```
        x[i]=sum/p[i];
    }
}
```

A typical use of `choldc` and `cholsl` is in the inversion of covariance matrices describing the fit of data to a model; see, e.g., §15.6. In this, and many other applications, one often needs $\mathbf{L}^{-1}$. The lower triangle of this matrix can be efficiently found from the output of `choldc`:

```
    for (i=1;i<=n;i++) {
        a[i][i]=1.0/p[i];
        for (j=i+1;j<=n;j++) {
            sum=0.0;
            for (k=i;k<j;k++) sum -= a[j][k]*a[k][i];
            a[j][i]=sum/p[j];
        }
    }
```

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/1.

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), §4.9.2.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.5.

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.2.

## 2.10 QR Decomposition

There is another matrix factorization that is sometimes very useful, the so-called $QR$ *decomposition*,

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{2.10.1}$$

Here $\mathbf{R}$ is upper triangular, while $\mathbf{Q}$ is orthogonal, that is,

$$\mathbf{Q}^T \cdot \mathbf{Q} = \mathbf{1} \tag{2.10.2}$$

where $\mathbf{Q}^T$ is the transpose matrix of $\mathbf{Q}$. Although the decomposition exists for a general rectangular matrix, we shall restrict our treatment to the case when all the matrices are square, with dimensions $N \times N$.

Like the other matrix factorizations we have met ($LU$, SVD, Cholesky), $QR$ decomposition can be used to solve systems of linear equations. To solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.10.3}$$

first form $\mathbf{Q}^T \cdot \mathbf{b}$ and then solve

$$\mathbf{R} \cdot \mathbf{x} = \mathbf{Q}^T \cdot \mathbf{b} \tag{2.10.4}$$

by backsubstitution. Since $QR$ decomposition involves about twice as many operations as $LU$ decomposition, it is not used for typical systems of linear equations. However, we will meet special cases where $QR$ is the method of choice.

The standard algorithm for the $QR$ decomposition involves successive Householder transformations (to be discussed later in §11.2). We write a Householder matrix in the form $\mathbf{1} - \mathbf{u} \otimes \mathbf{u}/c$ where $c = \frac{1}{2}\mathbf{u} \cdot \mathbf{u}$. An appropriate Householder matrix applied to a given matrix can zero all elements in a column of the matrix situated below a chosen element. Thus we arrange for the first Householder matrix $\mathbf{Q}_1$ to zero all elements in the first column of $\mathbf{A}$ below the first element. Similarly $\mathbf{Q}_2$ zeroes all elements in the second column below the second element, and so on up to $\mathbf{Q}_{n-1}$. Thus

$$\mathbf{R} = \mathbf{Q}_{n-1} \cdots \mathbf{Q}_1 \cdot \mathbf{A} \tag{2.10.5}$$

Since the Householder matrices are orthogonal,

$$\mathbf{Q} = (\mathbf{Q}_{n-1} \cdots \mathbf{Q}_1)^{-1} = \mathbf{Q}_1 \cdots \mathbf{Q}_{n-1} \tag{2.10.6}$$

In most applications we don't need to form $\mathbf{Q}$ explicitly; we instead store it in the factored form (2.10.6). Pivoting is not usually necessary unless the matrix $\mathbf{A}$ is very close to singular. A general $QR$ algorithm for rectangular matrices including pivoting is given in [1]. For square matrices, an implementation is the following:

```
#include <math.h>
#include "nrutil.h"

void qrdcmp(float **a, int n, float *c, float *d, int *sing)
```

Constructs the $QR$ decomposition of a[1..n][1..n]. The upper triangular matrix $\mathbf{R}$ is returned in the upper triangle of a, except for the diagonal elements of $\mathbf{R}$ which are returned in d[1..n]. The orthogonal matrix $\mathbf{Q}$ is represented as a product of $n-1$ Householder matrices $\mathbf{Q}_1 \ldots \mathbf{Q}_{n-1}$, where $\mathbf{Q}_j = \mathbf{1} - \mathbf{u}_j \otimes \mathbf{u}_j/c_j$. The $i$th component of $\mathbf{u}_j$ is zero for $i = 1, \ldots, j-1$ while the nonzero components are returned in a[i][j] for $i = j, \ldots, n$. sing returns as true (1) if singularity is encountered during the decomposition, but the decomposition is still completed in this case; otherwise it returns false (0).

```
{
    int i,j,k;
    float scale,sigma,sum,tau;

    *sing=0;
    for (k=1;k<n;k++) {
        scale=0.0;
        for (i=k;i<=n;i++) scale=FMAX(scale,fabs(a[i][k]));
        if (scale == 0.0) {                     Singular case.
            *sing=1;
            c[k]=d[k]=0.0;
        } else {                                Form Q_k and Q_k · A.
            for (i=k;i<=n;i++) a[i][k] /= scale;
            for (sum=0.0,i=k;i<=n;i++) sum += SQR(a[i][k]);
            sigma=SIGN(sqrt(sum),a[k][k]);
            a[k][k] += sigma;
            c[k]=sigma*a[k][k];
            d[k] = -scale*sigma;
            for (j=k+1;j<=n;j++) {
                for (sum=0.0,i=k;i<=n;i++) sum += a[i][k]*a[i][j];
                tau=sum/c[k];
                for (i=k;i<=n;i++) a[i][j] -= tau*a[i][k];
            }
        }
    }
    d[n]=a[n][n];
    if (d[n] == 0.0) *sing=1;
}
```

The next routine, qrsolv, is used to solve linear systems. In many applications only the part (2.10.4) of the algorithm is needed, so we separate it off into its own routine rsolv.

```
void qrsolv(float **a, int n, float c[], float d[], float b[])
Solves the set of n linear equations A · x = b. a[1..n][1..n], c[1..n], and d[1..n] are
input as the output of the routine qrdcmp and are not modified. b[1..n] is input as the
right-hand side vector, and is overwritten with the solution vector on output.
{
    void rsolv(float **a, int n, float d[], float b[]);
    int i,j;
    float sum,tau;

    for (j=1;j<n;j++) {          Form Q^T · b.
        for (sum=0.0,i=j;i<=n;i++) sum += a[i][j]*b[i];
        tau=sum/c[j];
        for (i=j;i<=n;i++) b[i] -= tau*a[i][j];
    }
    rsolv(a,n,d,b);              Solve R · x = Q^T · b.
}
```

```
void rsolv(float **a, int n, float d[], float b[])
Solves the set of n linear equations R · x = b, where R is an upper triangular matrix stored in
a and d. a[1..n][1..n] and d[1..n] are input as the output of the routine qrdcmp and
are not modified. b[1..n] is input as the right-hand side vector, and is overwritten with the
solution vector on output.
{
    int i,j;
    float sum;

    b[n] /= d[n];
    for (i=n-1;i>=1;i--) {
        for (sum=0.0,j=i+1;j<=n;j++) sum += a[i][j]*b[j];
        b[i]=(b[i]-sum)/d[i];
    }
}
```

See [2] for details on how to use $QR$ decomposition for constructing orthogonal bases, and for solving least-squares problems. (We prefer to use SVD, §2.6, for these purposes, because of its greater diagnostic capability in pathological cases.)

## Updating a QR decomposition

Some numerical algorithms involve solving a succession of linear systems each of which differs only slightly from its predecessor. Instead of doing $O(N^3)$ operations each time to solve the equations from scratch, one can often update a matrix factorization in $O(N^2)$ operations and use the new factorization to solve the next set of linear equations. The $LU$ decomposition is complicated to update because of pivoting. However, $QR$ turns out to be quite simple for a very common kind of update,

$$\mathbf{A} \rightarrow \mathbf{A} + \mathbf{s} \otimes \mathbf{t} \tag{2.10.7}$$

(compare equation 2.7.1). In practice it is more convenient to work with the equivalent form

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \quad \rightarrow \quad \mathbf{A}' = \mathbf{Q}' \cdot \mathbf{R}' = \mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v}) \tag{2.10.8}$$

One can go back and forth between equations (2.10.7) and (2.10.8) using the fact that $\mathbf{Q}$ is orthogonal, giving

$$\mathbf{t} = \mathbf{v} \quad \text{and either} \quad \mathbf{s} = \mathbf{Q} \cdot \mathbf{u} \quad \text{or} \quad \mathbf{u} = \mathbf{Q}^T \cdot \mathbf{s} \tag{2.10.9}$$

The algorithm [2] has two phases. In the first we apply $N-1$ Jacobi rotations (§11.1) to reduce $\mathbf{R} + \mathbf{u} \otimes \mathbf{v}$ to upper Hessenberg form. Another $N-1$ Jacobi rotations transform this upper Hessenberg matrix to the new upper triangular matrix $\mathbf{R}'$. The matrix $\mathbf{Q}'$ is simply the product of $\mathbf{Q}$ with the $2(N-1)$ Jacobi rotations. In applications we usually want $\mathbf{Q}^T$, and the algorithm can easily be rearranged to work with this matrix instead of with $\mathbf{Q}$.

```
#include <math.h>
#include "nrutil.h"

void qrupdt(float **r, float **qt, int n, float u[], float v[])
```

Given the $QR$ decomposition of some n × n matrix, calculates the $QR$ decomposition of the matrix $\mathbf{Q}\cdot(\mathbf{R}+\mathbf{u}\otimes\mathbf{v})$. The quantities are dimensioned as r[1..n][1..n], qt[1..n][1..n], u[1..n], and v[1..n]. Note that $\mathbf{Q}^T$ is input and returned in qt.

```
{
    void rotate(float **r, float **qt, int n, int i, float a, float b);
    int i,j,k;

    for (k=n;k>=1;k--) {                    Find largest k such that u[k] ≠ 0.
        if (u[k]) break;
    }
    if (k < 1) k=1;
    for (i=k-1;i>=1;i--) {                  Transform R + u ⊗ v to upper Hessenberg.
        rotate(r,qt,n,i,u[i],-u[i+1]);
        if (u[i] == 0.0) u[i]=fabs(u[i+1]);
        else if (fabs(u[i]) > fabs(u[i+1]))
            u[i]=fabs(u[i])*sqrt(1.0+SQR(u[i+1]/u[i]));
        else u[i]=fabs(u[i+1])*sqrt(1.0+SQR(u[i]/u[i+1]));
    }
    for (j=1;j<=n;j++) r[1][j] += u[1]*v[j];
    for (i=1;i<k;i++)                       Transform upper Hessenberg matrix to upper tri-
        rotate(r,qt,n,i,r[i][i],-r[i+1][i]);    angular.
}
```

```
#include <math.h>
#include "nrutil.h"

void rotate(float **r, float **qt, int n, int i, float a, float b)
```

Given matrices r[1..n][1..n] and qt[1..n][1..n], carry out a Jacobi rotation on rows i and $i+1$ of each matrix. a and b are the parameters of the rotation: $\cos\theta = a/\sqrt{a^2+b^2}$, $\sin\theta = b/\sqrt{a^2+b^2}$.

```
{
    int j;
    float c,fact,s,w,y;

    if (a == 0.0) {                 Avoid unnecessary overflow or underflow.
        c=0.0;
        s=(b >= 0.0 ? 1.0 : -1.0);
    } else if (fabs(a) > fabs(b)) {
        fact=b/a;
        c=SIGN(1.0/sqrt(1.0+(fact*fact)),a);
        s=fact*c;
    } else {
        fact=a/b;
        s=SIGN(1.0/sqrt(1.0+(fact*fact)),b);
        c=fact*s;
    }
    for (j=i;j<=n;j++) {            Premultiply r by Jacobi rotation.
        y=r[i][j];
        w=r[i+1][j];
        r[i][j]=c*y-s*w;
        r[i+1][j]=s*y+c*w;
    }
    for (j=1;j<=n;j++) {            Premultiply qt by Jacobi rotation.
        y=qt[i][j];
        w=qt[i+1][j];
        qt[i][j]=c*y-s*w;
        qt[i+1][j]=s*y+c*w;
    }
}
```

We will make use of $QR$ decomposition, and its updating, in §9.7.

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/8. [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §§5.2, 5.3, 12.6. [2]



## 2.11 Is Matrix Inversion an $N^3$ Process?

We close this chapter with a little entertainment, a bit of algorithmic prestidigitation which probes more deeply into the subject of matrix inversion. We start with a seemingly simple question:

How many individual multiplications does it take to perform the matrix multiplication of two $2 \times 2$ matrices,

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \tag{2.11.1}$$

Eight, right? Here they are written explicitly:

$$\begin{aligned} c_{11} &= a_{11} \times b_{11} + a_{12} \times b_{21} \\ c_{12} &= a_{11} \times b_{12} + a_{12} \times b_{22} \\ c_{21} &= a_{21} \times b_{11} + a_{22} \times b_{21} \\ c_{22} &= a_{21} \times b_{12} + a_{22} \times b_{22} \end{aligned} \tag{2.11.2}$$

Do you think that one can write formulas for the $c$'s that involve only *seven* multiplications? (Try it yourself, before reading on.)

Such a set of formulas was, in fact, discovered by Strassen [1]. The formulas are:

$$\begin{aligned} Q_1 &\equiv (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\ Q_2 &\equiv (a_{21} + a_{22}) \times b_{11} \\ Q_3 &\equiv a_{11} \times (b_{12} - b_{22}) \\ Q_4 &\equiv a_{22} \times (-b_{11} + b_{21}) \\ Q_5 &\equiv (a_{11} + a_{12}) \times b_{22} \\ Q_6 &\equiv (-a_{11} + a_{21}) \times (b_{11} + b_{12}) \\ Q_7 &\equiv (a_{12} - a_{22}) \times (b_{21} + b_{22}) \end{aligned} \tag{2.11.3}$$