is equivalent to the $2n \times 2n$ real problem

$$\begin{bmatrix} \mathbf{A} & -\mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \lambda \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \tag{11.4.2}$$

Note that the $2n \times 2n$ matrix in (11.4.2) is symmetric: $\mathbf{A}^T = \mathbf{A}$ and $\mathbf{B}^T = -\mathbf{B}$ if $\mathbf{C}$ is Hermitian.

Corresponding to a given eigenvalue λ, the vector

$$\begin{bmatrix} -\mathbf{v} \\ \mathbf{u} \end{bmatrix} \tag{11.4.3}$$

is also an eigenvector, as you can verify by writing out the two matrix equations implied by (11.4.2). Thus if $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues of $\mathbf{C}$, then the $2n$ eigenvalues of the augmented problem (11.4.2) are $\lambda_1, \lambda_1, \lambda_2, \lambda_2, \dots, \lambda_n, \lambda_n$; each, in other words, is repeated twice. The eigenvectors are pairs of the form $\mathbf{u} + i\mathbf{v}$ and $i(\mathbf{u} + i\mathbf{v})$; that is, they are the same up to an inessential phase. Thus we solve the augmented problem (11.4.2), and choose one eigenvalue and eigenvector from each pair. These give the eigenvalues and eigenvectors of the original matrix $\mathbf{C}$.

Working with the augmented matrix requires a factor of 2 more storage than the original complex matrix. In principle, a complex algorithm is also a factor of 2 more efficient in computer time than is the solution of the augmented problem.

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). [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

11.5 Reduction of a General Matrix to Hessenberg Form

The algorithms for symmetric matrices, given in the preceding sections, are highly satisfactory in practice. By contrast, it is impossible to design equally satisfactory algorithms for the nonsymmetric case. There are two reasons for this. First, the eigenvalues of a nonsymmetric matrix can be very sensitive to small changes in the matrix elements. Second, the matrix itself can be defective, so that there is no complete set of eigenvectors. We emphasize that these difficulties are intrinsic properties of certain nonsymmetric matrices, and no numerical procedure can "cure" them. The best we can hope for are procedures that don't exacerbate such problems.

The presence of rounding error can only make the situation worse. With finite-precision arithmetic, one cannot even design a foolproof algorithm to determine whether a given matrix is defective or not. Thus current algorithms generally *try* to find a *complete* set of eigenvectors, and rely on the user to inspect the results. If any eigenvectors are almost parallel, the matrix is probably defective.

Apart from referring you to the literature, and to the collected routines in [1,2], we are going to sidestep the problem of eigenvectors, giving algorithms for eigenvalues only. If you require just a few eigenvectors, you can read §11.7 and consider finding them by inverse iteration. We consider the problem of finding *all* eigenvectors of a nonsymmetric matrix as lying beyond the scope of this book.

Balancing

The sensitivity of eigenvalues to rounding errors during the execution of some algorithms can be reduced by the procedure of *balancing*. The errors in the eigensystem found by a numerical procedure are generally proportional to the Euclidean norm of the matrix, that is, to the square root of the sum of the squares of the elements. The idea of balancing is to use similarity transformations to make corresponding rows and columns of the matrix have comparable norms, thus reducing the overall norm of the matrix while leaving the eigenvalues unchanged. A symmetric matrix is already balanced.

Balancing is a procedure with of order N^2 operations. Thus, the time taken by the procedure balanc, given below, should never be more than a few percent of the total time required to find the eigenvalues. It is therefore recommended that you *always* balance nonsymmetric matrices. It never hurts, and it can substantially improve the accuracy of the eigenvalues computed for a badly balanced matrix.

The actual algorithm used is due to Osborne, as discussed in [1]. It consists of a sequence of similarity transformations by diagonal matrices **D**. To avoid introducing rounding errors during the balancing process, the elements of **D** are restricted to be exact powers of the radix base employed for floating-point arithmetic (i.e., 2 for most machines, but 16 for IBM mainframe architectures). The output is a matrix that is balanced in the norm given by summing the absolute magnitudes of the matrix elements. This is more efficient than using the Euclidean norm, and equally effective: A large reduction in one norm implies a large reduction in the other.

Note that if the off-diagonal elements of any row or column of a matrix are all zero, then the diagonal element is an eigenvalue. If the eigenvalue happens to be ill-conditioned (sensitive to small changes in the matrix elements), it will have relatively large errors when determined by the routine hqr (§11.6). Had we merely inspected the matrix beforehand, we could have determined the isolated eigenvalue exactly and then deleted the corresponding row and column from the matrix. You should consider whether such a pre-inspection might be useful in your application. (For symmetric matrices, the routines we gave will determine isolated eigenvalues accurately in all cases.)

The routine balanc does not keep track of the accumulated similarity transformation of the original matrix, since we will only be concerned with finding eigenvalues of nonsymmetric matrices, not eigenvectors. Consult [1-3] if you want to keep track of the transformation.

```
#include <math.h>
#define RADIX 2.0

void balanc(float **a, int n)
```
Given a matrix a[1..n][1..n], this routine replaces it by a balanced matrix with identical eigenvalues. A symmetric matrix is already balanced and is unaffected by this procedure. The parameter RADIX should be the machine's floating-point radix.

```
{
    int last,j,i;
    float s,r,g,f,c,sqrdx;

    sqrdx=RADIX*RADIX;
    last=0;
    while (last == 0) {
        last=1;
        for (i=1;i<=n;i++) {            Calculate row and column norms.
            r=c=0.0;
            for (j=1;j<=n;j++)
                if (j != i) {
                    c += fabs(a[j][i]);
                    r += fabs(a[i][j]);
                }
            if (c && r) {               If both are nonzero,
                g=r/RADIX;
                f=1.0;
                s=c+r;
                while (c<g) {           find the integer power of the machine radix that
                    f *= RADIX;             comes closest to balancing the matrix.
                    c *= sqrdx;
                }
                g=r*RADIX;
                while (c>g) {
                    f /= RADIX;
                    c /= sqrdx;
                }
                if ((c+r)/f < 0.95*s) {
                    last=0;
                    g=1.0/f;
                    for (j=1;j<=n;j++) a[i][j] *= g;      Apply similarity transforma-
                    for (j=1;j<=n;j++) a[j][i] *= f;          tion.
                }
            }
        }
    }
}
```

Reduction to Hessenberg Form

The strategy for finding the eigensystem of a general matrix parallels that of the symmetric case. First we reduce the matrix to a simpler form, and then we perform an iterative procedure on the simplified matrix. The simpler structure we use here is called *Hessenberg* form. An *upper Hessenberg* matrix has zeros everywhere below the diagonal except for the first subdiagonal row. For example, in the 6×6 case, the nonzero elements are:

$$
\begin{bmatrix}
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
 & \times & \times & \times & \times & \times \\
 & & \times & \times & \times & \times \\
 & & & \times & \times & \times \\
 & & & & \times & \times
\end{bmatrix}
$$

By now you should be able to tell at a glance that such a structure can be achieved by a sequence of Householder transformations, each one zeroing the required elements in a column of the matrix. Householder reduction to Hessenberg form is in fact an accepted technique. An alternative, however, is a procedure analogous to Gaussian elimination with pivoting. We will use this elimination procedure since it is about a factor of 2 more efficient than the Householder method, and also since we want to teach you the method. It is possible to construct matrices for which the Householder reduction, being orthogonal, is stable and elimination is not, but such matrices are extremely rare in practice.

Straight Gaussian elimination is not a similarity transformation of the matrix. Accordingly, the actual elimination procedure used is slightly different. Before the rth stage, the original matrix $\mathbf{A} \equiv \mathbf{A}_1$ has become $\mathbf{A}_r$, which is upper Hessenberg in its first $r-1$ rows and columns. The rth stage then consists of the following sequence of operations:

- Find the element of maximum magnitude in the rth column below the diagonal. If it is zero, skip the next two "bullets" and the stage is done. Otherwise, suppose the maximum element was in row r'.
- Interchange rows r' and $r+1$. This is the pivoting procedure. To make the permutation a similarity transformation, also interchange columns r' and $r+1$.
- For $i = r+2, r+3, \ldots, N$, compute the multiplier

$$n_{i,r+1} \equiv \frac{a_{ir}}{a_{r+1,r}}$$

Subtract $n_{i,r+1}$ times row $r+1$ from row i. To make the elimination a similarity transformation, also *add* $n_{i,r+1}$ times column i to column $r+1$.

A total of $N-2$ such stages are required.

When the magnitudes of the matrix elements vary over many orders, you should try to rearrange the matrix so that the largest elements are in the top left-hand corner. This reduces the roundoff error, since the reduction proceeds from left to right.

Since we are concerned only with eigenvalues, the routine `elmhes` does not keep track of the accumulated similarity transformation. The operation count is about $5N^3/6$ for large N.

```
#include <math.h>
#define SWAP(g,h) {y=(g);(g)=(h);(h)=y;}

void elmhes(float **a, int n)
```

Reduction to Hessenberg form by the elimination method. The real, nonsymmetric matrix `a[1..n][1..n]` is replaced by an upper Hessenberg matrix with identical eigenvalues. Recommended, but not required, is that this routine be preceded by `balanc`. On output, the Hessenberg matrix is in elements `a[i][j]` with `i` $\leq$ `j+1`. Elements with `i` $>$ `j+1` are to be thought of as zero, but are returned with random values.

```
{
    int m,j,i;
    float y,x;

    for (m=2;m<n;m++) {                 m is called r + 1 in the text.
        x=0.0;
        i=m;
        for (j=m;j<=n;j++) {            Find the pivot.
            if (fabs(a[j][m-1]) > fabs(x)) {
                x=a[j][m-1];
                i=j;
```

```
            }
        }
        if (i != m) {                   Interchange rows and columns.
            for (j=m-1;j<=n;j++) SWAP(a[i][j],a[m][j])
            for (j=1;j<=n;j++) SWAP(a[j][i],a[j][m])
        }
        if (x) {                        Carry out the elimination.
            for (i=m+1;i<=n;i++) {
                if ((y=a[i][m-1]) != 0.0) {
                    y /= x;
                    a[i][m-1]=y;
                    for (j=m;j<=n;j++)
                        a[i][j] -= y*a[m][j];
                    for (j=1;j<=n;j++)
                        a[j][m] += y*a[j][i];
                }
            }
        }
    }
}
```

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). [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.5.4. [3]

11.6 The QR Algorithm for Real Hessenberg Matrices

Recall the following relations for the QR algorithm with shifts:

$$\mathbf{Q}_s \cdot (\mathbf{A}_s - k_s \mathbf{1}) = \mathbf{R}_s \tag{11.6.1}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{R}$ is upper triangular, and

$$\begin{aligned} \mathbf{A}_{s+1} &= \mathbf{R}_s \cdot \mathbf{Q}_s^T + k_s \mathbf{1} \\ &= \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \end{aligned} \tag{11.6.2}$$

The QR transformation preserves the upper Hessenberg form of the original matrix $\mathbf{A} \equiv \mathbf{A}_1$, and the workload on such a matrix is $O(n^2)$ per iteration as opposed to $O(n^3)$ on a general matrix. As $s \to \infty$, $\mathbf{A}_s$ converges to a form where the eigenvalues are either isolated on the diagonal or are eigenvalues of a 2×2 submatrix on the diagonal.

As we pointed out in §11.3, shifting is essential for rapid convergence. A key difference here is that a nonsymmetric real matrix can have complex eigenvalues.