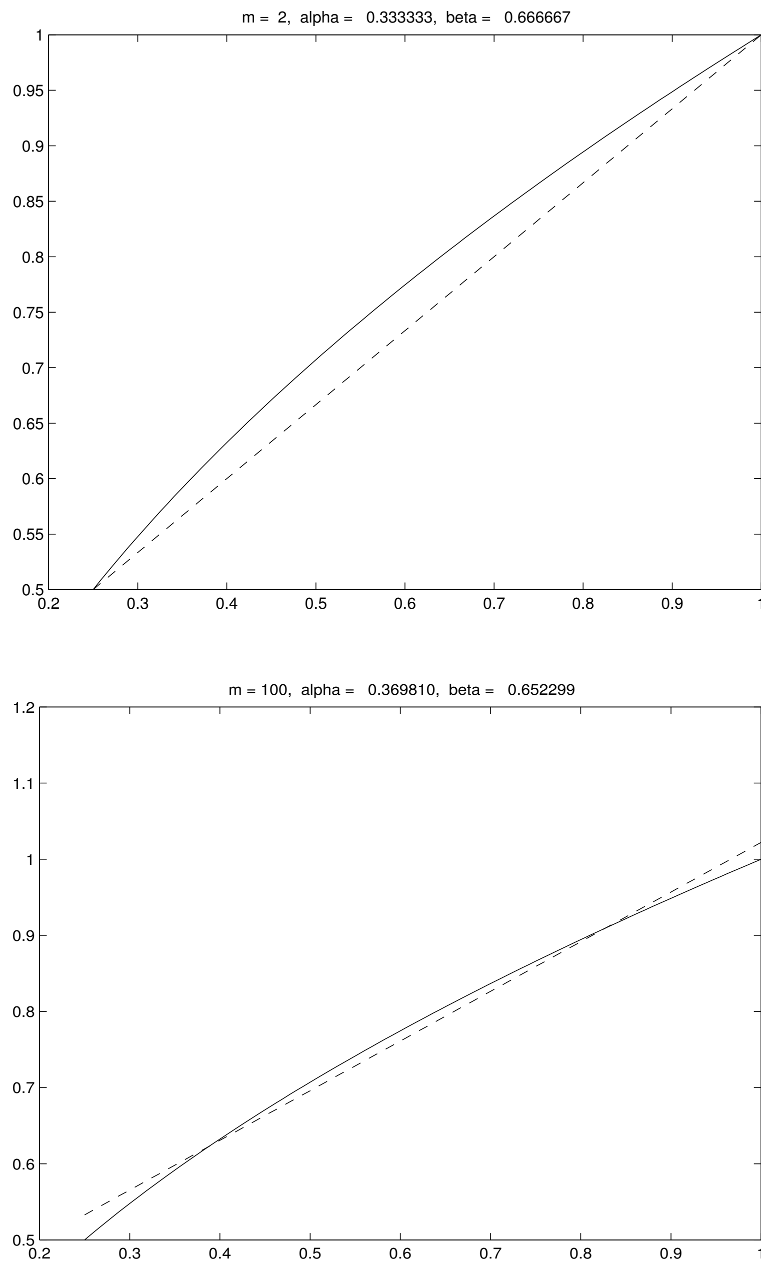

FIGURE 7.1 The LS fitting of a line to $\sqrt{x}$

plots the approximating polynomial and the data. The generation of the polynomial coefficients in c involves about $O(md^2)$ flops.

Finally we mention that some sparse LS problems can be effectively solved using the "\" operator. The following script applies both full and sparse backslash methods on a sequence of bandlike LS problems.

```
% Script File: ShowSparseLS
% Illustrate sparse backslash solving for LS problems

clc
n = 50;
disp(' m      n   full A flops   sparse A flops')
disp('----------------------------------------')
for m = 100:100:1000
   A = tril(triu(rand(m,m),-5),5);
   p = m/n;
   A = A(:,1:p:m);
   A_sparse = sparse(A);
   b = rand(m,1);
   % Solve an m-by-n LS problem where the A matrix has about
   % 10 nonzeros per column. In column j these nonzero entries
   % are more or less A(j*m/n+k,j), k=-5:5.
   flops(0)
   x = A\b;
   f1 = flops;
   flops(0)
   x = A_sparse\b;
   f2 = flops;
   disp(sprintf('%4d  %4d  %10d  %10d  ',m,n,f1,f2))
end
```

The average number of nonzeros per column is fixed at 10. Here are the results:

m	n	full A flops	sparse A flops
100	50	425920	73232
200	50	819338	17230
300	50	1199094	22076
400	50	1472866	19300
500	50	1862884	15878
600	50	1865862	13560
700	50	2174622	14560
800	50	2468358	15560
900	50	2778624	16560
1000	50	3113100	17560

Problems

P7.1.1 Consider the problem

$$\min_{x\in\mathbb{R}} \left\| \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} x - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right\|_p ,$$

where $p = 1, 2$, or ∞. Suppose $b_1 \geq b_2 \geq b_3$. Show that

$$\begin{array}{lcl} p=1 & \Rightarrow & x_{opt} = b_2 \\ p=2 & \Rightarrow & x_{opt} = (b_1+b_2+b_3)/3 \\ p=\infty & \Rightarrow & x_{opt} = (b_1+b_3)/2 \end{array}$$

P7.1.2 Complete the following function:

```
  function [a,b,c] = LSFit(L,R,fname,m)\\
% Assume that fname is a string that names an available function f(x)\\
% that is defined on [L,R]. Returns in a,b, and c values so that if\\
% q(x) = ax$\car$2 + bx + c, then \
%      [q(x(1)) - f(x(1))]$\car$2 + ... + [q(x(m)) - f(x(m))]$\car$2
% is minimized. Here, x(i) = L + h(i-1) where h = (R-L)/(m-1) and m >=3.
```

Efficiency matters.

P7.1.3 Suppose $E \in \mathbb{R}^{n\times n}$ is a given matrix and that $(x_1, y_1), \ldots, (x_n, y_n)$ are distinct. Given a scalar h_1, we wish to determine scalars $h_2, \ldots, h_n$ so that

$$\phi(h_2, \ldots, h_n) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j\neq i}}^{n} |h_j - h_i - e_{ij}|^2 / d_{ij}$$

is minimized, where $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Write a function `h = Leveling(E,x,y,h1)` that does this. `x` and `y` should be column vectors and `h` should be a column n-vector with `h(1) = h1` and `h(2:n)` as the values that minimize the objective function ϕ above. Proceed by setting up a least squares problem $\min\| Ax - b \|_2$ and use the "\" operator. However, you should represent the A-matrix with `sparse` since it has at most two nonzero entries per row.

P7.1.4 Complete the following function:

```
    function  alpha = LSsine(tau,y,n)
%
% tau and y are column m-vectors and n is a positive integer with n<=m.
% alpha is a column n-vector that minimizes the sum
%
%              (f(tau(1)) - y(1))^2 + (f(tau(2)) - y(2))^2 + ... + (f(tau(m)) - y(m))^2
% where
%              f(t) = alpha(1)*sin(pi*t) + alpha(2)*sin(2*pi*t) + ... + alpha(n)*sin(n*pi*t).
%
```

Your implementation should be vectorized. Solve the least square problem using "\".

Let's see how `LSsine` can be used to approximate sinusoidal data that has been contaminated with noise. Define

$$g(t) = 1.7\sin(2\pi t) + .47\sin(4\pi t) + .73\sin(6\pi t) + .8\sin(8\pi t)$$

and compute

```
tau = linspace(0,3,50)';
y   = feval('g',tau) + .4*randn(50,1);
```

Apply `LSsine` to this fuzzy periodic data with `n = 8`. Print the `alpha` vector that is produced and display in a single plot across [0,3], the function $g(\tau)$, the function

$$\tilde{g}(t) = \sum_{j=1}^{8} \alpha_j \sin(j\pi t),$$

and the 50 data points upon which the least squares fit is based. (Use `'o'` in the plot for the display of the data.) Repeat with $n = 4$.

P7.1.5 Assume that `A` is a given n-by-n nonsingular matrix and that `b`, `c`, and `d` are given column n-vectors. Write a MATLAB script that assigns to `alpha` and `beta` the scalars α and β that minimize the 2-norm of the solution to the linear system $Ax = b + \alpha c + \beta d$. Use \ for all linear system solving and all least squares minimization.

7.2 The QR Factorization

In the linear equation setting, the Gaussian elimination approach converts the given linear system $Ax = b$ into an equivalent, easy-to-solve linear system $(MA)x = (Mb)$. The transition from the given system to the transformed system is subject to a strict rule: All row operations applied to A must also be applied to b. We seek a comparable strategy in the least squares setting. Our goal is to produce an m-by-m matrix Q so the given least squares problem

$$\min_{x \in \mathbb{R}^n} \parallel Ax - b \parallel_2$$

is equivalent to a transformed problem

$$\min_{x \in \mathbb{R}^n} \parallel (Q^T A)x - (Q^T b) \parallel_2,$$

where, by design, $Q^T A$ is "simple."

A family of matrices known as *orthogonal matrices* can be used for this purpose. A matrix $Q \in \mathbb{R}^{m \times m}$ is orthogonal if $Q^T = Q^{-1}$, or, equivalently, if $QQ^T = Q^T Q = I$. Here is a 2-by-2 example:

$$Q = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}.$$

For this particular Q, it is easy to show that Qx is obtained by rotating x clockwise by θ radians.

The key property of orthogonal matrices that makes them useful in the least squares context is that they preserve 2-norm. Indeed, if $Q \in \mathbb{R}^{m \times m}$ is orthogonal and $r \in \mathbb{R}^m$, then

$$\parallel Q^T r \parallel_2^2 = (Q^T r)^T (Q^T r) = (r^T Q)(Q^T r) = r^T (QQ^T) r = r^T I_m r = r^T r = \parallel r \parallel_2^2.$$

If Q is orthogonal, then any x that minimizes $\parallel Ax - b \parallel_2$ also minimizes $\parallel (Q^T A)x - (Q^T b) \parallel_2$ since

$$\parallel (Q^T A)x - (Q^T b) \parallel_2 = \parallel Q^T (Ax - b) \parallel_2 = \parallel Ax - b \parallel_2.$$

Our plan is to apply a sequence of orthogonal transformations to A that reduce it to upper triangular form. This will render an equivalent, easy-to-solve problem. For example,

$$\min_{x\in\mathbb{R}^2} \| Ax - b \|_2 = \min_{x\in\mathbb{R}^2} \| (Q^TA)x - (Q^Tb) \|_2 = \min_{x\in\mathbb{R}^2} \left\| \begin{bmatrix} r_{11} & r_{12} \\ 0 & r_{22} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \right\|_2 .$$

Here, $Q^Tb = c$ is the transformed right-hand side. Note that no matter how x_1 and x_2 are chosen, the sum of squares must be at least $c_3^2 + c_4^2$. Thus, we "write off" components 3 and 4 and focus on the reduction in size of components 1 and 2. It follows that the optimum choice for x is that which solves the 2-by-2 upper triangular system

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

Call this solution x_{LS} and note that in this case

$$\min_{x\in\mathbb{R}^2} \| Ax - b \|_2 = \| Ax_{LS} - b \|_2 = \sqrt{c_3^2 + c_4^2}.$$

The columns of an orthogonal matrix define an *orthonormal basis*. From the equation $Q^TQ = I$ we see that the inner product of $Q(:,j)$ with any other column is zero. The columns of Q are mutually orthogonal. Moreover, since $Q(:,j)^TQ(:,j) = 1$, each column has unit 2-norm, explaining the "normal" in "orthonormal."

Finding an orthogonal $Q \in \mathbb{R}^{m\times m}$ and an upper triangular $R \in \mathbb{R}^{m\times n}$ so that $A = QR$ is the QR factorization problem. It amounts to finding an orthonormal basis for the subspace defined by the columns of A. To see this, note that the jth column of the equation $A = QR$ says that

$$A(:,j) = Q(:,1) * R(1,j) + Q(:,2) * R(2,j) + \cdots + Q(:,j) * R(j,j).$$

Thus, *any* column of A is in the span of $\{Q(:,1),\ldots,Q(:,n)\}$. In the example

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

we see that

$$\begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} = 3 \begin{bmatrix} 1/3 \\ 2/3 \\ 2/3 \end{bmatrix} \qquad \begin{bmatrix} -8 \\ -1 \\ 14 \end{bmatrix} = 6 \begin{bmatrix} 1/3 \\ 2/3 \\ 2/3 \end{bmatrix} + 15 \begin{bmatrix} -2/3 \\ -1/3 \\ 2/3 \end{bmatrix}.$$

The observation that the QR factorization can help us solve the least squares problem tells us once again that finding the "right" basis is often the key to solving a linear algebra problem. Recall from Chapter 2 on page 83 the attractiveness of the Newton basis for the polynomial interpolation problem.

7.2.1 Rotations

The Q in the QR factorization can be computed using a special family of orthogonal matrices that are called *rotations*. A 2-by-2 *rotation* is an orthogonal matrix of the form

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

If $x = [\, x_1\ x_2\,]^T$, then it is possible to choose (c, s) so that if $y = Gx$, then $y_2 = 0$. Indeed, from the requirement that

$$y_2 = -sx_1 + cx_2 = 0$$

and the stipulation that $c^2 + s^2 = 1$, we merely set

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

However, a preferred algorithm for computing the (c, s) pair is the following:

```
  function [c,s] = Rotate(x1,x2)
% [c,s] = Rotate(x1,x2)
% x1 and x2 are real scalars and c and s is a cosine-sine pair so
% -s*x1 + c*x2 = 0.

if x2==0
   c = 1;
   s = 0;
else
   if abs(x2)>=abs(x1)
      cotangent = x1/x2;
      s = 1/sqrt(1+cotangent^2);
      c = s*cotangent;
   else
      tangent = x2/x1;
      c = 1/sqrt(1+tangent^2);
      s = c*tangent;
   end
end
```

In this alternative, we guard against the squaring of arbitrarily large numbers and thereby circumvent the problem of *overflow*. Note that the sine and cosine are computed *without* computing the underlying rotation angle θ. No inverse trigonometric functions are involved in the implementation.

Introducing zeros into a vector by rotation extends to higher dimensions. Suppose $m = 4$ and define

$$G(1, 3, \theta) == \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad c = \cos(\theta),\ s = \sin(\theta).$$

This is a rotation in the $(1, 3)$ plane. It is easy to check that $G(1, 3, \theta)$ is orthogonal. Note that

$$G(1,3,\theta)\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} cx_1 + sx_3 \\ x_2 \\ -sx_1 + cx_3 \\ x_4 \end{bmatrix},$$

and we can determine the cosine-sine pair so that the third component is zeroed as follows:

```
[c,s] = rotate(x(1),x(3))
```

For general m, rotations in the (i, k) plane look like this:

$$G(i,k,\theta) \;=\; \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix},$$
$$\qquad\qquad\qquad i \qquad\quad k$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ for some θ. Premultiplication of a matrix by $G(i, k, \theta)$ may be organized as follows assuming that `c` and `s` house the cosine and sine:

```
A([i k],:)  = [c s ; -s c]*A([i k],:)
```

The integer vector `[i k]` is used in this context to extract rows i and k from the matrix A. Note that premultiplication by $G(i, k, \theta)$ effects only rows i and k.

7.2.2 Reduction to Upper Triangular Form

We now show how a sequence of row rotations can be used to upper triangularize a rectangular matrix. The 4-by-3 case illustrates the general idea:

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

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

The index pairs over the arrows indicate the rows that are being rotated. Notice that the zeroing proceeds column-by-column and that within each column, the entries are zeroed from the bottom row on up to the subdiagonal entry. In the m-by-n case we have

```
for j=1:n
   for i=m:-1:j+1
      %Zero A(i,j)
      [c,s] = Rotate(A(i-1,j),A(i,j));
      A(i-1:i,:)  = [c s ; -s c]*A(i-1:i,:);
   end
end
```

Note that when working to zero, the subdiagonal elements in column j, columns 1 through $j-1$ are already zero. It follows that the A update is better written as

```
A(i-1:i,j:n) = [c s ; -s c]*A(i-1:i,j:n);
```

The algorithm produces a sequence of rotations $G_1, G_2, \ldots, G_t$ with the property that

$$G_t \cdots G_1 A = R \quad \text{(upper triangular)}.$$

Thus, if we define

$$Q^T = G_t \cdots G_1$$

then $Q^T A = R$ and

$$A = I \cdot A = (QQ^T)A = Q(Q^T A) = QR.$$

The Q matrix can be built up by accumulating the rotations as they are produced:

$$\begin{array}{ccc} Q & \leftarrow & I \\ Q & \leftarrow & QG_1^T \\ Q & \leftarrow & QG_2^T \\ & \vdots & \end{array}$$

Packaging all these ideas, we get

```
  function [Q,R] = QRRot(A)
% [Q,R] = QRRot(A)
%
% The QR factorization of an m-by-n matrix A. (m>=n).
% Q is m-by-m orthogonal and R is m-by-n upper triangular.

[m,n] = size(A);
Q = eye(m,m);
for j=1:n
   for i=m:-1:j+1
      %Zero A(i,j)
      [c,s] = Rotate(A(i-1,j),A(i,j));
      A(i-1:i,j:n) = [c s; -s c]*A(i-1:i,j:n);
      Q(:,i-1:i) = Q(:,i-1:i)*[c s; -s c]';
   end
end
R = triu(A);
```

A flop count reveals that this algorithm requires about $9mn^2 - 3n^3$ flops. If $n = m$, then this is about six times as costly as the LU factorization.

7.2.3 The Least Squares Solution

Once we have the QR factorization of A, then the given least squares problem of minimizing $\| Ax - b \|_2$ transforms as follows:

$$\begin{aligned} \| Ax - b \|_2 &= \| Q^T(Ax - b) \|_2 \;=\; \| (Q^TA)x - (Q^Tb) \|_2 \;=\; \| Rx - c \|_2 \\ &= \left\| \begin{bmatrix} A_1 \\ 0 \end{bmatrix} x - \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \right\|_2 = \left\| \begin{pmatrix} A_1x - c_1 \\ -c_2 \end{pmatrix} \right\|_2 . \end{aligned}$$

Here $A_1 = Q(:,1{:}n)^TA$ is made up of the first n rows of Q^TA, $c_1 = Q(:,1{:}n)^Tb$ is made up of the top n components of Q^Tb, and $c_2 = Q(:,n+1{:}m)^Tb$ is made up of the bottom $m-n$ components of Q^Tb. Notice that no matter how x is chosen, $\| Ax - b \|_2 \geq \| c_2 \|_2$. The lower bound can be achieved if we solve the upper triangular system $A_1x = c_1$. It follows that the least squares solution x_{LS} and the norm of the *residual* $\| Ax_{LS} - b \|_2$ are prescribed by

```
  function [xLS,res] = LSq(A,b)
% [xLS,res] = LSq(A,b)
% Solution to the LS problem min norm(Ax-b) where A is a full
% rank m-by-n matrix with m>=n and b is a column m-vector.
% xLS is the n-by-1 vector that minimizes the norm(Ax-b) and
% res = norm(A*xLS-b).
[m,n] = size(A);
for j=1:n
   for i=m:-1:j+1
      %Zero A(i,j)
      [c,s] = Rotate(A(i-1,j),A(i,j));
      A(i-1:i,j:n) = [c s; -s c]*A(i-1:i,j:n);
      b(i-1:i) = [c s; -s c]*b(i-1:i);
   end
end
xLS = UTriSol(A(1:n,1:n),b(1:n));
if m==n
   res = 0;
else
   res = norm(b(n+1:m));
end
```

In this implementation, Q is not explicitly formed as in `QRrot`. Instead, the rotations that "make up" Q are applied to b as they are generated. This algorithm requires about $3mn^2 - n^3$ flops. Note that if $m = n$, then the solution to the square linear system $Ax = b$ is produced at a cost of $2n^3$ flops, three times what is required by Gaussian elimination.

7.2.4 Solution Sensitivity

As in the linear equation problem, it is important to appreciate the sensitivity of the LS solution to perturbations in the data. The notion of condition extends to rectangular matrices. We cannot use the definition $\kappa_2(A) = \| A \| \| A^{-1} \|$ anymore since A, being rectangular, does not have an inverse. (It is natural to use the 2-norm in the LS problem.) Instead we use an equivalent formulation that makes sense for both square and rectangular matrices:

$$\frac{1}{\kappa_2(A)} = \min_{\text{rank}(A+E)<n} \frac{\| E \|_2}{\| A \|_2}.$$

Roughly speaking, the inverse of the condition is the relative distance to the nearest rank deficient matrix. If the columns of A are nearly dependent, then its condition number is large.

It can be shown with reasonable assumptions that if $\tilde{x}_{LS}$ solves the perturbed LS problem

$$\min \| (A + \Delta A)x - (b + \Delta b) \|_2, \quad \text{where} \quad \| \Delta A \|_2 \approx \texttt{eps} \| A \|_2, \ \| \Delta b \|_2 \approx \texttt{eps} \| b \|_2, \qquad (7.1)$$

then

$$\frac{\| \tilde{x}_{LS} - x_{LS} \|}{\| x_{LS} \|} \approx \texttt{eps} \left(\kappa_2(A) + \rho_{LS} \kappa_2(A)^2 \right),$$

where $\rho_{LS} = \| Ax_{LS} - b \|_2$. If b is in the span of A's columns, then this is essentially the same result obtained for the linear equation problem: $O(\texttt{eps})$ errors in the data show up as $O(\kappa_2(A)\texttt{eps})$ errors in the solution. However, if the minimum residual is nonzero, then the *square* of the condition number is involved, and this can be much larger than $\kappa_2(A)$. The script file `ShowLSq` illustrates this point. Here are the results for a pair of randomly generated, 10-by-4 LS problems:

```
m = 10, n =  4, cond(A) = 1.000e+07

Zero residual problem:

    Exact Solution        Computed Solution
-----------------------------------------
1.0000000000000000     1.0000000003318523
1.0000000000000000     1.0000000003900065
1.0000000000000000     0.9999999992968892
1.0000000000000000     1.0000000001162717

Nonzero residual problem:

    Exact Solution        Computed Solution
-----------------------------------------
1.0000000000000000     1.0010816095684492
1.0000000000000000     1.0012711526858316
1.0000000000000000     0.9977083453465591
1.0000000000000000     1.0003789656343001
```

Both problems are set up so that the exact solution is the vector of all ones. Notice that the errors are greater in the nonzero residual example, reflecting the effect of the $\kappa_2(A)^2$ factor.

The computed solution obtained by `LSq` satisfies a nearby LS problem in the sense of (7.1). The method is therefore stable. But as we learned in the previous chapter and as the preceding example shows, stability does not guarantee accuracy.

Problems

P7.2.1 Suppose $b \in \mathbb{R}^n$ and $H \in \mathbb{R}^{n\times n}$ is upper Hessenberg ($h_{ij} = 0,\ i > j+1$). Show how to construct rotations $G_1, \ldots, G_{n-1}$ so that $G_{n-1}^T \cdots G_1^T H = R$ is upper triangular. (Hint: G_k should zero $h_{k+1,k}$.) Write a MATLAB function `x = HessQRSolve(H,b)` that uses this reduction to solve the linear system $Hx = b$.

P7.2.2 Given

$$A = \begin{bmatrix} w & x \\ y & z \end{bmatrix},$$

show how to construct a rotation

$$Q = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

so that the $(1,2)$ and $(2,1)$ entries in Q^TA are the same.

P7.2.3 Suppose $A \in \mathbb{R}^{n\times n}$. Show how to premultiply A by a sequence of rotations so that A is transformed to *lower* triangular form.

P7.2.4 Modify `QRrot` so that it efficiently handles the case when A is upper Hessenberg.

P7.2.5 Write a MATLAB fragment that prints the a and b that minimize

$$\phi(a,b) = \sum_{i=1}^{m} [(a + bx_i) - \sqrt{x_i}\,]^2 \quad \text{where} \quad x = .25{:}h{:}1,\ h = .75/(m-1)$$

for $m = 2,3,4,5,6,7,8,9,10,20,40,80,200$. Use `LS`.

P7.2.6 Write a MATLAB function `Rotate1` that is just like `Rotate` except that it zeros the top component of a 2-vector.

7.3 The Cholesky Factorization

A matrix $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A = A^T$ and *positive definite* if $x^TAx > 0$ for all nonzero $x \in \mathbb{R}^n$. Symmetric positive definite (SPD) matrices are the single most important class of specially structured matrices that arise in applications. Here are some important facts associated with such matrices:

1. If $A = (a_{ij})$ is SPD, then its diagonal entries are positive and for all i and j,

$$|a_{ij}| \le (a_{ii} + a_{jj})/2.$$

   This says that the largest entry in an SPD matrix is on the diagonal and, more qualitatively, that SPD matrices have more "mass" on the diagonal than off the diagonal.

2. If $A \in \mathbb{R}^{n \times n}$ and
$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|, \qquad i = 1{:}n,$$
then A is *strictly diagonally dominant*. If A is also symmetric with positive diagonal entries, then it is SPD. (See Theorem 7 below.)

3. If $C \in \mathbb{R}^{m \times n}$ has independent columns and $A = C^T C$, then A is SPD. This fact paves the way to the *method of normal equations* for the full-rank LS problem min $\| Ax - b \|_2$. It turns out that x_{LS} solves the SPD system $A^T Ax = A^T b$. (See §7.3.5.)

4. If A is SPD, then it is possible to find a lower triangular matrix G so that
$$A = GG^T.$$
This is called the *Cholesky factorization*. Once it is obtained the solution to $Ax = b$ can be determined via forward and back substitution: $Gy = b$, $G^T x = y$. It is an attractive solution procedure, because it involves half the number of flops required by Gaussian elimination and there is no need to pivot for stability.

Our primary goal is to develop a sequence of Cholesky implementations that feature different "kernel operations," just as we did with matrix multiplication in §5.2.3. The design of a high-performance linear equation solver for an advanced computer hinges on being able to formulate the algorithm in terms of linear algebra that is "friendly" to the underlying architecture. Our Cholesky presentation is an occasion to elevate our matrix-vector skills in this direction.

7.3.1 Positive Definiteness

We start with an application that builds intuition for positive definiteness and leads to a particularly simple Cholesky problem. Positive definite matrices frequently arise when differential equations are discretized. Suppose $p(x)$, $q(x)$, and $r(x)$ are known functions on an interval $[a, b]$, and that we wish to find an unknown function $u(x)$ that satisfies
$$-D\left[p(x)Du(x)\right] + q(x)u(x) = r(x), \qquad a \leq x \leq b \tag{7.2}$$
with $u(a) = u(b) = 0$. Here, D denotes differentiation with respect to x. This is an example of a *two-point boundary value problem*, and there are several possible solution frameworks. We illustrate the method of *finite differences*, a technique that discretizes the derivatives and culminates in a system of linear equations.

Let n be a positive integer and set $h = (b - a)/(n - 1)$. Define
$$\begin{array}{lcll} x_i &=& a + (i-1)h & i = 1{:}n \\ p_i &=& p(x_i + h/2) & i = 1{:}n-1 \\ q_i &=& q(x_i) & i = 1{:}n \\ r_i &=& r(x_i) & i = 1{:}n \end{array}$$

and let u_i designate an approximation to $u(x_i)$ whose value we seek. Set $u_1 = u_n = 0$ since $u(x)$ is zero at the endpoints. From our experience with divided differences,

$$Du(x_i + h/2) \approx \frac{u_{i+1} - u_i}{h}$$

$$Du(x_i - h/2) \approx \frac{u_i - u_{i-1}}{h}$$

and so we obtain the following approximation:

$$D\left[p(x)Du(x)\right]\big|_{x=x_i} \approx \frac{p_i Du(x_i + h/2) - p_{i-1}Du(x_i - h/2)}{h} \approx \frac{p_i \dfrac{u_{i+1} - u_i}{h} - p_{i-1}\dfrac{u_i - u_{i-1}}{h}}{h}.$$

If we set $x = x_i$ in (7.2) and substitute this discretized version of the $D[p(x)Du(x)]$ term, then we get

$$\frac{1}{h^2}\left(-p_{i-1}u_{i-1} + (p_{i-1} + p_i)u_i - p_i u_{i+1}\right) + q_i u_i = r_i, \qquad i = 2{:}n-1.$$

Again recalling that $u_1 = u_n = 0$, we see that this defines a system of $n - 2$ equations in the $n - 2$ unknowns $u_2, \ldots, u_{n-1}$. For example, if $n = 6$, then we obtain

$$\begin{bmatrix} p_1 + p_2 + h^2 q_2 & -p_2 & 0 & 0 \\ -p_2 & p_2 + p_3 + h^2 q_3 & -p_3 & 0 \\ 0 & -p_3 & p_3 + p_4 + h^2 q_4 & -p_4 \\ 0 & 0 & -p_4 & p_4 + p_5 + h^2 q_5 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} h^2 r_2 \\ h^2 r_3 \\ h^2 r_4 \\ h^2 r_5 \end{bmatrix}.$$

This is a symmetric tridiagonal system. If the functions $p(x)$ and $q(x)$ are positive on $[a, b]$, then the matrix T has strict diagonal dominance and is positive definite. The following theorem offers a rigorous proof of this fact.

Theorem 7 *If*

$$T = \begin{bmatrix} d_1 & e_2 & 0 & \cdots & 0 \\ e_2 & d_2 & \ddots & & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & d_{n-1} & e_{n-1} \\ 0 & 0 & \cdots & e_{n-1} & d_n \end{bmatrix}$$

has the property that

$$d_i > \begin{cases} |e_2| & \text{if } i = 1 \\ |e_i| + |e_{i+1}| & \text{if } 2 \le i \le n-1 \\ |e_n| & \text{if } i = n \end{cases},$$

then T is positive definite.

Proof If $n = 5$, then

$$x^T T x = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} \begin{bmatrix} d_1 & e_2 & 0 & 0 & 0 \\ e_2 & d_2 & e_3 & 0 & 0 \\ 0 & e_3 & d_3 & e_4 & 0 \\ 0 & 0 & e_4 & d_4 & e_5 \\ 0 & 0 & 0 & e_5 & d_5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

$$= \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} \begin{bmatrix} d_1x_1 + e_2x_2 \\ e_2x_1 + d_2x_2 + e_3x_3 \\ e_3x_2 + d_3x_3 + e_4x_4 \\ e_4x_3 + d_4x_4 + e_5x_5 \\ e_5x_4 + d_5x_5 \end{bmatrix}$$

$$= (d_1x_1^2 + d_2x_2^2 + d_3x_3^2 + d_4x_4^2 + d_5x_5^2) \; + \; 2(e_2x_1x_2 + e_3x_2x_3 + e_4x_3x_4 + e_5x_4x_5).$$

In general, we find that

$$x^T T x = \sum_{i=1}^{n} d_i x_i^2 \; + \; 2\sum_{i=2}^{n} e_i x_{i-1} x_i.$$

Our goal is to show that if $x \neq 0$, then this quantity is strictly positive. The first summation is obviously positive. The challenge is to show that the second summation cannot be too negative. Since $0 \leq (x_{i-1} - x_i)^2 = x_{i-1}^2 - 2x_{i-1}x_i + x_i^2$, it follows that $2|x_{i-1}||x_i| \; \leq \; x_{i-1}^2 + x_i^2$. Therefore,

$$\begin{aligned} x^T T x &\geq \sum_{i=1}^{n} d_i x_i^2 \; - \; 2\sum_{i=2}^{n} |e_i||x_{i-1}||x_i| \\ &\geq \sum_{i=1}^{n} d_i x_i^2 \; - \; \sum_{i=2}^{n} |e_i|(x_{i-1}^2 + x_i^2) \\ &= (d_1 - |e_2|)\, x_1^2 \; + \; \sum_{i=2}^{n-1} (d_i - |e_i| - |e_{i+1}|)\, x_i^2 \; + \; (d_n - |e_n|)\, x_n^2. \end{aligned}$$

The theorem follows because in this last expression, every quantity in parentheses is positive and at least one of the x_i is nonzero. □

Some proofs of positive definiteness are straightforward like this, and others are more difficult. Once positive definiteness is established, then we have a license to compute the Cholesky factorization. We are ready to show how this is done.

7.3.2 Tridiagonal Cholesky

It turns out that a symmetric tridiagonal positive definite matrix

$$T = \begin{bmatrix} d_1 & e_2 & 0 & 0 & 0 \\ e_2 & d_2 & e_3 & 0 & 0 \\ 0 & e_3 & d_3 & e_4 & 0 \\ 0 & 0 & e_4 & d_4 & e_5 \\ 0 & 0 & 0 & e_5 & d_5 \end{bmatrix}, \qquad (n = 5),$$

has a Cholesky factorization $T = GG^T$ with a lower bidiagonal G:

$$G = \begin{bmatrix} g_1 & 0 & 0 & 0 & 0 \\ h_2 & g_2 & 0 & 0 & 0 \\ 0 & h_3 & g_3 & 0 & 0 \\ 0 & 0 & h_4 & g_4 & 0 \\ 0 & 0 & 0 & h_5 & g_5 \end{bmatrix}.$$

To see this, we equate coefficients

$$\begin{array}{lllllll} (1,1): & d_1 & = & g_1^2 & \Rightarrow & g_1 & = & \sqrt{d_1} \\ (2,1): & e_2 & = & h_2 g_1 & \Rightarrow & h_2 & = & e_2/g_1 \\ (2,2): & d_2 & = & h_2^2 + g_2^2 & \Rightarrow & g_2 & = & \sqrt{d_2 - h_2^2} \\ (3,2): & e_3 & = & h_3 g_2 & \Rightarrow & h_3 & = & e_3/g_2 \\ (3,3): & d_3 & = & h_3^2 + g_3^2 & \Rightarrow & g_3 & = & \sqrt{d_3 - h_3^2} \\ & & & & \vdots & & & \end{array}$$

and conclude that for $i = 1{:}n$, $g_i = \sqrt{d_i - h_i^2}$ and $h_i = e_i/g_{i-1}$ (set $h_1 \equiv 0$). This yields the following function:

```
  function [g,h] = CholTrid(d,e)
% G = CholTrid(d,e)
% Cholesky factorization of a symmetric, tridiagonal positive definite matrix A.
% d and e are column n-vectors with the property that
% A = diag(d) + diag(e(2:n),-1) + diag(e(2:n),1)
%
% g and h are column n-vectors with the property that the lower bidiagonal
% G = diag(g) + diag(h(2:n),-1) satisfies A = GG^T.

n = length(d);
g = zeros(n,1);
h = zeros(n,1);
g(1) = sqrt(d(1));
for i=2:n
   h(i) = e(i)/g(i-1);
   g(i) = sqrt(d(i) - h(i)^2);
end
```

It is clear that this algorithm requires $O(n)$ flops. Since G is lower bidiagonal, the solution of $Ax = b$ via $Gy = b$ and $G^T x = y$ proceeds as follows:

```
  function x = CholTridSol(g,h,b)
% x = CholTridSol(g,h,b)
%
% Solves the linear system G*G'x = b where b is a column n-vector and
% G is a nonsingular lower bidiagonal matrix. g and h are column n-vectors
% with G = diag(g) + diag(h(2:n),-1).

n = length(g);
y = zeros(n,1);

% Solve Gy = b
y(1) = b(1)/g(1);
for k=2:n
   y(k) = (b(k) - h(k)*y(k-1))/g(k);
end

% Solve G'x = y
x = zeros(n,1);
x(n) = y(n)/g(n);
for k=n-1:-1:1
   x(k) = (y(k) - h(k+1)*x(k+1))/g(k);
end
```

See §6.2 for a discussion of bidiagonal system solvers. Overall we see that the complete solution to a tridiagonal SPD system involves $O(n)$ flops.

7.3.3 Five Implementations for Full Matrices

Here is an example of a 3-by-3 Cholesky factorization:

$$\begin{bmatrix} 4 & -10 & 2 \\ -10 & 34 & 17 \\ 2 & -17 & 18 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ -5 & 3 & 0 \\ 1 & -4 & 1 \end{bmatrix} \begin{bmatrix} 2 & -5 & 1 \\ 0 & 3 & -4 \\ 0 & 0 & 1 \end{bmatrix}.$$

An algorithm for computing the entries in the Cholesky factor can be derived by equating entries in the equation $A = GG^T$:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} g_{11} & 0 & 0 \\ g_{21} & g_{22} & 0 \\ g_{31} & g_{32} & g_{33} \end{bmatrix} \begin{bmatrix} g_{11} & g_{21} & g_{31} \\ 0 & g_{22} & g_{32} \\ 0 & 0 & g_{33} \end{bmatrix}.$$

Since

$$\begin{aligned} a_{11} &= g_{11}^2 \\ a_{21} &= g_{11}g_{21} \quad a_{22} = g_{21}^2 + g_{22}^2 \\ a_{31} &= g_{11}g_{31} \quad a_{32} = g_{21}g_{31} + g_{22}g_{32} \quad a_{33} = g_{31}^2 + g_{32}^2 + g_{33}^2 \end{aligned}$$

we obtain

$$\begin{aligned} g_{11} &= \sqrt{a_{11}} \\ g_{21} &= a_{21}/g_{11} \\ g_{22} &= \sqrt{a_{22} - g_{21}^2} \\ g_{31} &= a_{31}/g_{11} \\ g_{32} &= (a_{32} - g_{21}g_{31})/g_{22} \\ g_{33} &= \sqrt{a_{33} - g_{31}^2 - g_{32}^2}\,. \end{aligned}$$

The algorithm would break down if any of the g_{ii} are zero or complex. But the property of being positive definite guarantees that neither of these things happen.

To derive the Cholesky algorithm for general n, we repeat the preceding methodology and compare entries in the equation $A = GG^T$. If $i \geq j$, then

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

and so

$$g_{ij} = \begin{cases} \sqrt{s_{jj}} & i = j \\ s_{ij}/g_{jj} & i > j \end{cases}.$$

If we compute the lower triangular matrix G row-by-row as we did in the 3-by-3 example, then we obtain the following implementation:

```
  function G = CholScalar(A)
% G = CholScalar(A)
% Cholesky factorization of a symmetric and positive definite matrix A.
% G is lower triangular so A = G*G'.
[n,n] = size(A);
G = zeros(n,n);
for i=1:n
   % Compute G(i,1:i)
   for j=1:i
      s = A(j,i);
      for k=1:j-1
         s = s - G(j,k)*G(i,k);
      end
      if j<i
         G(i,j) = s/G(j,j);
      else
         G(i,i) = sqrt(s);
      end
   end
end
```

An assessment of the work involved reveals that this implementation of Cholesky requires $n^3/3$ flops. This is half of the work required by Gaussian elimination, to be expected since the problem involves half of the data.

Notice that the k-loop in `CholScalar` oversees an inner product between subrows of G. With this observation we obtain the following dot product implementation:

```
  function G = CholDot(A)
% G = CholDot(A)
% Cholesky factorization of a symmetric and positive definite matrix A.
% G is lower triangular so A = G*G'.
[n,n] = size(A); G = zeros(n,n);
for i=1:n
   % Compute G(i,1:i)
   for j=1:i
      if j==1
         s = A(j,i);
      else
         s = A(j,i) - G(j,1:j-1)*G(i,1:j-1)';
      end
      if j<i
         G(i,j) = s/G(j,j);
      else
         G(i,i) = sqrt(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 saxpys are level-1 operations. Notice that `CholDot` is row oriented because the ith and jth rows of G are accessed during the inner product.

A column-oriented version that features the saxpy operation can be derived by comparing the j-th columns in the equation $A = GG^T$:

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

This can be solved for $G(:,j)$:

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

But since G is lower triangular, we need only focus on the nonzero portion of this vector:

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

Since $G(j,j) = \sqrt{s(j)}$, it follows that $G(j{:}n,j) = s(j{:}n)/\sqrt{s(j)}$. This leads to the following implementation:

```
  function G = CholSax(A)
% G = CholSax(A)
% Cholesky factorization of a symmetric and positive definite matrix A.
% G is lower triangular so A = G*G'.
[n,n] = size(A); G = zeros(n,n); s = zeros(n,1);
for j=1:n
   s(j:n) = A(j:n,j);
   for k=1:j-1
      s(j:n) = s(j:n) - G(j:n,k)*G(j,k);
   end
   G(j:n,j) = s(j:n)/sqrt(s(j));
end
```

Notice that as i ranges from j to n, the kth column of G is accessed in the inner loop. From the flop point of view, `CholSax` is identical to `CholDot`.

An update of the form

$$\text{Vector} \;\leftarrow\; \text{Vector} \;+\; \text{Matrix} \times \text{Vector}$$

is called a *gaxpy* operation. Notice that the `k`-loop in `CholSax` oversees the gaxpy operation

$$\begin{aligned} s(j{:}n) \quad &\leftarrow \quad s(j{:}n) - \begin{bmatrix} G(j,1{:}j-1) \\ G(j+1,1{:}j-1) \\ \vdots \\ G(n,1{:}j-1) \end{bmatrix} \begin{bmatrix} G(j,1) \\ G(j,2) \\ \vdots \\ G(j,j-1) \end{bmatrix} \\ &= \quad s(j{:}n) - G(j{:}n,1{:}j-1)G(j,1{:}j-1)^T. \end{aligned}$$

Substituting this observation into `CholSax` gives

```
  function G = CholGax(A)
% G = CholGax(A)
% Cholesky factorization of a symmetric and positive definite matrix A.
% G is lower triangular so A = G*G'.
[n,n] = size(A); G = zeros(n,n); s = zeros(n,1);
for j=1:n
   if j==1
      s(j:n) = A(j:n,j);
   else
      s(j:n) = A(j:n,j) - G(j:n,1:j-1)*G(j,1:j-1)';
   end
   G(j:n,j)  = s(j:n)/sqrt(s(j));
end
```

The gaxpy operation is a *level-2* operation. Level-2 operations are characterized by quadratic work and quadratic data. For example, in an m-by-n gaxpy operation, $O(mn)$ data is involved and $O(mn)$ flops are required.

Finally, we developed a recursive implementation. Suppose that $n > 1$ and that

$$A = \begin{bmatrix} B & v \\ v & \alpha \end{bmatrix}$$

is SPD, where $B = A(1{:}n-1, 1{:}n-1)$, $v = A(1{:}n-1, n)$, and $\alpha = A(n, n)$. It is easy to show that B is also SPD. If

$$G = \begin{bmatrix} G_1 & 0 \\ w & \beta \end{bmatrix}$$

is partitioned the same way and we equate blocks in the equation

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

then we find that $B = G_1 G_1^T$, $v = G_1 w$, and $\alpha = \beta^2 + w^T w$. This says that if G is to be the Cholesky factor of A, then it can be synthesized by (1) computing the Cholesky factor G_1 of B, (2) solving the lower triangular system $G_1 w = v$ for w, and (3) computing $\beta = \sqrt{\alpha - w^T w}$. The square root is guaranteed to render a real number because

$$\begin{aligned} 0 &< \begin{bmatrix} -B^{-1}v \\ 1 \end{bmatrix}^T \begin{bmatrix} B & v \\ v & \alpha \end{bmatrix} \begin{bmatrix} -B^{-1}v \\ 1 \end{bmatrix} \\ &= \alpha - v^T B^{-1} v \\ &= \alpha - v^T (G_1 G_1^T)^{-1} v \\ &= \alpha - (G_1^{-1} v)^T (G_1^{-1} v) \\ &= \alpha - w^T w. \end{aligned}$$

This is the basis for the following recursive implementation:

```
  function G = CholRecur(A);
% G = CholRecur(A)
% Cholesky factorization of a symmetric and positive definite matrix A.
% G is lower triangular so A = G*G'.
[n,n] = size(A);
if n==1
   G = sqrt(A);
else
   G(1:n-1,1:n-1) = CholRecur(A(1:n-1,1:n-1));
   G(n,1:n-1)     = LTriSol(G(1:n-1,1:n-1),A(1:n-1,n))';
   G(n,n)         = sqrt(A(n,n) - G(n,1:n-1)*G(n,1:n-1)');
end
```

If this function is applied to the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{bmatrix}^T$$

and the semicolon is deleted from the recursive call command, then the following sequence of matrices is displayed:

$$\begin{bmatrix} 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{bmatrix}.$$

From this we infer that `CholRecur` computes the Cholesky factor G row by row, just like `CholScalar`.

7.3.4 Efficiency, Stability, and Accuracy

The script file `CholBench` benchmarks the five implementations. Here are the results from a sample $n = 64$ run:

```
Algorithm     Time     Flops
------------------------------
CholScalar    1.000   91538
CholDot       0.405   95443
CholSax       0.273   89586
CholGax       0.050   91601
CholRecur     0.455   96072
```

The times reported are relative to the time required by `CholScalar`. The concern is not so much about the actual values in the table but the fact that different implementations of the same matrix algorithm can have different levels of performance.

In Gaussian elimination we had to worry about large multipliers. This is not an issue in the Cholesky factorization because the equation

$$a_{ii} = \sum_{j=1}^{i} g_{ij}^2$$

implies $|g_{ij}| \leq \sqrt{a_{ii}}$. Thus no entry in G can be larger than the square root of A's largest diagonal entry. This does *not* imply that all symmetric positive definite systems are well conditioned. The computed solution is prone to the same kind of error that we saw in Gaussian elimination. In particular, if $\hat{x}$ is the computed vector produced by

```
G = CholScalar(A);
y = LTriSol(G,b);
x = UTriSol(G',y);
```

then

$$(A+E)\hat{x} = b, \qquad \| E \| \approx \texttt{eps}\| A \|,$$

and

$$\frac{\| \hat{x} - x \|}{\| x \|} \approx \texttt{eps}\, \kappa(A).$$

The script file `CholErr` can be used to examine these heuristics. It solves a series of SPD systems that involve the *Hilbert* matrices. These matrices can be generated with the built-in MATLAB function `hilb(n)`. Their condition number increases steeply with n and their exact inverse can be obtained by calling `invhilb`. This enables us to compute the true solution and therefore the exact relative error. Here are the results:

n	cond(A)	relerr
2	1.928e+01	1.063e-16
3	5.241e+02	1.231e-15
4	1.551e+04	1.393e-13
5	4.766e+05	1.293e-12
6	1.495e+07	9.436e-11
7	4.754e+08	3.401e-09
8	1.526e+10	1.090e-08
9	4.932e+11	3.590e-06
10	1.603e+13	1.081e-04
11	5.216e+14	2.804e-03
12	1.668e+16	5.733e-02

Relative error deteriorates with increasing condition number in the expected way. `CholErr` uses `CholScalar`, but the results would be no different were any of the other implementations used.

7.3.5 The MATLAB CHOL Function and the LS Problem

When applied to an SPD matrix A, the MATLAB function `Chol(A)` returns an *upper* triangular matrix R so that $A = R^T R$. Thus, R is the transpose of the matrix G that we have been calling the Cholesky factor.

The MATLAB Cholesky style highlights an important connection between the Cholesky and QR factorizations. If $A \in \mathbb{R}^{m \times n}$ has full column rank and $A = QR$ is its QR factorization, then $\tilde{R} = R(1{:}n, 1{:}n)$ defines the MATLAB Cholesky factor of $A^T A$:

$$A^T A = (QR)^T (QR) = (R^T Q^T)(QR) = R^T (Q^T Q) R = R^T R = \tilde{R}^T \tilde{R}.$$

Note that

$$A = QR = Q(:, 1{:}n)\tilde{R},$$

and so the LS minimizer of $\| Ax - b \|_2$ is given by

$$x_{LS} = \tilde{R}^{-1} Q(:, 1{:}n)^T b = \tilde{R}^{-1}(A\tilde{R}^{-1})^T b = (\tilde{R}^T \tilde{R})^{-1} A^T b = (A^T A)^{-1} A^T b.$$

Thus, x_{LS} is the solution to the SPD system

$$A^T A x = A^T b.$$

Solving the LS problem by computing the Cholesky factorization of A^TA is called the method of normal equations. The method works well on some problems but in general is not as numerically sound as the QR method outlined in the previous section.

Problems

P7.3.1 Complete the following function:

```
  function uvals = TwoPtBVP(n,a,b,pname,qname,rname)
% uvals = TwoPtBVP(n,a,b,pname,qname,rname)
%
% a and b are reals with a<b and n is an integer with >= 3.
% pname, qname, and rname are strings that name functions defined on [a,b], the
% first two of which are positive on the interval.
%
% uvals is a column n-vector with the property that uvals(i) approximates
% the solution to the 2-point boundary value problem
%
%              -D[p(x)Du(x)] + q(x)u(x) = r(x)    a<=x<=b
%               u(a) = u(b) = 0
%
% at x = a+(i-1)h where h = (b-a)/(n-1)
```

Use it to solve

$$-D[(2 + x\cos(20\pi x)Du(x)] + (20\sin(1/(.1+x)^2))u(x) \;=\; 1000(x-.5)^3$$

with $u(0) = u(1) = 0$. Set $n = 400$ and plot the solution.

P7.3.2 The derivation of an outer product Cholesky implementation is based on the following triplet of events:

- Compute `G(1,1) = sqrt(A(1,1))`.
- Scale to get the rest of G's first column: `G(2:n,1) = A(2:n,1)/G(1,1)`.
- Repeat on the reduced problem `A(2:n,2:n)-G(2:n,1)*G(2:n,1)'`.

Using this framework, write a recursive implementation of `CholOuter`. To handle the base case, observe that if $n = 1$, then the Cholesky factor is just the square root of $A = A(1,1)$.

P7.3.3 A positive definite matrix of the form

$$A = \begin{bmatrix} d_1 & e_2 & f_3 & 0 & 0 & 0 & 0 & 0 \\ e_2 & d_2 & e_3 & f_4 & 0 & 0 & 0 & 0 \\ f_3 & e_3 & d_3 & e_4 & f_5 & 0 & 0 & 0 \\ 0 & f_4 & e_4 & d_4 & e_5 & f_6 & 0 & 0 \\ 0 & 0 & f_5 & e_5 & d_5 & e_6 & f_7 & 0 \\ 0 & 0 & 0 & f_6 & e_6 & d_6 & e_7 & f_8 \\ 0 & 0 & 0 & 0 & f_7 & e_7 & d_7 & e_8 \\ 0 & 0 & 0 & 0 & 0 & f_8 & e_8 & d_8 \end{bmatrix}$$

has a factorization $A = GG^T$, where

$$G = \begin{bmatrix} g_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ h_2 & g_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ p_3 & h_3 & g_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & p_4 & h_4 & g_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & p_5 & h_5 & g_5 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_6 & h_6 & g_6 & 0 & 0 \\ 0 & 0 & 0 & 0 & p_7 & h_7 & g_7 & 0 \\ 0 & 0 & 0 & 0 & 0 & p_8 & h_8 & g_8 \end{bmatrix}.$$

By comparing coefficients in the equation $A = GG^T$, develop a MATLAB function `[g,h,p] = Chol5(d,e,f)` that computes the vectors `g(1:n)`, `h(2:n)`, and `p(3:n)` from `d(1:n)`, `e(2:n)`, and `f(3:n)`. Likewise, develop triangular

system solvers `LTriSol5(g,h,p,b)` and `UTriSol5(g,h,p,b)` that can be used to solve systems of the form $Gx = b$ and $G^T x = b$. To establish the correctness of your functions, use them to solve a 10-by-10 system $Ax = b$, where $d_i = 100 + i$, $e_i = 10 + i$, $f_i = i$ with $b = 100 * ones(10, 1)$. Produce a plot that shows how long it takes your functions to solve an n-by-n system for $10 \leq n \leq 100$. Print a table showing the number of flops required to solve $Ax = b$ for $n = 10, 20, 40, 80, 160$.

P7.3.4 Assume that

$$A = \begin{bmatrix} a_1 & c_2 & 0 & 0 \\ c_2 & a_2 & c_3 & 0 \\ 0 & c_3 & a_3 & c_4 \\ 0 & 0 & c_4 & a_4 \end{bmatrix}$$

is positive definite. **(a)** Show how to compute $d_1, \ldots, d_5$ and $e_2, \ldots, e_5$ so that if

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ e_2 & 1 & 0 & 0 \\ 0 & e_3 & 1 & 0 \\ 0 & 0 & e_4 & 1 \end{bmatrix} \text{ and } D = \begin{bmatrix} d_1 & 0 & 0 & 0 \\ 0 & d_2 & 0 & 0 \\ 0 & 0 & d_3 & 0 \\ 0 & 0 & 0 & d_4 \end{bmatrix},$$

then $A = LDL^T$. **(b)** Generalize to arbitrary n and write a function `[d,e] = LTL(a,c)` that computes this factorization.

P7.3.5 Suppose we have the Cholesky factor G of an n-by-n symmetric positive definite matrix A. The solution to the linear system $(A + uu^T)x = b$ is given by

$$x = A^{-1}b - \left(\frac{u^T A^{-1} b}{1 + u^T A^{-1} u} \right) A^{-1}u,$$

where u and b are given n-by-1. Write a MATLAB script that computes this vector. Make effective use of the functions `LTriSol(T,b)` and `UTriSol(T,b)`.

P7.3.6 Suppose N is a positive integer and that `A` is a $3N$-by-2 array and `b` is a $3N$-by-1 array. Our goal is to solve the N, 3-by-2 least square problems

$$\min \| A(3k-2{:}3k, :)x - b(3k-2{:}3k) \|_2, \qquad k = 1{:}N.$$

Note that an individual 3-by-2 LS problem $\min \| Cx - d \|_2$ can be solved via the 2-by-2 normal equation system $C^T Cx = C^T d$ as follows:

- Form $M = C^T C$ and $z = C^T d$.
- Compute the Cholesky factorization of M: $M = LL^T$.
- Solve $Ly = z$ for y and $L^T x = y$ for x.

Complete the following function:

```
  function x = LS32(A,b)
% x = LS32(A,b)
%
% A is a 3N-by-2 array and b is a 3N-by-1 array for some integer N>0.
% Assume that A(3k-2:3k,:) has rank 2 for k=1:N.
%
% x is a 3N-by-1 array with the property that z = x(3k-2:3k) minimizes
% the 2-norm of A(3k-2:3k,:)z - b(3k-2:3k) for k=1:N.
```

`LS32` should have *no* loops. To do this you'll have to solve the N problems "in parallel." For example, the Cholesky factorization for a 2-by-2 matrix M is a 3-liner:

$$\ell_{11} = \sqrt{m_{11}}$$

$$\ell_{21} = m_{21}/\ell_{11}$$

$$\ell_{22} = \sqrt{m_{22} - \ell_{21}^2}$$

To vectorize this, get all N of the ℓ_{11}'s first, then get all N of the ℓ_{21}'s, and then get all N of the ℓ_{22}'s.

7.4 High-Performance Cholesky

We discuss two implementations of the Cholesky factorization that shed light on what it is like to design a linear equation solver that runs fast in an advanced computing environment. The block Cholesky algorithm shows how to make a linear algebra computation rich in matrix-matrix multiplication. This has the effect of reducing the amount of data motion. The shared memory implementation shows how the work involved in a matrix factorization can be subdivided into roughly equal parts that can be simultaneously computed by separate processors.

7.4.1 Level-3 Implementation

Level-3 operations involve a quadratic amount of data and a cubic amount of work. Matrix multiplication is the leading example of a level-3 operation. In the n-by-n, $C = AB$ case, there are $2n^2$ input values and $2n^3$ flops to perform. Unlike level-1 and level-2 operations, the ratio of work to data grows with problem size in the level-3 setting. This makes it possible to bury data motion overheads in a computation that is rich in level-3 operations. Algorithms with this property usually run very fast on advanced architectures.

As an exercise in "level-3 thinking" we show how the Cholesky computation can be arranged so that all but a small fraction of the arithmetic occurs in the context of matrix multiplication. The key is to partition the A and G matrices into blocks (submatrices) and to organize the calculations at that level. For simplicity, we develop a block version of `CholScalar`. Assume that $n = pm$ and partition A and G into p-by-p blocks as follows:

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & & \vdots \\ A_{m1} & \cdots & A_{mm} \end{bmatrix} \qquad G = \begin{bmatrix} G_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ G_{m1} & \cdots & G_{mm} \end{bmatrix}.$$

This means that we are regarding A and G as m-by-m matrices with p-by-p blocks. For example, here is a partitioning of a 12-by-12 symmetric positive definite matrix into 3-by-3 blocks:

$$A = \left[\begin{array}{rrr|rrr|rrr|rrr}
34 & 1 & 14 & 17 & 12 & 9 & 6 & 17 & 5 & 9 & 12 & 8 \\
1 & 38 & 10 & 11 & 10 & 9 & 17 & 11 & 8 & 7 & 16 & 10 \\
14 & 10 & 45 & 10 & 2 & 8 & 11 & 9 & 9 & 18 & 6 & 11 \\ \hline
17 & 11 & 10 & 43 & 6 & 16 & 17 & 6 & 6 & 9 & 7 & 8 \\
12 & 10 & 2 & 6 & 48 & 10 & 2 & 14 & 11 & 7 & 6 & 19 \\
9 & 9 & 8 & 16 & 10 & 40 & 4 & 9 & 17 & 12 & 14 & 15 \\ \hline
6 & 17 & 11 & 17 & 2 & 4 & 44 & 17 & 7 & 9 & 14 & 11 \\
17 & 11 & 9 & 6 & 14 & 9 & 17 & 38 & 14 & 4 & 6 & 15 \\
5 & 8 & 9 & 6 & 11 & 17 & 7 & 14 & 40 & 12 & 14 & 10 \\ \hline
9 & 7 & 18 & 9 & 7 & 12 & 9 & 4 & 12 & 30 & 8 & 2 \\
12 & 16 & 6 & 7 & 6 & 14 & 14 & 6 & 14 & 8 & 38 & 11 \\
8 & 10 & 11 & 8 & 19 & 15 & 11 & 15 & 10 & 2 & 11 & 35
\end{array}\right].$$

Note that the (i,j) block is the transpose of the (j,i) block. For example,

$$A_{23} = \begin{bmatrix} 17 & 6 & 6 \\ 2 & 14 & 11 \\ 4 & 9 & 17 \end{bmatrix} = \begin{bmatrix} 17 & 2 & 4 \\ 6 & 14 & 9 \\ 6 & 11 & 17 \end{bmatrix}^T = A_{32}^T.$$

Comparing (i,j) blocks in the equation $A = GG^T$ with $i \geq j$ gives

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

and so

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

Corresponding to the derivation of `CholScalar`, we obtain the following framework for computing the G_{ij}:

```
for i=1:m
   for j=1:i
      Compute S_ij
      if i<j
         G_ij is the solution of XG_jj^T = S_ij                    (7.3)
      else
         G_ii is the Cholesky factor of S_ii.
      end
   end
end
```

To implement this method we use cell arrays to represent both A and G as block matrices. The idea is to store matrix block (i,j) in cell (i,j). For this we use the `MakeBlock` function developed in §5.1.4 (see page 175) :

```
  function A = MakeBlock(A_scalar,p)
% A = MakeBlock(A_scalar,p)
% Represents and n-by-n matrix A_scalar as an (n/p)-by-(n/p) block matrix with
% p-by-p blocks. It is assumed that n is divisible by p.
```

If applied to the 4-by-4 block matrix above, then this function would set

$$\texttt{A}\{1,2\} = \begin{bmatrix} 17 & 12 & 9 \\ 11 & 10 & 9 \\ 10 & 2 & 8 \end{bmatrix}.$$

The conversion from this representation back to conventional matrix form is also required:

```
  function A = MakeScalar(A_block)
% A = MakeScalar(A_block)
% Represents the m-by-m block matrix A_block as an n-by-n matrix of scalars
% where each block is p-by-p and n=mp.
[m,m] = size(A_block);
[p,p] = size(A_block{1,1});
for i=1:m
   for j=1:m
      if ~isempty(A_block{i,j})
         A(1+(i-1)*p:i*p,1+(j-1)*p:j*p) = A_block{i,j};
      end
   end
end
```

With these functions the implementation of (7.3) can proceed:

```
  function G = CholBlock(A,p)
% G = CholBlock(A,p)
% Cholesky factorization of a symmetric and positive definite n-by-n matrix A.
% G is lower triangular so A = G*G' and p is the block size and must divide n.

% Represent A and G as m-by-m block matrices where m = n/p.
[n,n] = size(A);
m = n/p;
A = MakeBlock(A,p);
G = cell(m,m);
for i=1:m
   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} = (G{j,j}\S')';
      else
         G{i,i} = CholScalar(S);
      end
   end
end
% Convert G to a matrix of scalars.
G = MakeScalar(G);
```

This algorithm involves $n^3/3$ flops, the same number of flops as any of the §7.3 methods. The only flops that are *not* level-3 flops are those associated with the computation of the p-by-p Cholesky factors $G_{11},\ldots,G_{mm}$. It follows that the fraction of flops that are level-3 flops is given by

$$L_3 = \frac{(n^3/3) - (mp^3/3)}{n^3/3} = 1 - \frac{1}{m^2}.$$

A tacit assumption in all this is that the block size p is large enough 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. The script file `ShowCholBlock` benchmarks `CholBlock` on an $n = 192$ problem for various block sizes p. Here are the results with the unit block size time normalized to 1.000:

Block Size	Normalized Time
8	1.000
12	0.667
16	0.667
24	0.828
32	1.222
48	2.273
96	7.828

Notice that the optimum block size is around $\sqrt{n}$.

7.4.2 A Shared Memory Implementation

We now turn our attention to the implementation of the Cholesky factorization in a shared memory environment. This framework for parallel computation is introduced in §4.5. Assume at the start that $A \in \mathbb{R}^{n\times n}$ is housed in shared memory and that we have p processors to apply to the problem, $p << n$. Our goal is to write the node program for Proc(1),. . .,Proc(p).

We identify the computation of $G(j{:}n, j)$ as the jth task. Analogous to the development of `CholSax`, we compare jth columns in the equation $A = GG^T$, with

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

and obtain the key result

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

Note that $G(j{:}n, j) = s(j{:}n)/\sqrt{s_j}$. From this we conclude that the processor in charge of computing $G(j{:}n, j)$ must oversee the update

$$A(j{:}n, j) \leftarrow A(j{:}n, j) \; - \sum_{k=1}^{j-1} G(j{:}n, k)G(j, k) \tag{7.4}$$

and the scaling

$$A(j{:}n, j) \leftarrow A(j{:}n, j)/\sqrt{A(j, j)}. \tag{7.5}$$

This requires about $2n(n-j)$ flops. Since the cost of computing column j is a decreasing function of j, it does not makes sense for Proc(1) to handle tasks 1 to n/p, Proc(2) to handle tasks $1+n/p$ through $2n/p$, etc. To achieve some measure of flop load balancing, we assign to Proc(μ) tasks $\mu{:}p{:}n$. This amounts to "dealing" out tasks as you would a deck of n cards. Here is a who-does-what table for the case $n = 22$, $p = 4$:

Processor	Tasks					
1	1	5	9	13	17	21
2	2	6	10	14	18	22
3	3	7	11	15	19	
4	4	8	12	16	20	

Thus Proc(μ) is charged with the production of $G(:\mu{:}p{:}n)$.

Suppose $G(k{:}n, k)$ has just been computed and is contained in `A(k:n,k)` in shared memory. Proc(μ) can then carry out the updates

```
for j=k+1:n
   if j is one of the integers in mu:p:n
      A(k:n,j) = A(k:n,j) - A(k:n,k)A(j,k)
   end
end
```

In other words, for each of the remaining tasks that it must perform (i.e., the indices in `mu:p:n` that are greater than k), Proc(μ) incorporates the kth term in the corresponding update summation (7.4). Assuming that `A` is in shared memory and that all other variables are local, here is the node program for Proc(μ):

```
MyCols = mu:p:n;
for k=1:n
   if any(MyCols==k)
      % My turn to generate a G-column
      A(k:n,k) = A(k:n,k)/sqrt(A(k,k));
   end
   barrier
   % Update columns whose indices are greater than k and in mu:p:n.
   for j=k+1:n
      if any(MyCols==j)
         A(k:n,j) = A(k:n,j)-A(k:n,k)*A(j,k);
      end
   end
   barrier
end
```

The first `if` is executed by only one processor, the processor that "owns" column `k`. The first `barrier` is necessary to ensure that no processor uses the new G-column until it is safely stored in shared memory. Once that happens, the p processors share in the update of columns `k+1` through `n`. The second `barrier` is necessary to guarantee that column `k` of `A` is ready at the start of the next step.

See Figure 7.2 on the next page for clarification about what goes on at each step in an $n = 22$, $p = 4$ example. The boxed integers indicate the index of the G-column being produced. The other indices name columns that the processor must update.

k	Proc(1)	Proc(2)	Proc(3)	Proc(4)
1	$\boxed{1}$,5,9,13,17,21	2,6,10,14,18,22	3,7,11,15,19	4,8,12,16,20
2	5,9,13,17,21	$\boxed{2}$,6,10,14,18	3,7,11,15,19	4,8,12,16,20
3	5,9,13,17,21	6,10,14,18,22	$\boxed{3}$,7,11,15,19	4,8,12,16,20
4	5,9,13,17,21	6,10,14,18,22	7,11,15,19	$\boxed{4}$,8,12,16,20
5	$\boxed{5}$, 9,13,17,21	6,10,14,18,22	7,11,15,19	8,12,16,20
6	9,13,17,21	$\boxed{6}$,10,14,18,22	7,11,15,19	8,12,16,20
7	9,13,17,21	10,14,18,22	$\boxed{7}$,11,15,19	8,12,16,20
8	9,13,17,21	10,14,18,22	11,15,19	$\boxed{8}$,12,16,20
9	$\boxed{9}$,13,17,21	10,14,18	11,15,19	12,16,20
10	13,17,21	$\boxed{10}$,14,18,22	11,15,19	12,16,20
11	13,17,21	14,18,22	$\boxed{11}$,15,19	12,16,20
12	13,17,21	14,18,22	15,19	$\boxed{12}$,16,20
13	$\boxed{13}$,17,21	14,18,22	15,19	16,20
14	17,21	$\boxed{14}$,18,22	15,19	16,20
15	17,21	18,22	$\boxed{15}$,19	16,20
16	17,21	18,22	19	$\boxed{16}$,20
17	$\boxed{17}$,21	18,22	19	20
18	21	$\boxed{18}$,22	19	20
19	21	22	$\boxed{19}$	20
20	21	22		$\boxed{20}$
21	$\boxed{21}$	22		
22		$\boxed{22}$		

FIGURE 7.2 Distribution of tasks ($n = 22,\ p = 4$)

Problems

P7.4.1 Generalize `CholBlock` so that it can handle the case when the dimension n is not a multiple of the block dimension p. (This means that there may be some less-than-full-size blocks on the right and bottom edges of A.)

P7.4.2 Write a function `f = FlopBalance(n,p)` that returns a p-vector f with the property that f_μ is the number of flops Proc(μ) must perform in shared memory implementation developed in the text.

P7.4.3 Assume that $n = mp$. Rewrite the node program with the assumption that Proc(μ) computes

$$G(:, (\mu - 1)m + 1{:}\mu m).$$

As in the previous problem, analyze the distribution of flops.

M-Files and References

Script Files

`ShowLSFit`	Fits a line to the square root function.
`ShowLSq`	Sensitivity of LS solutions.
`ShowSparseLS`	Examines \ LS solving with sparse arrays.
`ShowQR`	Illustrates QRrot.
`ShowBVP`	Solves a two-point boundary value problem.
`CholBench`	Benchmarks various Cholesky implementations.
`CholErr`	Sensitivity of symmetric positive definite systems.
`ShowCholBlock`	Explores block size selection in CholBlock.

Function Files

`Rotate`	Computes a rotation to zero bottom of 2-vector.
`QRrot`	QR factorization via rotations.
`LSq`	Least squares solution via QR factorization.
`CholTrid`	Cholesky factorization (tridiagonal version).
`CholTridSol`	Solves a factored tridiagonal system.
`CholScalar`	Cholesky factorization (scalar version).
`CholDot`	Cholesky factorization (dot product version).
`CholSax`	Cholesky factorization (saxpy version).
`CholGax`	Cholesky factorization (gaxpy version).
`CholRecur`	Cholesky factorization (recursive version).
`CholBlock`	Cholesky factorization (block version).

References

Å. Björck (1996). *Numerical Methods for Least Squares Problems,* SIAM Publications, Philadelphia, PA.

J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst (1990). *Solving Linear Systems on Vector and Shared Memory Computers,* SIAM Publications, Philadelphia, PA.

G.H. Golub and C.F. Van Loan (1996). *Matrix Computations, Third Edition*, Johns Hopkins University Press, Baltimore, MD.

C.L. Lawson and R.J. Hanson (1996). *Solving Least Squares Problems*, SIAM Publications, Philadelphia, PA.