◊ III.4

Polar Matrix Decomposition

Ken Shoemake
University of Pennsylvania
Philadelphia, PA
shoemake@graphics.cis.upenn.edu

◊ Introduction ◊

Extracting meaning from matrices is a compelling challenge, judging by the number of previous Gems on the subject. Affine matrices are especially awkward to dissect (Thomas 1991, Goldman 1992). A combination of rotating, scaling, shearing, and translating will generate any affine transform. Yet while a rotation, say, has a simple geometric meaning by itself, the rotation chosen by these previous Gems generally does not—as a factor. Since it is entirely dependent on the basis chosen to express the transformation as a matrix, the *decomposition* forfeits any claim of being geometrically meaningful. There is, however, an alternative: a physical, intuitive way to decompose an affine matrix. It is based on the linear algebra *polar decomposition.* The text of this Gem sketches the theory and benefits of polar decomposition,[1] and the code gives a sample implementation.

Why does it matter if a decomposition is geometric, and what does it mean? It may not matter if, for example, you simply need to recreate the effect of a given matrix using a graphics library with a limited choice of primitives. But if a human being is going to try to interpret the results, or if an animation program is going to try to interpolate the results, it matters a great deal. A good decomposition will give interpretations and interpolations that correspond to our perceptually based intuitions. Physics and psychology suggest two important criteria: coordinate independence and rigidity preservation. The first of these is the minimum needed for decomposition to have true geometric meaning.

Coordinate independence is an elaborate name for a simple observation. So far as we can tell, the universe has no built-in coordinate system. There is no special direction to call x or y, no self-evident unit of distance, no designated origin. The implication for matrix decomposition is that the results should not depend on the particular coordinate basis used, so long as the axes are perpendicular and scaled the same in all directions. In mathematical terms this has the following consequence. Suppose $\mathbf{M}$ is a linear transformation in one such basis, and $\mathbf{M}'$ is the same transformation expressed in another,

[1]See (Shoemake and Duff 1992) for a longer discussion.

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

so that $\mathbf{M}' = \mathbf{BMB}^{-1}$. Now suppose $\mathbf{M}$ and $\mathbf{M}'$ decompose as, say, $\mathbf{M} = \mathbf{QS}$ and $\mathbf{M}' = \mathbf{Q}'\mathbf{S}'$. Then we should find that $\mathbf{Q}' = \mathbf{BQB}^{-1}$ and $\mathbf{S}' = \mathbf{BSB}^{-1}$. The Thomas and Goldman decompositions fail this test.

The second test, rigidity preservation, is more subtle and less certain, but important nevertheless. A variety of experiments in psychology give compelling evidence that the human visual system will interpret motion as rigid motion if possible. All rigid motion consists of rotations and translations, so this is a stringent demand, and not always possible. We can only say that, to the extent we can decompose a transform rigidly, we should do so. The Thomas and Goldman decompositions also fail this test.

◇ Polar Decomposition ◇

The polar form of a non-zero complex number $z = re^{i\theta}$ consists of two factors: a positive scaling r, and a rotation $e^{i\theta}$. The polar decomposition of a non-singular matrix $\mathbf{M} = \mathbf{QS}$ also consists of a positive scaling $\mathbf{S}$, and a rotation $\mathbf{Q}$, with two quibbles. The first quibble is that the $\mathbf{S}$ factor is a little more general than the usual scale matrix, since it need not be diagonal. Formally, it is a symmetric positive semi-definite matrix, which means it is only diagonal in *some* orthonormal basis, and has positive (or zero if $\mathbf{M}$ is singular) scale factors that can differ from each other. There is no standard computer graphics name for such a matrix, but I call it a *stretch* matrix, since it reminds me of cartoon animation's squash and stretch. The second quibble is that the $\mathbf{Q}$ factor is not necessarily a rotation. It is a more general *orthogonal* matrix (which can reflect as well as rotate), having a determinant of the same sign as $\mathbf{M}$. Quibbles aside, the motivation for the name should be clear.

Polar decomposition is ideally suited for our purposes, for three reasons. First, it is coordinate independent, so it tells us about the transformation in physically meaningful terms. Second, any non-singular matrix $\mathbf{M}$ has a unique polar decomposition, and $\mathbf{Q}$ is as close to $\mathbf{M}$ as possible,[2] equaling $\mathbf{M}$ if $\mathbf{M}$ is orthogonal. Thus, to the extent possible, rigidity is preserved. Third, it is simple to calculate. The $\mathbf{Q}$ factor can be obtained by setting $\mathbf{Q}_0 = \mathbf{M}$, then iterating $\mathbf{Q}_{i+1} = (\mathbf{Q}_i + (\mathbf{Q}_i^{-1})^{\mathrm{T}})/2$ until there is negligible change. An earlier Gem (Raible 1990) gives another application and a fast approximate calculation method, though polar decomposition is not explicitly mentioned.

◇ Decomposing an Affine Transformation ◇

We know that any 3D affine transformation can be factored as a 3D linear transformation followed by a translation. To find the translation, transform the zero vector. Assuming the transformation is given as a homogeneous matrix that acts on column

[2]Closeness can be measured using the sum of the squares of the element differences.

vectors, this step is trivial: strip off the last column of the matrix. This is the factorization $\mathbf{A} = \mathbf{TM}$. Then find the polar decomposition of $\mathbf{M}$, so we have $\mathbf{A} = \mathbf{TQS}$.

We could stop here, but more often we will want to further decompose the $\mathbf{Q}$ and $\mathbf{S}$ factors. If $\mathbf{Q}$ is not a rotation, it must be the negative of one (since 3 is an odd number of dimensions). Thus we can factor $\mathbf{Q} = \mathbf{FR}$, where $\mathbf{R}$ is a rotation and $\mathbf{F}$ (for flip) is $\pm\mathbf{I}$. If $\mathbf{Q}$ is a rotation, $\det(\mathbf{Q}) = +1$, otherwise $\det(\mathbf{Q}) = -1$, so this factorization is easy to compute. Since a rotation matrix can easily be described with a quaternion or an axis and angle, we will not decompose it further.

As noted above, the $\mathbf{S}$ factor is symmetric positive semi-definite. Thus it has a *spectral decomposition* $\mathbf{S} = \mathbf{UKU}^{\mathrm{T}}$, where $\mathbf{U}$ is a rotation and $\mathbf{K}$ (for scale) is diagonal with positive or zero entries. The scaling is usually non-uniform, i.e., different along each coordinate axis.[3] For human comprehension and simpler primitives, it will be helpful to include this extra step.

Be aware, however, that the spectral decomposition is not unique. Although $\mathbf{S}$ as a whole has geometric significance, its $\mathbf{U}$ and $\mathbf{K}$ factors are more ambiguous. Changing the labels and reversing the directions of the axes along which $\mathbf{S}$ is diagonalized[4] changes $\mathbf{U}$ and $\mathbf{K}$, but not the product $\mathbf{UKU}^{\mathrm{T}}$. I suggest picking the $\mathbf{U}$ with the smallest rotation angle; but consult (Shoemake and Duff 1992) for a more extended discussion of polar and spectral decomposition in the context of animation.

We now have a complete decomposition, $\mathbf{M} = \mathbf{TQS} = \mathbf{TFRUKU}^{\mathrm{T}}$. As a reality check, let's count degrees of freedom. The original matrix, $\mathbf{A}$, has 12 freely chosen entries, and so 12 degrees of freedom. The flip $\mathbf{F}$ is only a sign choice, so it contributes no degrees of freedom. The rotations $\mathbf{R}$ and $\mathbf{U}$ each have three degrees of freedom, as do the scaling $\mathbf{K}$ and the translation $\mathbf{T}$. (We count $\mathbf{U}$ only once, though it's used twice.) Thus we exactly match the needed degrees of freedom.

◇ Implementation ◇

Complete $\mathbf{TFRUKU}^{\mathrm{T}}$ factoring requires computing polar and spectral decompositions. If you already have a singular value decomposition routine, you can use its results to do both. The SVD of $\mathbf{M}$ has the form $\mathbf{VKU}^{\mathrm{T}}$, directly giving $\mathbf{U}$ and $\mathbf{K}$, and indirectly giving $\mathbf{Q} = \mathbf{VU}^{\mathrm{T}}$. (It may be necessary to negate $\mathbf{U}$ and $\mathbf{V}$ to ensure that $\mathbf{U}$ is a rotation.) Though SVD routines are complicated, they are also reliable.

But you don't need SVD code. If you have a symmetric eigenvalue routine (another name for spectral decomposition), you can use it instead. Decompose $\mathbf{M}^{\mathrm{T}}\mathbf{M}$ as $\mathbf{UDU}^{\mathrm{T}}$, where $\mathbf{D}$ is diagonal, and compute $\mathbf{K}$ from $\mathbf{D}$ by taking the positive square root of each entry. Then if $\mathbf{M}$ is non-singular, $\mathbf{Q}$ can be computed as $\mathbf{MUK}^{-1}\mathbf{U}^{\mathrm{T}}$. This approach

[3]The scale factors are the eigenvalues of $\mathbf{S}$ and the singular values of $\mathbf{M}$.

[4]The axis directions are the eigenvectors of $\mathbf{S}$.

should be used with caution, for $\mathbf{K}$ has no inverse if $\mathbf{M}$ is singular, and some accuracy will always be lost.

As mentioned earlier, however, the simplest approach is to set $\mathbf{Q}_0 = \mathbf{M}$, then iteratively compute $\mathbf{Q}_{i+1} = (\mathbf{Q}_i + (\mathbf{Q}_i^{-1})^{\mathrm{T}})/2$ until the difference between the entries of $\mathbf{Q}_i$ and $\mathbf{Q}_{i+1}$ is nearly zero. (By definition an orthogonal matrix satisfies $\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{I}$, so $\mathbf{Q} = (\mathbf{Q}^{-1})^{\mathrm{T}}$.) When $\mathbf{M}$ is already nearly orthogonal, this iteration will converge quadratically, but the code given below includes a trick (Higham and Schreiber 1988) to accelerate convergence when there are large scale factors.

A singular $\mathbf{M}$ need only be a nuisance for this code, not an impenetrable barrier; but a spectral decomposition will still be needed after computing $\mathbf{S} = \mathbf{Q}^{\mathrm{T}}\mathbf{M}$. Since $\mathbf{S}$ has a characteristic polynomial which is only cubic, its roots (the diagonal of $\mathbf{K}$) can be found in closed form. But accuracy requires care, and $\mathbf{U}$ must still be found; so instead, try the method of Jacobi. A series of plane rotations $\mathbf{U}_i$ will force $\mathbf{S}$ to converge to diagonal form: $\mathbf{U}_n^{\mathrm{T}} \cdots \mathbf{U}_2^{\mathrm{T}}\mathbf{U}_1^{\mathrm{T}}\mathbf{S}\mathbf{U}_1\mathbf{U}_2 \cdots \mathbf{U}_n = \mathbf{K}$; then $\mathbf{U} = \mathbf{U}_1\mathbf{U}_2 \cdots \mathbf{U}_n$.

◇ Decomposing the Inverse ◇

If $\mathbf{A}$ has been factored as $\mathbf{TFRUKU}^{\mathrm{T}}$, it is cheap to compute its inverse. The derivation begins by distributing inversion to each of the factors:

$$(\mathbf{TFRUKU}^{\mathrm{T}})^{-1} = (\mathbf{U}^{\mathrm{T}})^{-1}\mathbf{K}^{-1}\mathbf{U}^{-1}\mathbf{R}^{-1}\mathbf{F}^{-1}\mathbf{T}^{-1}.$$

Since $\mathbf{U}$ and $\mathbf{R}$ are orthogonal and $\mathbf{F}$ is its own inverse, this simplifies to

$$\mathbf{U}\mathbf{K}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{R}^{\mathrm{T}}\mathbf{F}\mathbf{T}^{-1}.$$

These matrices are easy. The inverse of $\mathbf{K} = \mathrm{Scale}(k_x, k_y, k_z)$ is $\mathrm{Scale}(k_x^{-1}, k_y^{-1}, k_z^{-1})$. The inverse of $\mathbf{T} = \mathrm{Translate}(t_x, t_y, t_z)$ is $\mathrm{Translate}(-t_x, -t_y, -t_z)$. In the event we have stored the rotations as quaternions, we simply conjugate them.

If we only need the inverse as a matrix, we multiply everything and quit. But if we want it in the form $\mathbf{A}^{-1} = \mathbf{T}'\mathbf{F}'\mathbf{R}'\mathbf{U}'\mathbf{K}'(\mathbf{U}')^{\mathrm{T}}$, we must manipulate our terms into the correct order. First, premultiply by $\mathbf{R}^{\mathrm{T}}\mathbf{R}$, which is the identity, and regroup to obtain

$$\mathbf{R}^{\mathrm{T}}(\mathbf{RU})\mathbf{K}^{-1}(\mathbf{RU})^{\mathrm{T}}\mathbf{F}\mathbf{T}^{-1}.$$

Now $\mathbf{F}$ commutes with everything, so we can move it to the front and identify the terms $\mathbf{F}' = \mathbf{F}$, $\mathbf{R}' = \mathbf{R}^{\mathrm{T}}$, $\mathbf{U}' = \mathbf{RU}$, and $\mathbf{K}' = \mathbf{K}^{-1}$.

This gives us the form $\mathbf{M}'\mathbf{T}^{-1}$, but what we really want is $\mathbf{T}'\mathbf{M}'$. To order the translation correctly, we take $\mathbf{T}' = \mathrm{Translate}(-\mathbf{M}'T)$, where $\mathbf{T} = \mathrm{Translate}(T)$.

◇ Code ◇

```
/**** Decompose.h - Basic declarations ****/
#ifndef _H_Decompose
#define _H_Decompose
typedef struct {float x, y, z, w;} Quat; /* Quaternion */
enum QuatPart {X, Y, Z, W};
typedef Quat HVect; /* Homogeneous 3D vector */
typedef float HMatrix[4][4]; /* Right-handed, for column vectors */
typedef struct {
    HVect t;    /* Translation components */
    Quat  q;    /* Essential rotation     */
    Quat  u;    /* Stretch rotation       */
    HVect k;    /* Stretch factors        */
    float f;    /* Sign of determinant    */
} AffineParts;
float polar_decomp(HMatrix M, HMatrix Q, HMatrix S);
HVect spect_decomp(HMatrix S, HMatrix U);
Quat snuggle(Quat q, HVect *k);
void decomp_affine(HMatrix A, AffineParts *parts);
void invert_affine(AffineParts *parts, AffineParts *inverse);
#endif
/**** EOF ****/

/**** Decompose.c ****/
/* Ken Shoemake, 1993 */
#include <math.h>
#include "Decompose.h"

/******* Matrix Preliminaries *******/

/** Fill out 3x3 matrix to 4x4 **/
#define mat_pad(A) (A[W][X]=A[X][W]=A[W][Y]=A[Y][W]=A[W][Z]=A[Z][W]=0,A[W][W]=1)

/** Copy nxn matrix A to C using "gets" for assignment **/
#define mat_copy(C,gets,A,n) {int i,j; for(i=0;i<n;i++) for(j=0;j<n;j++)\
    C[i][j] gets (A[i][j]);}

/** Copy transpose of nxn matrix A to C using "gets" for assignment **/
#define mat_tpose(AT,gets,A,n) {int i,j; for(i=0;i<n;i++) for(j=0;j<n;j++)\
    AT[i][j] gets (A[j][i]);}

/** Assign nxn matrix C the element-wise combination of A and B using "op" **/
#define mat_binop(C,gets,A,op,B,n) {int i,j; for(i=0;i<n;i++) for(j=0;j<n;j++)\
    C[i][j] gets (A[i][j]) op (B[i][j]);}

/** Multiply the upper left 3x3 parts of A and B to get AB **/
void mat_mult(HMatrix A, HMatrix B, HMatrix AB)
{
    int i, j;
    for (i=0; i<3; i++) for (j=0; j<3; j++)
        AB[i][j] = A[i][0]*B[0][j] + A[i][1]*B[1][j] + A[i][2]*B[2][j];
}
```

```
/** Return dot product of length 3 vectors va and vb **/
float vdot(float *va, float *vb)
{
    return (va[0]*vb[0] + va[1]*vb[1] + va[2]*vb[2]);
}

/** Set v to cross product of length 3 vectors va and vb **/
void vcross(float *va, float *vb, float *v)
{
    v[0] = va[1]*vb[2] - va[2]*vb[1];
    v[1] = va[2]*vb[0] - va[0]*vb[2];
    v[2] = va[0]*vb[1] - va[1]*vb[0];
}

/** Set MadjT to transpose of inverse of M times determinant of M **/
void adjoint_transpose(HMatrix M, HMatrix MadjT)
{
    vcross(M[1], M[2], MadjT[0]);
    vcross(M[2], M[0], MadjT[1]);
    vcross(M[0], M[1], MadjT[2]);
}

/******* Quaternion Preliminaries *******/

/* Construct a (possibly non-unit) quaternion from real components */
Quat Qt_(float x, float y, float z, float w)
{
    Quat qq;
    qq.x = x; qq.y = y; qq.z = z; qq.w = w;
    return (qq);
}

/* Return conjugate of quaternion */
Quat Qt_Conj(Quat q)
{
    Quat qq;
    qq.x = -q.x; qq.y = -q.y; qq.z = -q.z; qq.w = q.w;
    return (qq);
}

/* Return quaternion product qL * qR.  Note: order is important!
 * To combine rotations, use the product Mul(qSecond, qFirst),
 * which gives the effect of rotating by qFirst then qSecond. */
Quat Qt_Mul(Quat qL, Quat qR)
{
    Quat qq;
    qq.w = qL.w*qR.w - qL.x*qR.x - qL.y*qR.y - qL.z*qR.z;
    qq.x = qL.w*qR.x + qL.x*qR.w + qL.y*qR.z - qL.z*qR.y;
    qq.y = qL.w*qR.y + qL.y*qR.w + qL.z*qR.x - qL.x*qR.z;
    qq.z = qL.w*qR.z + qL.z*qR.w + qL.x*qR.y - qL.y*qR.x;
    return (qq);
}
```

```
/* Return product of quaternion q by scalar w */
Quat Qt_Scale(Quat q, float w)
{
    Quat qq;
    qq.w = q.w*w; qq.x = q.x*w; qq.y = q.y*w; qq.z = q.z*w;
    return (qq);
}

/* Construct a unit quaternion from rotation matrix.  Assumes matrix is
 * used to multiply column vector on the left: vnew = mat vold.  Works
 * correctly for right-handed coordinate system and right-handed rotations.
 * Translation and perspective components ignored. */
Quat Qt_FromMatrix(HMatrix mat)
{
    /* This algorithm avoids near-zero divides by looking for a large component
     * -- first w, then x, y, or z.  When the trace is greater than zero,
     * |w| is greater than 1/2, which is as small as a largest component can be.
     * Otherwise, the largest diagonal entry corresponds to the largest of |x|,
     * |y|, or |z|, one of which must be larger than |w|, and at least 1/2. */
    Quat qu;
    register double tr, s;

    tr = mat[X][X] + mat[Y][Y]+ mat[Z][Z];
    if (tr >= 0.0) {
            s = sqrt(tr + mat[W][W]);
            qu.w = s*0.5;
            s = 0.5 / s;
            qu.x = (mat[Z][Y] - mat[Y][Z]) * s;
            qu.y = (mat[X][Z] - mat[Z][X]) * s;
            qu.z = (mat[Y][X] - mat[X][Y]) * s;
        } else {
            int h = X;
            if (mat[Y][Y] > mat[X][X]) h = Y;
            if (mat[Z][Z] > mat[h][h]) h = Z;
            switch (h) {
#define caseMacro(i,j,k,I,J,K) \
            case I:\
                s = sqrt( (mat[I][I] - (mat[J][J]+mat[K][K])) + mat[W][W] );\
                qu.i = s*0.5;\
                s = 0.5 / s;\
                qu.j = (mat[I][J] + mat[J][I]) * s;\
                qu.k = (mat[K][I] + mat[I][K]) * s;\
                qu.w = (mat[K][J] - mat[J][K]) * s;\
                break
            caseMacro(x,y,z,X,Y,Z);
            caseMacro(y,z,x,Y,Z,X);
            caseMacro(z,x,y,Z,X,Y);
            }
        }
    if (mat[W][W] != 1.0) qu = Qt_Scale(qu, 1/sqrt(mat[W][W]));
    return (qu);
}
```

```
/******* Decomp Auxiliaries *******/

static HMatrix mat_id = {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}};

/** Compute either the 1 or the infinity norm of M, depending on tpose **/
float mat_norm(HMatrix M, int tpose)
{
    int i;
    float sum, max;
    max = 0.0;
    for (i=0; i<3; i++) {
        if (tpose) sum = fabs(M[0][i])+fabs(M[1][i])+fabs(M[2][i]);
        else       sum = fabs(M[i][0])+fabs(M[i][1])+fabs(M[i][2]);
        if (max<sum) max = sum;
    }
    return max;
}

float norm_inf(HMatrix M) {mat_norm(M, 0);}
float norm_one(HMatrix M) {mat_norm(M, 1);}

/** Return index of column of M containing maximum abs entry, or -1 if M=0 **/
int find_max_col(HMatrix M)
{
    float abs, max;
    int i, j, col;
    max = 0.0; col = -1;
    for (i=0; i<3; i++) for (j=0; j<3; j++) {
        abs = M[i][j]; if (abs<0.0) abs = -abs;
        if (abs>max) {max = abs; col = j;}
    }
    return col;
}

/** Make u for Householder reflection to zero all v components but first **/
void make_reflector(float *v, float *u)
{
    float s = sqrt(vdot(v, v));
    u[0] = v[0]; u[1] = v[1];
    u[2] = v[2] + ((v[2]<0.0) ? -s : s);
    s = sqrt(2.0/vdot(u, u));
    u[0] = u[0]*s; u[1] = u[1]*s; u[2] = u[2]*s;
}

/** Apply Householder reflection represented by u to column vectors of M **/
void reflect_cols(HMatrix M, float *u)
{
    int i, j;
    for (i=0; i<3; i++) {
        float s = u[0]*M[0][i] + u[1]*M[1][i] + u[2]*M[2][i];
        for (j=0; j<3; j++) M[j][i] -= u[j]*s;
    }
}
```

```
/** Apply Householder reflection represented by u to row vectors of M **/
void reflect_rows(HMatrix M, float *u)
{
    int i, j;
    for (i=0; i<3; i++) {
        float s = vdot(u, M[i]);
        for (j=0; j<3; j++) M[i][j] -= u[j]*s;
    }
}

/** Find orthogonal factor Q of rank 1 (or less) M **/
void do_rank1(HMatrix M, HMatrix Q)
{
    float v1[3], v2[3], s;
    int col;
    mat_copy(Q,=,mat_id,4);
    /* If rank(M) is 1, we should find a non-zero column in M */
    col = find_max_col(M);
    if (col<0) return; /* Rank is 0 */
    v1[0] = M[0][col]; v1[1] = M[1][col]; v1[2] = M[2][col];
    make_reflector(v1, v1); reflect_cols(M, v1);
    v2[0] = M[2][0]; v2[1] = M[2][1]; v2[2] = M[2][2];
    make_reflector(v2, v2); reflect_rows(M, v2);
    s = M[2][2];
    if (s<0.0) Q[2][2] = -1.0;
    reflect_cols(Q, v1); reflect_rows(Q, v2);
}

/** Find orthogonal factor Q of rank 2 (or less) M using adjoint transpose **/
void do_rank2(HMatrix M, HMatrix MadjT, HMatrix Q)
{
    float v1[3], v2[3];
    float w, x, y, z, c, s, d;
    int i, j, col;
    /* If rank(M) is 2, we should find a non-zero column in MadjT */
    col = find_max_col(MadjT);
    if (col<0) {do_rank1(M, Q); return;} /* Rank<2 */
    v1[0] = MadjT[0][col]; v1[1] = MadjT[1][col]; v1[2] = MadjT[2][col];
    make_reflector(v1, v1); reflect_cols(M, v1);
    vcross(M[0], M[1], v2);
    make_reflector(v2, v2); reflect_rows(M, v2);
    w = M[0][0]; x = M[0][1]; y = M[1][0]; z = M[1][1];
    if (w*z>x*y) {
        c = z+w; s = y-x; d = sqrt(c*c+s*s); c = c/d; s = s/d;
        Q[0][0] = Q[1][1] = c; Q[0][1] = -(Q[1][0] = s);
    } else {
        c = z-w; s = y+x; d = sqrt(c*c+s*s); c = c/d; s = s/d;
        Q[0][0] = -(Q[1][1] = c); Q[0][1] = Q[1][0] = s;
    }
    Q[0][2] = Q[2][0] = Q[1][2] = Q[2][1] = 0.0; Q[2][2] = 1.0;
    reflect_cols(Q, v1); reflect_rows(Q, v2);
}
```

```
/******* Polar Decomposition *******/

/* Polar Decomposition of 3x3 matrix in 4x4,
 * M = QS.  See Nicholas Higham and Robert S. Schreiber,
 * Fast Polar Decomposition of an Arbitrary Matrix,
 * Technical Report 88-942, October 1988,
 * Department of Computer Science, Cornell University.
 */
float polar_decomp(HMatrix M, HMatrix Q, HMatrix S)
{
#define TOL 1.0e-6
    HMatrix Mk, MadjTk, Ek;
    float det, M_one, M_inf, MadjT_one, MadjT_inf, E_one, gamma, t1, t2, g1, g2;
    int i, j;
    mat_tpose(Mk,=,M,3);
    M_one = norm_one(Mk);  M_inf = norm_inf(Mk);
    do {
        adjoint_transpose(Mk, MadjTk);
        det = vdot(Mk[0], MadjTk[0]);
        if (det==0.0) {do_rank2(Mk, MadjTk, Mk); break;}
        MadjT_one = norm_one(MadjTk); MadjT_inf = norm_inf(MadjTk);
        gamma = sqrt(sqrt((MadjT_one*MadjT_inf)/(M_one*M_inf))/fabs(det));
        g1 = gamma*0.5;
        g2 = 0.5/(gamma*det);
        mat_copy(Ek,=,Mk,3);
        mat_binop(Mk,=,g1*Mk,+,g2*MadjTk,3);
        mat_copy(Ek,-=,Mk,3);
        E_one = norm_one(Ek);
        M_one = norm_one(Mk);  M_inf = norm_inf(Mk);
    } while (E_one>(M_one*TOL));
    mat_tpose(Q,=,Mk,3); mat_pad(Q);
    mat_mult(Mk, M, S);  mat_pad(S);
    for (i=0; i<3; i++) for (j=i; j<3; j++)
        S[i][j] = S[j][i] = 0.5*(S[i][j]+S[j][i]);
    return (det);
}
```

```
/******* Spectral Decomposition *******/

/* Compute the spectral decomposition of symmetric positive semi-definite S.
 * Returns rotation in U and scale factors in result, so that if K is a diagonal
 * matrix of the scale factors, then S = U K (U transpose). Uses Jacobi method.
 * See Gene H. Golub and Charles F. Van Loan, Matrix Computations, Hopkins 1983.
 */
HVect spect_decomp(HMatrix S, HMatrix U)
{
    HVect kv;
    double Diag[3],OffD[3]; /* OffD is off-diag (by omitted index) */
    double g,h,fabsh,fabsOffDi,t,theta,c,s,tau,ta,OffDq,a,b;
    static char nxt[] = {Y,Z,X};
    int sweep, i, j;
    mat_copy(U,=,mat_id,4);
    Diag[X] = S[X][X]; Diag[Y] = S[Y][Y]; Diag[Z] = S[Z][Z];
    OffD[X] = S[Y][Z]; OffD[Y] = S[Z][X]; OffD[Z] = S[X][Y];
    for (sweep=20; sweep>0; sweep--) {
        float sm = fabs(OffD[X])+fabs(OffD[Y])+fabs(OffD[Z]);
        if (sm==0.0) break;
        for (i=Z; i>=X; i--) {
            int p = nxt[i]; int q = nxt[p];
            fabsOffDi = fabs(OffD[i]);
            g = 100.0*fabsOffDi;
            if (fabsOffDi>0.0) {
                h = Diag[q] - Diag[p];
                fabsh = fabs(h);
                if (fabsh+g==fabsh) {
                    t = OffD[i]/h;
                } else {
                    theta = 0.5*h/OffD[i];
                    t = 1.0/(fabs(theta)+sqrt(theta*theta+1.0));
                    if (theta<0.0) t = -t;
                }
                c = 1.0/sqrt(t*t+1.0); s = t*c;
                tau = s/(c+1.0);
                ta = t*OffD[i]; OffD[i] = 0.0;
                Diag[p] -= ta; Diag[q] += ta;
                OffDq = OffD[q];
                OffD[q] -= s*(OffD[p] + tau*OffD[q]);
                OffD[p] += s*(OffDq   - tau*OffD[p]);
                for (j=Z; j>=X; j--) {
                    a = U[j][p]; b = U[j][q];
                    U[j][p] -= s*(b + tau*a);
                    U[j][q] += s*(a - tau*b);
                }
            }
        }
    }
    kv.x = Diag[X]; kv.y = Diag[Y]; kv.z = Diag[Z]; kv.w = 1.0;
    return (kv);
}
```

```
/******* Spectral Axis Adjustment *******/

/* Given a unit quaternion, q, and a scale vector, k, find a unit quaternion, p,
 * which permutes the axes and turns freely in the plane of duplicate scale
 * factors, such that q p has the largest possible w component, i.e., the
 * smallest possible angle. Permutes k's components to go with q p instead of q.
 * See Ken Shoemake and Tom Duff, Matrix Animation and Polar Decomposition,
 * Proceedings of Graphics Interface 1992. Details on pp. 262-263.
 */
Quat snuggle(Quat q, HVect *k)
{
#define SQRTHALF (0.7071067811865475244)
#define sgn(n,v)    ((n)?-(v):(v))
#define swap(a,i,j) {a[3]=a[i]; a[i]=a[j]; a[j]=a[3];}
#define cycle(a,p)  if (p) {a[3]=a[0]; a[0]=a[1]; a[1]=a[2]; a[2]=a[3];}\
                    else   {a[3]=a[2]; a[2]=a[1]; a[1]=a[0]; a[0]=a[3];}
    Quat p;
    float ka[4];
    int i, turn = -1;
    ka[X] = k->x; ka[Y] = k->y; ka[Z] = k->z;
    if (ka[X]==ka[Y]) {if (ka[X]==ka[Z]) turn = W; else turn = Z;}
    else {if (ka[X]==ka[Z]) turn = Y; else if (ka[Y]==ka[Z]) turn = X;}
    if (turn>=0) {
        Quat qtoz, qp;
        unsigned neg[3], win;
        double mag[3], c, s, t;
        static Quat qxtoz = {0,SQRTHALF,0,SQRTHALF};
        static Quat qytoz = {SQRTHALF,0,0,SQRTHALF};
        static Quat qppmm = { 0.5, 0.5,-0.5,-0.5};
        static Quat qpppp = { 0.5, 0.5, 0.5, 0.5};
        static Quat qmpmm = {-0.5, 0.5,-0.5,-0.5};
        static Quat qpppm = { 0.5, 0.5, 0.5,-0.5};
        static Quat q0001 = { 0.0, 0.0, 0.0, 1.0};
        static Quat q1000 = { 1.0, 0.0, 0.0, 0.0};
        switch (turn) {
        default: return (Qt_Conj(q));
        case X: q = Qt_Mul(q, qtoz = qxtoz); swap(ka,X,Z) break;
        case Y: q = Qt_Mul(q, qtoz = qytoz); swap(ka,Y,Z) break;
        case Z: qtoz = q0001; break;
        }
        q = Qt_Conj(q);
        mag[0] = (double)q.z*q.z+(double)q.w*q.w-0.5;
        mag[1] = (double)q.x*q.z-(double)q.y*q.w;
        mag[2] = (double)q.y*q.z+(double)q.x*q.w;
        for (i=0; i<3; i++) if (neg[i] = (mag[i]<0.0)) mag[i] = -mag[i];
        if (mag[0]>mag[1]) {if (mag[0]>mag[2]) win = 0; else win = 2;}
        else               {if (mag[1]>mag[2]) win = 1; else win = 2;}
        switch (win) {
        case 0: if (neg[0]) p = q1000; else p = q0001; break;
        case 1: if (neg[1]) p = qppmm; else p = qpppp; cycle(ka,0) break;
        case 2: if (neg[2]) p = qmpmm; else p = qpppm; cycle(ka,1) break;
        }
        qp = Qt_Mul(q, p);
        t = sqrt(mag[win]+0.5);
        p = Qt_Mul(p, Qt_(0.0,0.0,-qp.z/t,qp.w/t));
        p = Qt_Mul(qtoz, Qt_Conj(p));
    } else {
        float qa[4], pa[4];
        unsigned lo, hi, neg[4], par = 0;
        double all, big, two;
        qa[0] = q.x; qa[1] = q.y; qa[2] = q.z; qa[3] = q.w;
        for (i=0; i<4; i++) {
            pa[i] = 0.0;
            if (neg[i] = (qa[i]<0.0)) qa[i] = -qa[i];
            par ^= neg[i];
        }
        /* Find two largest components, indices in hi and lo */
        if (qa[0]>qa[1]) lo = 0; else lo = 1;
        if (qa[2]>qa[3]) hi = 2; else hi = 3;
        if (qa[lo]>qa[hi]) {
            if (qa[lo^1]>qa[hi]) {hi = lo; lo ^= 1;}
            else {hi ^= lo; lo ^= hi; hi ^= lo;}
        } else {if (qa[hi^1]>qa[lo]) lo = hi^1;}
        all = (qa[0]+qa[1]+qa[2]+qa[3])*0.5;
        two = (qa[hi]+qa[lo])*SQRTHALF;
        big = qa[hi];
        if (all>two) {
            if (all>big) {/*all*/
                {int i; for (i=0; i<4; i++) pa[i] = sgn(neg[i], 0.5);}
                cycle(ka,par)
            } else {/*big*/ pa[hi] = sgn(neg[hi],1.0);}
        } else {
            if (two>big) {/*two*/
                pa[hi] = sgn(neg[hi],SQRTHALF); pa[lo] = sgn(neg[lo], SQRTHALF);
                if (lo>hi) {hi ^= lo; lo ^= hi; hi ^= lo;}
                if (hi==W) {hi = "\001\002\000"[lo]; lo = 3-hi-lo;}
                swap(ka,hi,lo)
            } else {/*big*/ pa[hi] = sgn(neg[hi],1.0);}
        }
        p.x = -pa[0]; p.y = -pa[1]; p.z = -pa[2]; p.w = pa[3];
    }
    k->x = ka[X]; k->y = ka[Y]; k->z = ka[Z];
    return (p);
}
```

```
/******* Decompose Affine Matrix *******/

/* Decompose 4x4 affine matrix A as TFRUK(U transpose), where t contains the
 * translation components, q contains the rotation R, u contains U, k contains
 * scale factors, and f contains the sign of the determinant.
 * Assumes A transforms column vectors in right-handed coordinates.
 * See Ken Shoemake and Tom Duff, Matrix Animation and Polar Decomposition,
 * Proceedings of Graphics Interface 1992.
 */
void decomp_affine(HMatrix A, AffineParts *parts)
{
    HMatrix Q, S, U;
    Quat p;
    float det;
    parts->t = Qt_(A[X][W], A[Y][W], A[Z][W], 0);
    det = polar_decomp(A, Q, S);
    if (det<0.0) {
        mat_copy(Q,=,-Q,3);
        parts->f = -1;
    } else parts->f = 1;
    parts->q = Qt_FromMatrix(Q);
    parts->k = spect_decomp(S, U);
    parts->u = Qt_FromMatrix(U);
    p = snuggle(parts->u, &parts->k);
    parts->u = Qt_Mul(parts->u, p);
}

/******* Invert Affine Decomposition *******/

/* Compute inverse of affine decomposition.
 */
void invert_affine(AffineParts *parts, AffineParts *inverse)
{
    Quat t, p;
    inverse->f = parts->f;
    inverse->q = Qt_Conj(parts->q);
    inverse->u = Qt_Mul(parts->q, parts->u);
    inverse->k.x = (parts->k.x==0.0) ? 0.0 : 1.0/parts->k.x;
    inverse->k.y = (parts->k.y==0.0) ? 0.0 : 1.0/parts->k.y;
    inverse->k.z = (parts->k.z==0.0) ? 0.0 : 1.0/parts->k.z;
    inverse->k.w = parts->k.w;
    t = Qt_(-parts->t.x, -parts->t.y, -parts->t.z, 0);
    t = Qt_Mul(Qt_Conj(inverse->u), Qt_Mul(t, inverse->u));
    t = Qt_(inverse->k.x*t.x, inverse->k.y*t.y, inverse->k.z*t.z, 0);
    p = Qt_Mul(inverse->q, inverse->u);
    t = Qt_Mul(p, Qt_Mul(t, Qt_Conj(p)));
    inverse->t = (inverse->f>0.0) ? t : Qt_(-t.x, -t.y, -t.z, 0);
}
/**** EOF ****/
```

◇ Bibliography ◇

(Goldman 1992) Ronald N. Goldman. Decomposing linear and affine transformations. In David Kirk, editor, *Graphics Gems III*, pages 108–116. Academic Press, Boston, 1992.

(Golub and Van Loan 1989) Gene H. Golub and Charles F. Van Loan. *Matrix Computations*, second edition. Johns Hopkins University Press, Baltimore, 1989.

(Higham and Schreiber 1988) Nicholas Higham and Robert S. Schreiber. Fast polar decomposition of an arbitrary matrix. Technical Report 88–942, Department of Computer Science, Cornell University, October 1988.

(Raible 1990) Eric Raible. Decomposing a matrix into simple transformations. In Andrew Glassner, editor, *Graphics Gems*, page 464. Academic Press, Boston, 1990.

(Shoemake and Duff 1992) Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. In *Proceedings of Graphics Interface '92*, pages 258–264, 1992.

(Thomas 1991) Spencer W. Thomas. Decomposing a matrix into simple transformations. In James Arvo, editor, *Graphics Gems II*, pages 320–323. Academic Press, Boston, 1991.