2.6 Singular Value Decomposition

There exists a very powerful set of techniques for dealing with sets of equations or matrices that are either singular or else numerically very close to singular. In many cases where Gaussian elimination and LU decomposition fail to give satisfactory results, this set of techniques, known as *singular value decomposition*, or *SVD*, will diagnose for you precisely what the problem is. In some cases, SVD will not only diagnose the problem, it will also solve it, in the sense of giving you a useful numerical answer, although, as we shall see, not necessarily "the" answer that you thought you should get.

SVD is also the method of choice for solving most *linear least-squares* problems. We will outline the relevant theory in this section, but defer detailed discussion of the use of SVD in this application to Chapter 15, whose subject is the parametric modeling of data.

SVD methods are based on the following theorem of linear algebra, whose proof is beyond our scope: Any $M \times N$ matrix **A** whose number of rows M is greater than or equal to its number of columns N, can be written as the product of an $M \times N$ column-orthogonal matrix **U**, an $N \times N$ diagonal matrix **W** with positive or zero elements (the *singular values*), and the transpose of an $N \times N$ orthogonal matrix **V**. The various shapes of these matrices will be made clearer by the following tableau:

$$\left(\begin{array}{c} \\ \\ \mathbf{A} \\ \\ \\ \end{array} \right) = \left(\begin{array}{c} \\ \\ \mathbf{U} \\ \\ \\ \end{array} \right) \cdot \begin{pmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_N \end{pmatrix} \cdot \left(\begin{array}{c} \\ \mathbf{V}^T \\ \\ \end{array} \right) \tag{2.6.1}$$

The matrices **U** and **V** are each orthogonal in the sense that their columns are orthonormal,

$$\sum_{i=1}^{M} U_{ik} U_{in} = \delta_{kn} \qquad \begin{array}{c} 1 \le k \le N \\ 1 \le n \le N \end{array} \tag{2.6.2}$$

$$\sum_{j=1}^{N} V_{jk} V_{jn} = \delta_{kn} \qquad \begin{array}{c} 1 \le k \le N \\ 1 \le n \le N \end{array} \tag{2.6.3}$$

or as a tableau,

$$\begin{pmatrix} & & \\ & \mathbf{U}^T & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \\ \mathbf{U} \\ \\ \\ \end{pmatrix} = \begin{pmatrix} & & \\ & \mathbf{V}^T & \\ & & \end{pmatrix} \cdot \begin{pmatrix} & & \\ & \mathbf{V} & \\ & & \end{pmatrix} = \begin{pmatrix} & & \\ & \mathbf{1} & \\ & & \end{pmatrix} \tag{2.6.4}$$

Since **V** is square, it is also row-orthonormal, $\mathbf{V} \cdot \mathbf{V}^T = 1$.

The SVD decomposition can also be carried out when $M < N$. In this case the singular values w_j for $j = M+1, \ldots, N$ are all zero, and the corresponding columns of **U** are also zero. Equation (2.6.2) then holds only for $k, n \le M$.

The decomposition (2.6.1) can always be done, no matter how singular the matrix is, and it is "almost" unique. That is to say, it is unique up to (i) making the same permutation of the columns of **U**, elements of **W**, and columns of **V** (or rows of $\mathbf{V}^T$), or (ii) forming linear combinations of any columns of **U** and **V** whose corresponding elements of **W** happen to be exactly equal. An important consequence of the permutation freedom is that for the case $M < N$, a numerical algorithm for the decomposition need not return zero w_j's for $j = M+1, \ldots, N$; the $N - M$ zero singular values can be scattered among all positions $j = 1, 2, \ldots, N$.

At the end of this section, we give a routine, `svdcmp`, that performs SVD on an arbitrary matrix **A**, replacing it by **U** (they are the same shape) and giving back **W** and **V** separately. The routine `svdcmp` is based on a routine by Forsythe et al. [1], which is in turn based on the original routine of Golub and Reinsch, found, in various forms, in [2-4] and elsewhere. These references include extensive discussion of the algorithm used. As much as we dislike the use of black-box routines, we are going to ask you to accept this one, since it would take us too far afield to cover its necessary background material here. Suffice it to say that the algorithm is very stable, and that it is very unusual for it ever to misbehave. Most of the concepts that enter the algorithm (Householder reduction to bidiagonal form, diagonalization by QR procedure with shifts) will be discussed further in Chapter 11.

If you are as suspicious of black boxes as we are, you will want to verify yourself that `svdcmp` does what we say it does. That is very easy to do: Generate an arbitrary matrix **A**, call the routine, and then verify by matrix multiplication that (2.6.1) and (2.6.4) are satisfied. Since these two equations are the only defining requirements for SVD, this procedure is (for the chosen **A**) a complete end-to-end check.

Now let us find out what SVD is good for.

SVD of a Square Matrix

If the matrix **A** is square, $N \times N$ say, then **U**, **V**, and **W** are all square matrices of the same size. Their inverses are also trivial to compute: **U** and **V** are orthogonal, so their inverses are equal to their transposes; **W** is diagonal, so its inverse is the diagonal matrix whose elements are the reciprocals of the elements w_j. From (2.6.1) it now follows immediately that the inverse of **A** is

$$\mathbf{A}^{-1} = \mathbf{V} \cdot [\text{diag}\,(1/w_j)] \cdot \mathbf{U}^T \tag{2.6.5}$$

The only thing that can go wrong with this construction is for one of the w_j's to be zero, or (numerically) for it to be so small that its value is dominated by roundoff error and therefore unknowable. If more than one of the w_j's have this problem, then the matrix is even more singular. So, first of all, SVD gives you a clear diagnosis of the situation.

Formally, the *condition number* of a matrix is defined as the ratio of the largest (in magnitude) of the w_j's to the smallest of the w_j's. A matrix is singular if its condition number is infinite, and it is *ill-conditioned* if its condition number is too large, that is, if its reciprocal approaches the machine's floating-point precision (for example, less than 10^{-6} for single precision or 10^{-12} for double).

For singular matrices, the concepts of *nullspace* and *range* are important. Consider the familiar set of simultaneous equations

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.6.6}$$

where **A** is a square matrix, **b** and **x** are vectors. Equation (2.6.6) defines **A** as a linear mapping from the vector space **x** to the vector space **b**. If **A** is singular, then there is some subspace of **x**, called the nullspace, that is mapped to zero, $\mathbf{A} \cdot \mathbf{x} = 0$. The dimension of the nullspace (the number of linearly independent vectors **x** that can be found in it) is called the *nullity* of **A**.

Now, there is also some subspace of **b** that can be "reached" by **A**, in the sense that there exists some **x** which is mapped there. This subspace of **b** is called the range of **A**. The dimension of the range is called the *rank* of **A**. If **A** is nonsingular, then its range will be all of the vector space **b**, so its rank is N. If **A** is singular, then the rank will be less than N. In fact, the relevant theorem is "rank plus nullity equals N."

What has this to do with SVD? SVD explicitly constructs orthonormal bases for the nullspace and range of a matrix. Specifically, the columns of **U** whose same-numbered elements w_j are *nonzero* are an orthonormal set of basis vectors that span the range; the columns of **V** whose same-numbered elements w_j are *zero* are an orthonormal basis for the nullspace.

Now let's have another look at solving the set of simultaneous linear equations (2.6.6) in the case that **A** is singular. First, the set of *homogeneous* equations, where $\mathbf{b} = 0$, is solved immediately by SVD: Any column of **V** whose corresponding w_j is zero yields a solution.

When the vector **b** on the right-hand side is not zero, the important question is whether it lies in the range of **A** or not. If it does, then the singular set of equations *does* have a solution **x**; in fact it has more than one solution, since any vector in the nullspace (any column of **V** with a corresponding zero w_j) can be added to **x** in any linear combination.

If we want to single out one particular member of this solution-set of vectors as a representative, we might want to pick the one with the smallest length $|\mathbf{x}|^2$. Here is how to find that vector using SVD: Simply *replace* $1/w_j$ *by zero if* $w_j = 0$. (It is not very often that one gets to set $\infty = 0$!) Then compute (working from right to left)

$$\mathbf{x} = \mathbf{V} \cdot [\mathrm{diag}\,(1/w_j)] \cdot (\mathbf{U}^T \cdot \mathbf{b}) \tag{2.6.7}$$

This will be the solution vector of smallest length; the columns of $\mathbf{V}$ that are in the nullspace complete the specification of the solution set.

Proof: Consider $|\mathbf{x} + \mathbf{x}'|$, where $\mathbf{x}'$ lies in the nullspace. Then, if $\mathbf{W}^{-1}$ denotes the modified inverse of $\mathbf{W}$ with some elements zeroed,

$$\begin{aligned} |\mathbf{x} + \mathbf{x}'| &= \left|\mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{x}'\right| \\ &= \left|\mathbf{V} \cdot (\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}')\right| \\ &= \left|\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}'\right| \end{aligned} \tag{2.6.8}$$

Here the first equality follows from (2.6.7), the second and third from the orthonormality of $\mathbf{V}$. If you now examine the two terms that make up the sum on the right-hand side, you will see that the first one has nonzero j components only where $w_j \neq 0$, while the second one, since $\mathbf{x}'$ is in the nullspace, has nonzero j components only where $w_j = 0$. Therefore the minimum length obtains for $\mathbf{x}' = 0$, q.e.d.

If $\mathbf{b}$ is not in the range of the singular matrix $\mathbf{A}$, then the set of equations (2.6.6) has no solution. But here is some good news: If $\mathbf{b}$ is not in the range of $\mathbf{A}$, then equation (2.6.7) can still be used to construct a "solution" vector $\mathbf{x}$. This vector $\mathbf{x}$ will not exactly solve $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. But, among all possible vectors $\mathbf{x}$, it will do the closest possible job in the least squares sense. In other words (2.6.7) finds

$$\mathbf{x} \quad \text{which minimizes} \quad r \equiv |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}| \tag{2.6.9}$$

The number r is called the *residual* of the solution.

The proof is similar to (2.6.8): Suppose we modify $\mathbf{x}$ by adding some arbitrary $\mathbf{x}'$. Then $\mathbf{A} \cdot \mathbf{x} - \mathbf{b}$ is modified by adding some $\mathbf{b}' \equiv \mathbf{A} \cdot \mathbf{x}'$. Obviously $\mathbf{b}'$ is in the range of $\mathbf{A}$. We then have

$$\begin{aligned} \left|\mathbf{A} \cdot \mathbf{x} - \mathbf{b} + \mathbf{b}'\right| &= \left|(\mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^T) \cdot (\mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b}) - \mathbf{b} + \mathbf{b}'\right| \\ &= \left|(\mathbf{U} \cdot \mathbf{W} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T - 1) \cdot \mathbf{b} + \mathbf{b}'\right| \\ &= \left|\mathbf{U} \cdot \left[(\mathbf{W} \cdot \mathbf{W}^{-1} - 1) \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{U}^T \cdot \mathbf{b}'\right]\right| \\ &= \left|(\mathbf{W} \cdot \mathbf{W}^{-1} - 1) \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{U}^T \cdot \mathbf{b}'\right| \end{aligned} \tag{2.6.10}$$

Now, $(\mathbf{W} \cdot \mathbf{W}^{-1} - 1)$ is a diagonal matrix which has nonzero j components only for $w_j = 0$, while $\mathbf{U}^T\mathbf{b}'$ has nonzero j components only for $w_j \neq 0$, since $\mathbf{b}'$ lies in the range of $\mathbf{A}$. Therefore the minimum obtains for $\mathbf{b}' = 0$, q.e.d.

Figure 2.6.1 summarizes our discussion of SVD thus far.

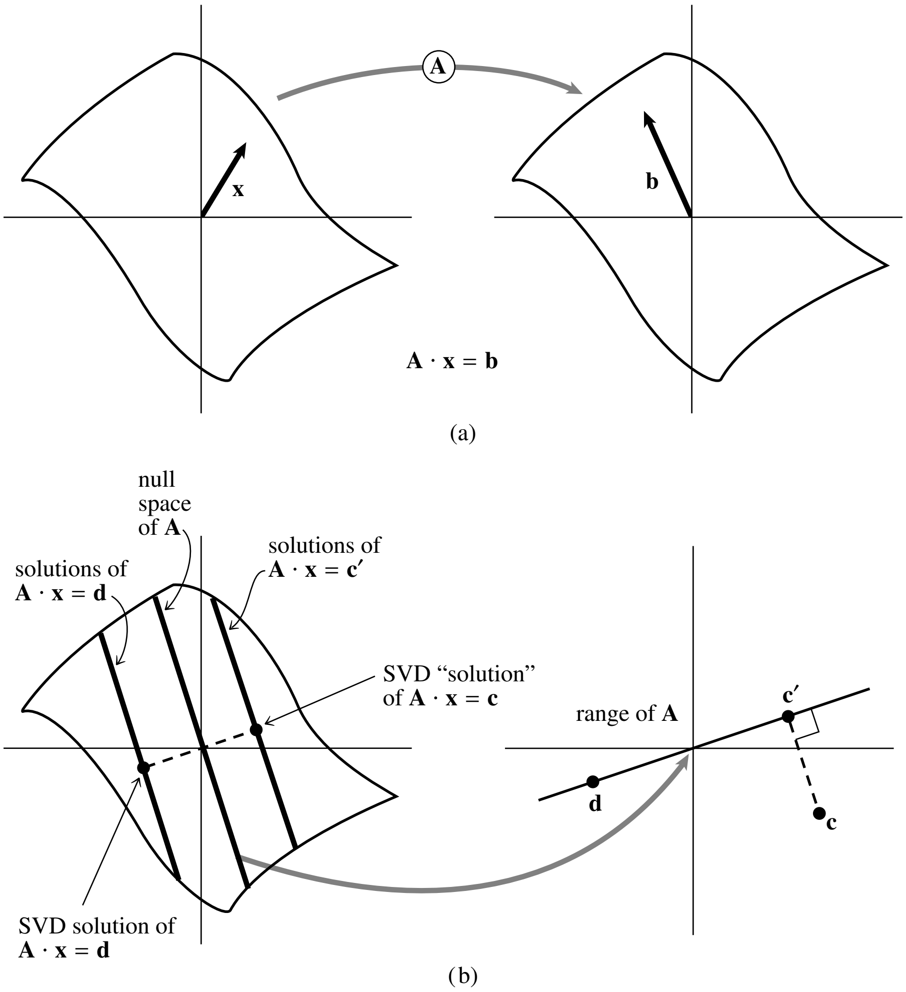

Figure 2.6.1. (a) A nonsingular matrix $\mathbf{A}$ maps a vector space into one of the same dimension. The vector $\mathbf{x}$ is mapped into $\mathbf{b}$, so that $\mathbf{x}$ satisfies the equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. (b) A singular matrix $\mathbf{A}$ maps a vector space into one of lower dimensionality, here a plane into a line, called the "range" of $\mathbf{A}$. The "nullspace" of $\mathbf{A}$ is mapped to zero. The solutions of $\mathbf{A} \cdot \mathbf{x} = \mathbf{d}$ consist of any one particular solution plus any vector in the nullspace, here forming a line parallel to the nullspace. Singular value decomposition (SVD) selects the particular solution closest to zero, as shown. The point $\mathbf{c}$ lies outside of the range of $\mathbf{A}$, so $\mathbf{A} \cdot \mathbf{x} = \mathbf{c}$ has no solution. SVD finds the least-squares best compromise solution, namely a solution of $\mathbf{A} \cdot \mathbf{x} = \mathbf{c}'$, as shown.

In the discussion since equation (2.6.6), we have been pretending that a matrix either is singular or else isn't. That is of course true analytically. Numerically, however, the far more common situation is that some of the w_j's are very small but nonzero, so that the matrix is ill-conditioned. In that case, the direct solution methods of LU decomposition or Gaussian elimination may actually give a formal solution to the set of equations (that is, a zero pivot may not be encountered); but the solution vector may have wildly large components whose algebraic cancellation, when multiplying by the matrix $\mathbf{A}$, may give a very poor approximation to the right-hand vector $\mathbf{b}$. In such cases, the solution vector $\mathbf{x}$ obtained by *zeroing* the small w_j's and then using equation (2.6.7) is very often better (in the sense of the residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ being smaller) than *both* the direct-method solution *and* the SVD solution where the small w_j's are left nonzero.

It may seem paradoxical that this can be so, since zeroing a singular value corresponds to throwing away one linear combination of the set of equations that we are trying to solve. The resolution of the paradox is that we are throwing away precisely a combination of equations that is so corrupted by roundoff error as to be at best useless; usually it is worse than useless since it "pulls" the solution vector way off towards infinity along some direction that is almost a nullspace vector. In doing this, it compounds the roundoff problem and makes the residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ larger.

SVD cannot be applied blindly, then. You have to exercise some discretion in deciding at what threshold to zero the small w_j's, and/or you have to have some idea what size of computed residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ is acceptable.

As an example, here is a "backsubstitution" routine `svbksb` for evaluating equation (2.6.7) and obtaining a solution vector $\mathbf{x}$ from a right-hand side $\mathbf{b}$, given that the SVD of a matrix $\mathbf{A}$ has already been calculated by a call to `svdcmp`. Note that this routine presumes that *you* have already zeroed the small w_j's. It does not do this for you. If you *haven't* zeroed the small w_j's, then this routine is just as ill-conditioned as any direct method, and you are misusing SVD.

```
#include "nrutil.h"

void svbksb(float **u, float w[], float **v, int m, int n, float b[], float x[])
Solves A·X = B for a vector X, where A is specified by the arrays u[1..m][1..n], w[1..n],
v[1..n][1..n] as returned by svdcmp. m and n are the dimensions of a, and will be equal for
square matrices. b[1..m] is the input right-hand side. x[1..n] is the output solution vector.
No input quantities are destroyed, so the routine may be called sequentially with different b's.
{
    int jj,j,i;
    float s,*tmp;

    tmp=vector(1,n);
    for (j=1;j<=n;j++) {            Calculate U^T B.
        s=0.0;
        if (w[j]) {                 Nonzero result only if w_j is nonzero.
            for (i=1;i<=m;i++) s += u[i][j]*b[i];
            s /= w[j];              This is the divide by w_j.
        }
        tmp[j]=s;
    }
    for (j=1;j<=n;j++) {            Matrix multiply by V to get answer.
        s=0.0;
        for (jj=1;jj<=n;jj++) s += v[j][jj]*tmp[jj];
        x[j]=s;
    }
    free_vector(tmp,1,n);
}
```

Note that a typical use of `svdcmp` and `svbksb` superficially resembles the typical use of `ludcmp` and `lubksb`: In both cases, you decompose the left-hand matrix $\mathbf{A}$ just once, and then can use the decomposition either once or many times with different right-hand sides. The crucial difference is the "editing" of the singular values before `svbksb` is called:

```
#define N ...
float wmax,wmin,**a,**u,*w,**v,*b,*x;
int i,j;
...
for(i=1;i<=N;i++)                   Copy a into u if you don't want it to be de-
    for j=1;j<=N;j++)                   stroyed.
        u[i][j]=a[i][j];
svdcmp(u,N,N,w,v);                  SVD the square matrix a.
wmax=0.0;                           Will be the maximum singular value obtained.
for(j=1;j<=N;j++) if (w[j] > wmax) wmax=w[j];
This is where we set the threshold for singular values allowed to be nonzero. The constant
is typical, but not universal. You have to experiment with your own application.
wmin=wmax*1.0e-6;
for(j=1;j<=N;j++) if (w[j] < wmin) w[j]=0.0;
svbksb(u,w,v,N,N,b,x);              Now we can backsubstitute.
```

SVD for Fewer Equations than Unknowns

If you have fewer linear equations M than unknowns N, then you are not expecting a unique solution. Usually there will be an $N - M$ dimensional family of solutions. If you want to find this whole solution space, then SVD can readily do the job.

The SVD decomposition will yield $N - M$ zero or negligible w_j's, since $M < N$. There may be additional zero w_j's from any degeneracies in your M equations. Be sure that you find this many small w_j's, and zero them before calling `svbksb`, which will give you the particular solution vector **x**. As before, the columns of **V** corresponding to zeroed w_j's are the basis vectors whose linear combinations, added to the particular solution, span the solution space.

SVD for More Equations than Unknowns

This situation will occur in Chapter 15, when we wish to find the least-squares solution to an overdetermined set of linear equations. In tableau, the equations to be solved are

$$\begin{pmatrix} & & \\ & & \\ & \mathbf{A} & \\ & & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{x} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \\ \mathbf{b} \\ \\ \\ \end{pmatrix} \tag{2.6.11}$$

The proofs that we gave above for the square case apply without modification to the case of more equations than unknowns. The least-squares solution vector **x** is given by (2.6.7), which, with nonsquare matrices, looks like this,

$$\begin{pmatrix} \mathbf{x} \end{pmatrix} = \begin{pmatrix} & \mathbf{V} & \end{pmatrix} \cdot \begin{pmatrix} \mathrm{diag}(1/w_j) \end{pmatrix} \cdot \begin{pmatrix} & & \mathbf{U}^T & & \end{pmatrix} \cdot \begin{pmatrix} \\ \\ \mathbf{b} \\ \\ \\ \end{pmatrix} \tag{2.6.12}$$

In general, the matrix **W** will not be singular, and no w_j's will need to be set to zero. Occasionally, however, there might be column degeneracies in **A**. In this case you will need to zero some small w_j values after all. The corresponding column in **V** gives the linear combination of **x**'s that is then ill-determined even by the supposedly overdetermined set.

Sometimes, although you do not need to zero any w_j's for *computational* reasons, you may nevertheless want to take note of any that are unusually small: Their corresponding columns in **V** are linear combinations of **x**'s which are insensitive to your data. In fact, you may then wish to zero these w_j's, to reduce the number of free parameters in the fit. These matters are discussed more fully in Chapter 15.

Constructing an Orthonormal Basis

Suppose that you have N vectors in an M-dimensional vector space, with $N \leq M$. Then the N vectors span some subspace of the full vector space. Often you want to construct an orthonormal set of N vectors that span the same subspace. The textbook way to do this is by Gram-Schmidt orthogonalization, starting with one vector and then expanding the subspace one dimension at a time. Numerically, however, because of the build-up of roundoff errors, naive Gram-Schmidt orthogonalization is *terrible*.

The right way to construct an orthonormal basis for a subspace is by SVD: Form an $M \times N$ matrix **A** whose N columns are your vectors. Run the matrix through `svdcmp`. The columns of the matrix **U** (which in fact replaces **A** on output from `svdcmp`) are your desired orthonormal basis vectors.

You might also want to check the output w_j's for zero values. If any occur, then the spanned subspace was not, in fact, N dimensional; the columns of **U** corresponding to zero w_j's should be discarded from the orthonormal basis set.

(QR factorization, discussed in §2.10, also constructs an orthonormal basis, see [5].)

Approximation of Matrices

Note that equation (2.6.1) can be rewritten to express any matrix A_{ij} as a sum of outer products of columns of **U** and rows of $\mathbf{V}^T$, with the "weighting factors" being the singular values w_j,

$$A_{ij} = \sum_{k=1}^{N} w_k \, U_{ik} V_{jk} \tag{2.6.13}$$

If you ever encounter a situation where *most* of the singular values w_j of a matrix **A** are very small, then **A** will be well-approximated by only a few terms in the sum (2.6.13). This means that you have to store only a few columns of **U** and **V** (the same k ones) and you will be able to recover, with good accuracy, the whole matrix.

Note also that it is very efficient to multiply such an approximated matrix by a vector **x**: You just dot **x** with each of the stored columns of **V**, multiply the resulting scalar by the corresponding w_k, and accumulate that multiple of the corresponding column of **U**. If your matrix is approximated by a small number K of singular values, then this computation of $\mathbf{A} \cdot \mathbf{x}$ takes only about $K(M+N)$ multiplications, instead of MN for the full matrix.

SVD Algorithm

Here is the algorithm for constructing the singular value decomposition of any matrix. See §11.2–§11.3, and also [4-5], for discussion relating to the underlying method.

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

void svdcmp(float **a, int m, int n, float w[], float **v)
```

Given a matrix `a[1..m][1..n]`, this routine computes its singular value decomposition, $A = U \cdot W \cdot V^T$. The matrix U replaces `a` on output. The diagonal matrix of singular values W is output as a vector `w[1..n]`. The matrix V (not the transpose V^T) is output as `v[1..n][1..n]`.

```
{
    float pythag(float a, float b);
    int flag,i,its,j,jj,k,l,nm;
    float anorm,c,f,g,h,s,scale,x,y,z,*rv1;

    rv1=vector(1,n);
    g=scale=anorm=0.0;                     Householder reduction to bidiagonal form.
    for (i=1;i<=n;i++) {
        l=i+1;
        rv1[i]=scale*g;
        g=s=scale=0.0;
        if (i <= m) {
            for (k=i;k<=m;k++) scale += fabs(a[k][i]);
            if (scale) {
                for (k=i;k<=m;k++) {
                    a[k][i] /= scale;
                    s += a[k][i]*a[k][i];
                }
                f=a[i][i];
                g = -SIGN(sqrt(s),f);
                h=f*g-s;
                a[i][i]=f-g;
                for (j=l;j<=n;j++) {
                    for (s=0.0,k=i;k<=m;k++) s += a[k][i]*a[k][j];
                    f=s/h;
                    for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
                }
                for (k=i;k<=m;k++) a[k][i] *= scale;
            }
        }
        w[i]=scale *g;
        g=s=scale=0.0;
        if (i <= m && i != n) {
            for (k=l;k<=n;k++) scale += fabs(a[i][k]);
            if (scale) {
```

```
                for (k=l;k<=n;k++) {
                    a[i][k] /= scale;
                    s += a[i][k]*a[i][k];
                }
                f=a[i][l];
                g = -SIGN(sqrt(s),f);
                h=f*g-s;
                a[i][l]=f-g;
                for (k=l;k<=n;k++) rv1[k]=a[i][k]/h;
                for (j=l;j<=m;j++) {
                    for (s=0.0,k=l;k<=n;k++) s += a[j][k]*a[i][k];
                    for (k=l;k<=n;k++) a[j][k] += s*rv1[k];
                }
                for (k=l;k<=n;k++) a[i][k] *= scale;
            }
        }
        anorm=FMAX(anorm,(fabs(w[i])+fabs(rv1[i])));
    }
    for (i=n;i>=1;i--) {                    Accumulation of right-hand transformations.
        if (i < n) {
            if (g) {
                for (j=l;j<=n;j++)          Double division to avoid possible underflow.
                    v[j][i]=(a[i][j]/a[i][l])/g;
                for (j=l;j<=n;j++) {
                    for (s=0.0,k=l;k<=n;k++) s += a[i][k]*v[k][j];
                    for (k=l;k<=n;k++) v[k][j] += s*v[k][i];
                }
            }
            for (j=l;j<=n;j++) v[i][j]=v[j][i]=0.0;
        }
        v[i][i]=1.0;
        g=rv1[i];
        l=i;
    }
    for (i=IMIN(m,n);i>=1;i--) {            Accumulation of left-hand transformations.
        l=i+1;
        g=w[i];
        for (j=l;j<=n;j++) a[i][j]=0.0;
        if (g) {
            g=1.0/g;
            for (j=l;j<=n;j++) {
                for (s=0.0,k=l;k<=m;k++) s += a[k][i]*a[k][j];
                f=(s/a[i][i])*g;
                for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
            }
            for (j=i;j<=m;j++) a[j][i] *= g;
        } else for (j=i;j<=m;j++) a[j][i]=0.0;
        ++a[i][i];
    }
    for (k=n;k>=1;k--) {                    Diagonalization of the bidiagonal form: Loop over
        for (its=1;its<=30;its++) {             singular values, and over allowed iterations.
            flag=1;
            for (l=k;l>=1;l--) {            Test for splitting.
                nm=l-1;                     Note that rv1[1] is always zero.
                if ((float)(fabs(rv1[l])+anorm) == anorm) {
                    flag=0;
                    break;
                }
                if ((float)(fabs(w[nm])+anorm) == anorm) break;
            }
            if (flag) {
                c=0.0;                      Cancellation of rv1[l], if l > 1.
                s=1.0;
                for (i=l;i<=k;i++) {
```

```
            f=s*rv1[i];
            rv1[i]=c*rv1[i];
            if ((float)(fabs(f)+anorm) == anorm) break;
            g=w[i];
            h=pythag(f,g);
            w[i]=h;
            h=1.0/h;
            c=g*h;
            s = -f*h;
            for (j=1;j<=m;j++) {
                y=a[j][nm];
                z=a[j][i];
                a[j][nm]=y*c+z*s;
                a[j][i]=z*c-y*s;
            }
        }
    }
    z=w[k];
    if (l == k) {                Convergence.
        if (z < 0.0) {           Singular value is made nonnegative.
            w[k] = -z;
            for (j=1;j<=n;j++) v[j][k] = -v[j][k];
        }
        break;
    }
    if (its == 30) nrerror("no convergence in 30 svdcmp iterations");
    x=w[l];                      Shift from bottom 2-by-2 minor.
    nm=k-1;
    y=w[nm];
    g=rv1[nm];
    h=rv1[k];
    f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y);
    g=pythag(f,1.0);
    f=((x-z)*(x+z)+h*((y/(f+SIGN(g,f)))-h))/x;
    c=s=1.0;                     Next QR transformation:
    for (j=l;j<=nm;j++) {
        i=j+1;
        g=rv1[i];
        y=w[i];
        h=s*g;
        g=c*g;
        z=pythag(f,h);
        rv1[j]=z;
        c=f/z;
        s=h/z;
        f=x*c+g*s;
        g = g*c-x*s;
        h=y*s;
        y *= c;
        for (jj=1;jj<=n;jj++) {
            x=v[jj][j];
            z=v[jj][i];
            v[jj][j]=x*c+z*s;
            v[jj][i]=z*c-x*s;
        }
        z=pythag(f,h);
        w[j]=z;                  Rotation can be arbitrary if z = 0.
        if (z) {
            z=1.0/z;
            c=f*z;
            s=h*z;
        }
        f=c*g+s*y;
        x=c*y-s*g;
```

```
                for (jj=1;jj<=m;jj++) {
                    y=a[jj][j];
                    z=a[jj][i];
                    a[jj][j]=y*c+z*s;
                    a[jj][i]=z*c-y*s;
                }
            }
            rv1[l]=0.0;
            rv1[k]=f;
            w[k]=x;
        }
    }
    free_vector(rv1,1,n);
}
```

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

float pythag(float a, float b)
Computes (a^2 + b^2)^1/2 without destructive underflow or overflow.
{
    float absa,absb;
    absa=fabs(a);
    absb=fabs(b);
    if (absa > absb) return absa*sqrt(1.0+SQR(absb/absa));
    else return (absb == 0.0 ? 0.0 : absb*sqrt(1.0+SQR(absa/absb)));
}
```

(Double precision versions of svdcmp, svbksb, and pythag, named dsvdcmp, dsvbksb, and dpythag, are used by the routine ratlsq in §5.13. You can easily make the conversions, or else get the converted routines from the *Numerical Recipes* diskette.)

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §8.3 and Chapter 12.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 18.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I.10 by G.H. Golub and C. Reinsch. [2]

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.), Chapter 11. [3]

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).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.7. [4]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §5.2.6. [5]

2.7 Sparse Linear Systems

A system of linear equations is called *sparse* if only a relatively small number of its matrix elements a_{ij} are nonzero. It is wasteful to use general methods of linear algebra on such problems, because most of the $O(N^3)$ arithmetic operations devoted to solving the set of equations or inverting the matrix involve zero operands. Furthermore, you might wish to work problems so large as to tax your available memory space, and it is wasteful to reserve storage for unfruitful zero elements. Note that there are two distinct (and not always compatible) goals for any sparse matrix method: saving time and/or saving space.

We have already considered one archetypal sparse form in §2.4, the band diagonal matrix. In the tridiagonal case, e.g., we saw that it was possible to save both time (order N instead of N^3) and space (order N instead of N^2). The method of solution was not different in principle from the general method of LU decomposition; it was just applied cleverly, and with due attention to the bookkeeping of zero elements. Many practical schemes for dealing with sparse problems have this same character. They are fundamentally decomposition schemes, or else elimination schemes akin to Gauss-Jordan, but carefully optimized so as to minimize the number of so-called *fill-ins*, initially zero elements which must become nonzero during the solution process, and for which storage must be reserved.

Direct methods for solving sparse equations, then, depend crucially on the precise pattern of sparsity of the matrix. Patterns that occur frequently, or that are useful as way-stations in the reduction of more general forms, already have special names and special methods of solution. We do not have space here for any detailed review of these. References listed at the end of this section will furnish you with an "in" to the specialized literature, and the following list of buzz words (and Figure 2.7.1) will at least let you hold your own at cocktail parties:

- tridiagonal
- band diagonal (or banded) with bandwidth M
- band triangular
- block diagonal
- block tridiagonal
- block triangular
- cyclic banded
- singly (or doubly) bordered block diagonal
- singly (or doubly) bordered block triangular
- singly (or doubly) bordered band diagonal
- singly (or doubly) bordered band triangular
- other (!)

You should also be aware of some of the special sparse forms that occur in the solution of partial differential equations in two or more dimensions. See Chapter 19.

If your particular pattern of sparsity is not a simple one, then you may wish to try an *analyze/factorize/operate* package, which automates the procedure of figuring out how fill-ins are to be minimized. The *analyze* stage is done once only for each pattern of sparsity. The *factorize* stage is done once for each particular matrix that fits the pattern. The *operate* stage is performed once for each right-hand side to