Constrained Quaternions Using Euler Angles

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: July 29, 2008

Contents

1 Introduction

I assume that you are familiar with quaternions, their algebraic properties, and how they represent rotations and orientations. A brief reminder of notation is in order. Unit-length quaternions are of the form

$$\hat{q} = w + x\hat{\imath} + y\hat{\jmath} + z\hat{k} \tag{1}$$

where $w^2 + x^2 + y^2 + z^2 = 1$. The quaternion may be written as a 4-tuple

$$\mathbf{q} = (w, x, y, z) = (q_0, q_1, q_2, q_3) \tag{2}$$

The last equality shows how the components of the quaternion are indexed within an array of four real numbers. The quaternion $\hat{q}$ may be written also as

$$\hat{q} = \cos(\theta) + \sin(\theta)\hat{u} \tag{3}$$

where $\hat{u} = u_0\hat{\imath} + u_1\hat{\jmath} + u_2\hat{k}$ with $\mathbf{u} = (u_0, u_1, u_2)$, a unit-length vector. The corresponding rotation matrix R has $\mathbf{u}$ as the direction of the axis of rotation and 2θ as the angle of rotation about that axis. If $\mathbf{v} = (v_0, v_1, v_2)$ is a vector to be rotated, define $\hat{v} = v_0\hat{\imath} + v_1\hat{\jmath} + v_2\hat{k}$. The rotated vector $R\mathbf{v} = (r_0, r_1, r_2)$ is generated by defining $\hat{r} = r_0\hat{\imath} + r_1\hat{\jmath} + r_2\hat{k}$ and computing the quaternion product $\hat{r} = \hat{q}\hat{v}\hat{q}^*$, where

$$\hat{q}^* = w - x\hat{\imath} - y\hat{\jmath} - z\hat{k} \tag{4}$$

is the conjugate of $\hat{q}$. The quaternion $-\hat{q}$ represents the same rotation matrix as $\hat{q}$ because

$$-\hat{q} = -\cos(\theta) - \sin(\theta)\hat{u} = \cos(\theta + \pi) + \sin(\theta + \pi)\hat{u} \tag{5}$$

The rotation axis has direction $\mathbf{u}$, the same as that corresponding to $\hat{q}$. The rotation angle is $2(\theta+\pi) = 2\theta+2\pi$. The rotation angle for $\hat{q}$ is 2θ. The addition of 2π does not change the results of the rotation.

Also, the convention used here is that matrix-vector multiplication has the order $M\mathbf{v}$; that is, the matrix is on the left and the vector is on the right. The implication is that a product of quaternions $\hat{p}\hat{q}$ represents a product of rotations $PQ\mathbf{v}$. The $\hat{q}$ is applied first and the $\hat{p}$ is applied second.

The orientation of an object is essentially a right-handed coordinate system assigned to that object. If $\mathbf{U}_0$, $\mathbf{U}_1$, and $\mathbf{U}_2$ are unit-length vectors that are mutually perpendicular and for which $\mathbf{U}_2 = \mathbf{U}_0 \times \mathbf{U}_1$, the matrix $R = [\mathbf{U}_0\,\mathbf{U}_1\,\mathbf{U}_2]$ whose columns are the specified vectors is a rotation matrix and may be represented by a quaternion.

This document is about specifying a quaternion $\hat{p}$ and constructing a quaternion $\hat{q}$ that is closest to $\hat{p}$, where the candidates for $\hat{q}$ are constrained in some manner. The closeness is measured by the angle between the corresponding 4-tuples, $\mathbf{p}$ and $\mathbf{q}$. The constraints are Euler-angle constraints for the rotation matrices represented by $\hat{q}$.

2 The Unconstrained Problem

2.1 Rotation X (The Analysis)

The quaternion that represents a rotation about the x-axis by an angle 2θ is

$$\hat{q}(\theta) = \cos(\theta) + \sin(\theta)\hat{\imath} \tag{6}$$

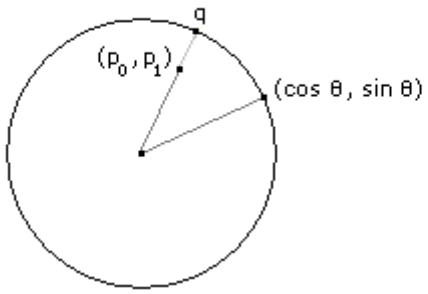
Given a quaternion $\hat{p} = p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k}$, we wish to compute the angle $\theta \in [-\pi, \pi]$ for which $\hat{q}(\theta)$ is the quaternion closest to $\hat{p}$ on the unit hypersphere in 4D. This problem is equivalent to selecting θ for which the angle between the 4-tuples $\mathbf{p}$ and $\mathbf{q}(\theta)$ is a minimum. This is equivalent to maximizing the dot product

$$f(\theta) = \mathbf{p} \cdot \mathbf{q}(\theta) = (p_0, p_1, p_2, p_3) \cdot (\cos(\theta), \sin(\theta), 0, 0) = p_0 \cos(\theta) + p_1 \sin(\theta) \tag{7}$$

because the dot product is the cosine of the angle between the vectors, and the angle is minimized when the cosine is largest.

The dot product may be thought of as the dot product of 2-tuples, say, $f(\theta) = (p_0, p_1) \cdot (\cos(\theta), \sin(\theta))$. Figure 2.1 illustrates when $(p_0, p_1) \neq (0, 0)$.

Figure 2.1 The geometric relationship between (p_0, p_1) and $(\cos(\theta), \sin(\theta))$.



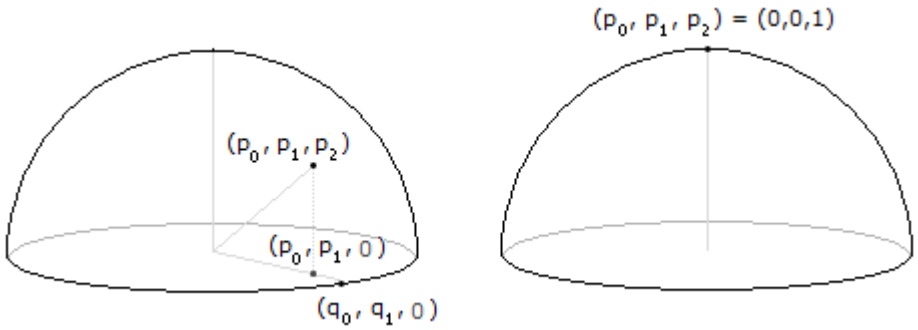
The maximum dot product occurs when (p_0, p_1) and $(\cos(\theta), \sin(\theta))$ are parallel and pointing in the same direction; that is,

$$(\cos(\theta), \sin(\theta)) = \frac{(p_0, p_1)}{\sqrt{p_0^2 + p_1^2}} \tag{8}$$

which immediately tells you the components of $\mathbf{q}(\theta)$. The maximum dot product is $f_{\max} = \sqrt{p_0^2 + p_1^2}$. If you want to know the angle, it may be computed numerically as $\theta = \text{atan2}(p_1, p_0)$.

When $(p_0, p_1) = (0, 0)$, $f(\theta)$ is identically zero, in which case all $\mathbf{q}(\theta)$ are equally close to $\mathbf{p}$. Figure 2.2 illustrates this in 3D, say, by ignoring the $\mathbf{k}$ component and thinking of $\mathbf{p} = (p_0, p_1, p_2)$.

Figure 2.2 Left: The case when $(p_0, p_1) \neq (0, 0)$. There is a unique closest quaternion on the circle. Right: The case when $(p_0, p_1) = (0, 0)$, illustrated by $(p_0, p_1, p_2) = (0, 0, 1)$. All quaternions on the circle are closest.



Pseudocode for computing the closest quaternion is listed next.

```
Quaternion ClosestX (Quaternion p)
{
    Quaternion q;
    q[2] = 0;
    q[3] = 0;

    float sqrLength = p[0]*p[0] + p[1]*p[1];
    if (sqrLength > 0)
    {
        // A unique closest point.
        float invLength = 1/sqrt(sqrLength);
        q[0] = p[0]*invLength;
        q[1] = p[1]*invLength;
    }
    else
    {
        // Infinitely many solutions, choose the identity (theta = 0).
        q[0] = 1;
        q[1] = 0;
    }

    return q;
}
```

2.2 Rotation X, Y, or Z

Analyses similar to that for quaternions representing rotations about the x-axis may be done for rotations about the y-axis or z-axis. The pseudocode is nearly identical. Using indices to denote axes, we may use a single function.

```
Quaternion Closest (Quaternion p, int axis)  // 1 (x-axis), 2 (y-axis), 3 (z-axis)
{
    // The appropriate nonzero components will be set later.
    Quaternion q(0,0,0,0);

    float sqrLength = p[0]*p[0] + p[axis]*p[axis];
```

```
    if (sqrLength > 0)
    {
        // A unique closest point.
        float invLength = 1/sqrt(sqrLength);
        q[0] = p[0]*invLength;
        q[axis] = p[axis]*invLength;
    }
    else
    {
        // Infinitely many solutions, choose the identity (theta = 0).
        q[0] = 1;
    }

    return q;
}
```

2.3 Rotation XY (The Analysis)

The quaternion that represents a product of rotations, one about the y-axis by an angle $2\theta_1$ and one about the x-axis by an angle $2\theta_0$, is

$$\hat{q}(\theta_0, \theta_1) = (c_0 + s_0\hat{\imath})(c_1 + s_1\hat{\jmath}) = c_0c_1 + s_0c_1\hat{\imath} + c_0s_1\hat{\jmath} + s_0s_1\hat{k} \tag{9}$$

where $c_\ell = \cos(\theta_\ell)$ and $s_\ell = \sin(\theta_\ell)$ for $\ell = 0, 1$.

Given a quaternion $\hat{p} = p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k}$, we wish to compute the angles $\theta_0 \in [-\pi, \pi]$ and $\theta_1 \in [-\pi, \pi]$ for which $\hat{q}(\theta_0, \theta_1)$ is the quaternion closest to $\hat{p}$ on the unit hypersphere in 4D. This problem is equivalent to selecting θ_0 and θ_1 for which the angle between the 4-tuples $\mathbf{p}$ and $\mathbf{q}(\theta_0, \theta_1)$ is a minimum. This is equivalent to maximizing the dot product

$$f(\theta_0, \theta_1) = \mathbf{p} \cdot \mathbf{q}(\theta_0, \theta_1) = p_0c_0c_1 + p_1s_0c_1 + p_2c_0s_1 + p_3s_0s_1 \tag{10}$$

Notice that

$$f = \begin{bmatrix} c_1 & s_1 \end{bmatrix} \begin{bmatrix} p_0 & p_1 \\ p_2 & p_3 \end{bmatrix} \begin{bmatrix} c_0 \\ s_0 \end{bmatrix} = \mathbf{u}_1^{\mathrm{T}} P \mathbf{u}_0 \tag{11}$$

where the last equality defines the 2×1 vectors $\mathbf{u}_0$ and $\mathbf{u}_1$ and the 2×2 matrix P. The value of f is maximized when the vectors $\mathbf{u}_1$ and $P\mathbf{u}_0$ are parallel and in the same direction. Thus, we need

$$\mathbf{u}_1 = \frac{P\mathbf{u}_0}{|P\mathbf{u}_0|} \tag{12}$$

in which case the maximum of f and its square are

$$f_{\max} = |P\mathbf{u}_0|, \quad f_{\max}^2 = \mathbf{u}_0^{\mathrm{T}} P^T P \mathbf{u}_0 \tag{13}$$

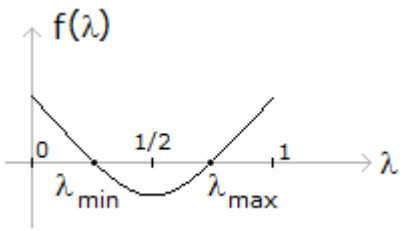
We still need to compute a vector $\mathbf{u}_0$ for which $f_{\max}^2$ is maximized. This is the classical problem of maximizing a quadratic form involving a unit-length vector and, in our case, one whose symmetric matrix is $P^{\mathrm{T}}P$. The maximum occurs when $\mathbf{u}_0$ is a unit-length eigenvector of $P^{\mathrm{T}}P$ corresponding to the maximum eigenvalue $\lambda_{\max}$ of the matrix.

The characteristic polynomial of $P^{\mathrm{T}}P$ is

$$0 = \det(P^{\mathrm{T}}P - \lambda I) = \lambda^2 - \lambda + (p_0p_3 - p_1p_2)^2 = f(\lambda) \tag{14}$$

The graph of this polynomial is shown in Figure 2.3.

Figure 2.3 The graph of the characteristic polynomial $f(\lambda)$ of $P^{\mathrm{T}}P$.



The eigenvalues are necessarily in the interval $[0, 1]$. The maximum eigenvalue is

$$\lambda_{\max} = \frac{1 + \sqrt{1 - 4(p_0 p_3 - p_1 p_2)^2}}{2} \tag{15}$$

2.3.1 Distinct Eigenvalues

When $1 - 4(p_0 p_3 - p_1 p_2)^2 > 0$, the eigenvalues are distinct. Observe that

$$P^{\mathrm{T}}P - \lambda_{\max} I = \begin{bmatrix} p_0^2 + p_2^2 - \lambda_{\max}^2 & p_0 p_1 + p_2 p_3 \\ p_0 p_1 + p_2 p_3 & p_1^2 + p_3^2 - \lambda_{\max}^2 \end{bmatrix} \tag{16}$$

An eigenvector corresponding to $\lambda_{\max}$ is perpendicular to the nonzero rows of this matrix, so the candidate eigenvectors are

$$\begin{bmatrix} p_0 p_1 + p_2 p_3 \\ \lambda_{\max} - (p_0^2 + p_2^2) \end{bmatrix}, \quad \begin{bmatrix} \lambda_{\max} - (p_1^2 + p_3^2) \\ p_0 p_1 + p_2 p_3 \end{bmatrix} \tag{17}$$

For numerical robustness, choose that vector with the larger of the two components involving $\lambda_{\max}$. Using $1 = p_0^2 + p_1^2 + p_2^2 + p_3^2$,

$$\lambda_{\max} - (p_0^2 + p_2^2) = \frac{-(p_0^2 - p_1^2 + p_2^2 - p_3^2) + \sqrt{1 - 4(p_0 p_3 - p_1 p_2)^2}}{2} \tag{18}$$

and

$$\lambda_{\max} - (p_1^2 + p_3^2) = \frac{p_0^2 - p_1^2 + p_2^2 - p_3^2 + \sqrt{1 - 4(p_0 p_3 - p_1 p_2)^2}}{2} \tag{19}$$

Choose the vector with the component of Equation (18) when $p_0^2 - p_1^2 + p_2^2 - p_3^2 < 0$; otherwise, choose the vector with the compoonent of Equation (19).

Once you have chosen the vector, normalize it to produce $\mathbf{u}_0 = (c_0, s_0) = (\cos\theta_0, \sin\theta_0)$. The value of θ_0 is

$$\theta_0 = \operatorname{atan2}(s_0, c_0) \tag{20}$$

Substitute $\mathbf{u}_0$ in Equation (12) to obtain $\mathbf{u}_1 = (c_1, s_1) = (\cos\theta_1, \sin\theta_1)$. The value of θ_1 is

$$\theta_1 = \operatorname{atan2}(s_1, c_1) \tag{21}$$

The combined angles produce the closest quaternion $\hat{q}$. We could just as well have chosen $-\mathbf{u}_0$ and $-\mathbf{u}_1$, but this leads only to $-\hat{q}$ which represents the same rotation.

2.3.2 Equal Eigenvalues

When $1 - 4(p_0p_3 - p_1p_2)^2 = 0$, the eigenvalues are the same value, namely, $\lambda = 1/2$. The matrix $P^{\mathrm{T}}P$ is necessary diagonal; specifically, it is half of the identity matrix,

$$\begin{bmatrix} p_0^2 + p_2^2 & p_0p_1 + p_2p_3 \\ p_0p_1 + p_2p_3 & p_1^2 + p_3^2 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{22}$$

and the following conditions are true

$$\begin{aligned} |(p_0, p_2)|^2 &= p_0^2 + p_2^2 = 1/2 \\ |(p_1, p_3)|^2 &= p_1^2 + p_3^2 = 1/2 \\ (p_0, p_2) \cdot (p_1, p_3) &= p_0p_1 + p_2p_3 = 0 \end{aligned} \tag{23}$$

These conditions in turn imply that the 2-tuples (p_0, p_2) and (p_1, p_3) have the same length and are perpendicular. The only possibilities for such vectors is

$$(p_1, p_3) = \pm(p_2, -p_0) \tag{24}$$

If $(p_1, p_3) = (p_2, -p_0)$, then $\hat{p} = (p_0, p_1, p_1, -p_0)$ and Equation (10) factors into

$$f(\theta_0, \theta_1) = p_0 \cos(\theta_0 + \theta_1) + p_1 \sin(\theta_0 + \theta_1) \tag{25}$$

The maximum occurs when

$$(\cos(\theta_0 + \theta_1), \sin(\theta_0 + \theta_1)) = \frac{(p_0, p_1)}{\sqrt{p_0^2 + p_1^2}} \tag{26}$$

and it must be that

$$\theta_0 + \theta_1 = \operatorname{atan2}(p_1, p_0) \tag{27}$$

There are infinitely many pairs (θ_0, θ_1) that satisfy this equation, so there are infinitely many $\hat{q}$ that are equally close to $\hat{p}$. Geometrically, the solution set is a curve on the unit hypersphere in four dimensions.

If $(p_1, p_3) = (-p_2, p_0)$, then $\hat{p} = (p_0, p_1, -p_1, p_0)$ and Equation (10) factors into

$$f(\theta_0, \theta_1) = p_0 \cos(\theta_0 - \theta_1) + p_1 \sin(\theta_0 - \theta_1) \tag{28}$$

The maximum occurs when

$$(\cos(\theta_0 - \theta_1), \sin(\theta_0 - \theta_1)) = \frac{(p_0, p_1)}{\sqrt{p_0^2 + p_1^2}} \tag{29}$$

and it must be that

$$\theta_0 - \theta_1 = \operatorname{atan2}(p_1, p_0) \tag{30}$$

There are infinitely many pairs (θ_0, θ_1) that satisfy this equation, so there are infinitely many $\hat{q}$ that are equally close to $\hat{p}$. Geometrically, the solution set is a curve on the unit hypersphere in four dimensions.

Observe that $(p_1, p_3) = (p_2, -p_0)$ implies $p_0p_3 - p_1p_2 = -(p_0^2 + p_1^2) < 0$ and $(p_1, p_3) = (-p_2, p_0)$ implies $p_0p_3 - p_1p_2 = p_0^2 + p_1^2 > 0$. We will use this in the pseudocode for the constrained problem in order to distinguish between the two cases. However, for the unconstrained problem, as long as the caller wants only one of the infinite solutions, we may as well choose $\theta_1 = 0$. The consequence is that θ_0 is chosen the same way regardless of the sign of $p_0p_3 - p_1p_2$.

```
Quaternion ClosestQuaternionXY (Quaternion p)
{
    Quaternion q;

    float det = p[0]*p[3] - p[1]*p[2];
    if(|det| < 0.5)
    {
        // The case of distinct eigenvalues.  The solution is unique.
        float discr = sqrt(1 - 4*det*det));
        float a = p[0]*p[1] + p[2]*p[3];
        float b = p[0]*p[0] - p[1]*p[1] + p[2]*p[2] - p[3]*p[3];
        float c0, s0, c1, s1, invLength;

        if (b >= 0)
        {
            c0 = 0.5*(discr + b);
            s0 = a;
        }
        else
        {
            c0 = a;
            s0 = 0.5*(discr - b);
        }
        invLength = 1/sqrt(c0*c0 + s0*s0);
        c0 *= invLength;
        s0 *= invLength;

        c1 = p[0]*c0 + p[1]*s0;
        s1 = p[2]*c0 + p[3]*s0;
        invLength = 1/sqrt(c1*c1 + s1*s1);
        c1 *= invLength;
        s1 *= invLength;

        q[0] = c0*c1;
        q[1] = s0*c1;
        q[2] = c0*s1;
        q[3] = s0*s1;
    }
    else
    {
        // The case of equal eigenvalues.  There are infinitely many
        // solutions.  Choose theta1 = 0.
        float invLength = 1/sqrt(|det|);
        q[0] = p[0]*invLength;
        q[1] = p[1]*invLength;
        q[2] = 0;
        q[3] = 0;
    }

    return q;
}
```

2.4 Rotation XY, YX, YZ, ZY, ZX, XZ

The code for rotation XY may be reused for the other pairs of rotations. The idea is to compare the dot-product function $f(\theta_0, \theta_1)$ for each pair with that of Equation (10), looking for permutations and sign changes of the components of $\hat{p}$ and $\hat{q}$.

2.4.1 Rotation YX

The quaternion that represents a product of rotations, one about the x-axis by an angle $2\theta_0$ and one about the y-axis by an angle $2\theta_1$, is

$$\hat{q}(\theta_0,\theta_1) = (c_1 + s_1\hat{\jmath})(c_0 + s_0\hat{\imath}) = c_0c_1 + s_0c_1\hat{\imath} + c_0s_1\hat{\jmath} - s_0s_1\hat{k} \tag{31}$$

where $c_\ell = \cos(\theta_\ell)$ and $s_\ell = \sin(\theta_\ell)$ for $\ell = 0, 1$.

Given a quaternion $\hat{p} = p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k}$, we wish to compute the angles $\theta_0 \in [-\pi, \pi]$ and $\theta_1 \in [-\pi, \pi]$ for which $\hat{q}(\theta_0, \theta_1)$ is the quaternion closest to $\hat{p}$ on the unit hypersphere in 4D. This problem is equivalent to selecting θ_0 and θ_1 for which the angle between the 4-tuples $\mathbf{p}$ and $\mathbf{q}(\theta_0, \theta_1)$ is a minimum. This is equivalent to maximizing the dot product

$$\begin{aligned} f(\theta_0,\theta_1) &= \mathbf{p}\cdot\mathbf{q}(\theta_0,\theta_1) \\ &= p_0c_0c_1 + p_1s_0c_1 + p_2c_0s_1 - p_3s_0s_1 \end{aligned} \tag{32}$$

Equation (32) is similar to Equation (10) except that the sign of the s_0s_1 term is negated. Therefore, we may use the XY code for the YX code but with the p_3 component of $\hat{p}$ negated on input and the q_3 term of $\hat{q}$ negated on output.

```
Quaternion ClosestQuaternionYX (Quaternion p)
{
    Quaternion q = ClosestQuaternionXY(Quaternion(p[0],p[1],p[2],-p[3]));
    q[3] = -q[3];
    return q;
}
```

2.4.2 Rotation ZX

The quaternion that represents a product of rotations, one about the x-axis by an angle $2\theta_0$ and one about the z-axis by an angle $2\theta_1$, is

$$\hat{q}(\theta_0,\theta_1) = (c_1 + s_1\hat{k})(c_0 + s_0\hat{\imath}) = c_0c_1 + s_0c_1\hat{\imath} + s_0s_1\hat{\jmath} + c_0s_1\hat{k} \tag{33}$$

where $c_\ell = \cos(\theta_\ell)$ and $s_\ell = \sin(\theta_\ell)$ for $\ell = 0, 1$.

Given a quaternion $\hat{p} = p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k}$, we wish to compute the angles $\theta_0 \in [-\pi, \pi]$ and $\theta_1 \in [-\pi, \pi]$ for which $\hat{q}(\theta_0, \theta_1)$ is the quaternion closest to $\hat{p}$ on the unit hypersphere in 4D. This problem is equivalent to selecting θ_0 and θ_1 for which the angle between the 4-tuples $\mathbf{p}$ and $\mathbf{q}(\theta_0, \theta_1)$ is a minimum. This is equivalent to maximizing the dot product

$$\begin{aligned} f(\theta_0,\theta_1) &= \mathbf{p}\cdot\mathbf{q}(\theta_0,\theta_1) \\ &= p_0c_0c_1 + p_1s_0c_1 + p_2s_0s_1 + p_3c_0s_1 \\ &= p_0c_0c_1 + p_1s_0c_1 + p_3c_0s_1 + p_2s_0s_1 \end{aligned} \tag{34}$$

Notice that the expression of the last equality has been written so that the sine and cosine terms appear in the same order as those of Equation (10). The difference is that components p_2 and p_3 have been swapped. The pseudocode for this case is therefore

```
Quaternion ClosestQuaternionZX(Quaternion p)
{
    Quaternion q = ClosestQuaternionXY(Quaternion(p[0],p[1],p[3],p[2]));
    float save = q[2];
    q[2] = q[3];
    q[3] = save;
    return q;
}
```

2.4.3 Rotation XZ

The quaternion that represents a product of rotations, one about the z-axis by an angle $2\theta_1$ and one about the x-axis by an angle $2\theta_0$, is

$$\hat{q}(\theta_0,\theta_1) = (c_0 + s_0\hat{\imath})(c_1 + s_1\hat{k}) = c_0c_1 + s_0c_1\hat{\imath} - s_0s_1\hat{\jmath} + c_0s_1\hat{k} \tag{35}$$

where $c_\ell = \cos(\theta_\ell)$ and $s_\ell = \sin(\theta_\ell)$ for $\ell = 0, 1$.

Given a quaternion $\hat{p} = p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k}$, we wish to compute the angles $\theta_0 \in [-\pi, \pi]$ and $\theta_1 \in [-\pi, \pi]$ for which $\hat{q}(\theta_0, \theta_1)$ is the quaternion closest to $\hat{p}$ on the unit hypersphere in 4D. This problem is equivalent to selecting θ_0 and θ_1 for which the angle between the 4-tuples $\mathbf{p}$ and $\mathbf{q}(\theta_0, \theta_1)$ is a minimum. This is equivalent to maximizing the dot product

$$\begin{aligned} f(\theta_0,\theta_1) &= \mathbf{p}\cdot\mathbf{q}(\theta_0,\theta_1) \\ &= p_0c_0c_1 + p_1s_0c_1 - p_2s_0s_1 + p_3c_0s_1 \\ &= p_0c_0c_1 + p_1s_0c_1 + p_3c_0s_1 - p_2s_0s_1 \end{aligned} \tag{36}$$

Notice that the expression of the last equality has been written so that the sine and cosine terms appear in the same order as those of Equation (10). The difference is that components p_2 and p_3 have been swapped and the p_2 term has been negated. The pseudocode for this case is therefore

```
Quaternion ClosestQuaternionXZ(Quaternion p)
{
    Quaternion q = ClosestQuaternionXY(Quaternion(p[0],p[1],-p[3],p[2]));
    float save = q[2];
    q[2] = q[3];
    q[3] = -save;
    return q;
}
```

2.4.4 Rotation YZ

The quaternion that represents a product of rotations, one about the z-axis by an angle $2\theta_1$ and one about the y-axis by an angle $2\theta_0$, is

$$\hat{q}(\theta_0,\theta_1) = (c_0 + s_0\hat{\jmath})(c_1 + s_1\hat{k}) = c_0c_1 + s_0s_1\hat{\imath} + s_0c_1\hat{\jmath} + c_0s_1\hat{k} \tag{37}$$

where $c_\ell = \cos(\theta_\ell)$ and $s_\ell = \sin(\theta_\ell)$ for $\ell = 0, 1$.

Given a quaternion $\hat{p} = p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k}$, we wish to compute the angles $\theta_0 \in [-\pi, \pi]$ and $\theta_1 \in [-\pi, \pi]$ for which $\hat{q}(\theta_0, \theta_1)$ is the quaternion closest to $\hat{p}$ on the unit hypersphere in 4D. This problem is equivalent to

selecting θ_0 and θ_1 for which the angle between the 4-tuples $\mathbf{p}$ and $\mathbf{q}(\theta_0,\theta_1)$ is a minimum. This is equivalent to maximizing the dot product

$$\begin{aligned} f(\theta_0,\theta_1) &= \mathbf{p}\cdot\mathbf{q}(\theta_0,\theta_1) \\ &= p_0c_0c_1 + p_1s_0s_1 + p_2s_0c_1 + p_3c_0s_1 \\ &= p_0c_0c_1 + p_2s_0c_1 + p_3c_0s_1 + p_1s_0s_1 \end{aligned} \tag{38}$$

Notice that the expression of the last equality has been written so that the sine and cosine terms appear in the same order as those of Equation (10). The difference is that components (p_1,p_2,p_3) have been permuted to (p_2,p_3,p_1). The pseudocode for this case is therefore

```
Quaternion ClosestQuaternionYZ(Quaternion p)
{
    Quaternion q = ClosestQuaternionXY(Quaternion(p[0],p[2],p[3],p[1]));
    float save = q[3];
    q[3] = q[2];
    q[2] = q[1];
    q[1] = save;
    return q;
}
```

2.4.5 Rotation ZY

The quaternion that represents a product of rotations, one about the y-axis by an angle $2\theta_0$ and one about the z-axis by an angle $2\theta_1$, is

$$\hat{q}(\theta_0,\theta_1) = (c_1 + s_1\hat{k})(c_0 + s_0\hat{\jmath}) = c_0c_1 - s_0s_1\hat{\imath} + s_0c_1\hat{\jmath} + c_0s_1\hat{k} \tag{39}$$

where $c_\ell = \cos(\theta_\ell)$ and $s_\ell = \sin(\theta_\ell)$ for $\ell = 0, 1$.

Given a quaternion $\hat{p} = p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k}$, we wish to compute the angles $\theta_0 \in [-\pi,\pi]$ and $\theta_1 \in [-\pi,\pi]$ for which $\hat{q}(\theta_0,\theta_1)$ is the quaternion closest to $\hat{p}$ on the unit hypersphere in 4D. This problem is equivalent to selecting θ_0 and θ_1 for which the angle between the 4-tuples $\mathbf{p}$ and $\mathbf{q}(\theta_0,\theta_1)$ is a minimum. This is equivalent to maximizing the dot product

$$\begin{aligned} f(\theta_0,\theta_1) &= \mathbf{p}\cdot\mathbf{q}(\theta_0,\theta_1) \\ &= p_0c_0c_1 - p_1s_0s_1 + p_2s_0c_1 + p_3c_0s_1 \\ &= p_0c_0c_1 + p_2s_0c_1 + p_3c_0s_1 - p_1s_0s_1 \end{aligned} \tag{40}$$

Notice that the expression of the last equality has been written so that the sine and cosine terms appear in the same order as those of Equation (10). The difference is that the p_1 component is negated and the components $(-p_1,p_2,p_3)$ have been permuted to $(p_2,p_3,-p_1)$. The pseudocode for this case is therefore

```
Quaternion ClosestQuaternionZY(Quaternion p)
{
    Quaternion q = ClosestQuaternionXY(Quaternion(p[0],p[2],p[3],-p[1]));
    float save = q[3];
    q[3] = q[2];
    q[2] = q[1];
    q[1] = -save;
    return q;
}
```

2.5 Rotation XYZ (The Analysis)

The unconstrained problem is equivalent to factoring a quaternion into a product of three coordinate-axis quaternions. In the case of XYZ rotation, we want to factor the quaternion $\hat{p}$ as

$$\begin{aligned}\hat{p} &= p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k} \\ &= (c_0 + s_0\hat{\imath})(c_1 + s_1\hat{\jmath})(c_2 + s_2\hat{k}) \\ &= (c_0c_1c_2 - s_0s_1s_2) + (s_0c_1c_2 + c_0s_1s_2)\hat{\imath} + (c_0s_1c_2 - s_0c_1s_2)\hat{\jmath} + (s_0s_1c_2 + c_0c_1s_2)\hat{k}\end{aligned} \tag{41}$$

where $c_\ell = \cos(\theta_\ell)$ and $s_\ell = \sin(\theta_\ell)$ for $\ell = 0, 1, 2$. It is important to observe that there *never* is a unique solution. For example, negating two of the three coordinate-axis quaternions is allowed:

$$(c_0 + s_0\hat{\imath})(c_1 + s_1\hat{\jmath})(c_2 + s_2\hat{k}) = (-c_0 - s_0\hat{\imath})(-c_1 - s_1\hat{\jmath})(c_2 + s_2\hat{k}) \tag{42}$$

In terms of rotations, $c_0 + s_0\hat{\imath}$ represents a rotation about the x-axis by an angle $2\theta_0$. Because $\cos(\theta_0 + \pi) = -\cos(\theta_0)$ and $\sin(\theta_0 + \pi) = -\sin(\theta)$, $-c_0 - s_0\hat{\imath}$ represents a rotation about the x-axis by an angle $2\theta_0 + 2\pi$, which is not different from a rotation by an angle $2\theta_0$.

A less obvious but equivalent factorization is

$$\begin{aligned}(c_0 + s_0\hat{\imath})(c_1 + s_1\hat{\jmath})(c_2 + s_2\hat{k}) &= (s_0 - c_0\hat{\imath})\hat{\imath}(c_1 + s_1\hat{\jmath})\hat{k}(s_2 - c_2\hat{k}) \\ &= (s_0 - c_0\hat{\imath})(c_1\hat{\imath} + s_1\hat{k})\hat{k}(s_2 - c_2\hat{k}) \\ &= (s_0 - c_0\hat{\imath})(-s_1 - c_1\hat{\jmath})(s_2 - c_2\hat{k})\end{aligned} \tag{43}$$

The left-hand side of the equations represents rotations of $2\theta_0$ about the x-axis, $2\theta_1$ about the y-axis, and $2\theta_2$ about the z-axis. Note that $\cos(\phi - \pi/2) = \sin(\phi)$ and $\sin(\phi - \pi/2) = -\cos(\phi)$. The right-hand side of the equations involves rotation by $2\theta_0 - \pi$ about the x-axis and by $2\theta_2 - \pi$ about the z-axis. Also note that $\cos(-\theta_1 - \pi/2) = -\sin(\theta_1)$ and $\sin(-\theta_1 - \pi/2) = -\cos(\theta_1)$. The right-hand side of the equations involves a rotation by $-2\theta_1 - \pi$ about the y-axis. Define $\gamma_i = \cos(2\theta_i)$ and $\sigma_i = \sin(2\theta_i)$. The product of rotation matrices corresponding to the left-hand side of the equations is

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \gamma_0 & -\sigma_0 \\ 0 & \sigma_0 & \gamma_0 \end{bmatrix} \begin{bmatrix} \gamma_1 & 0 & \sigma_1 \\ 0 & 1 & 0 \\ -\sigma_1 & 0 & \gamma_1 \end{bmatrix} \begin{bmatrix} \gamma_2 & -\sigma_2 & 0 \\ \sigma_2 & \gamma_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{44}$$

Using $\cos(2\phi - \pi) = -\cos(2\phi)$, $\sin(2\phi - \pi) = -\sin(2\phi)$, $\cos(-2\phi - \pi) = -\cos(2\phi)$, and $\sin(-2\phi - \pi) = \sin(2\phi)$, the product of rotation matrices corresponding to the right-hand side of the equations is

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\gamma_0 & \sigma_0 \\ 0 & -\sigma_0 & -\gamma_0 \end{bmatrix} \begin{bmatrix} -\gamma_1 & 0 & \sigma_1 \\ 0 & 1 & 0 \\ -\sigma_1 & 0 & -\gamma_1 \end{bmatrix} \begin{bmatrix} -\gamma_2 & \sigma_2 & 0 \\ -\sigma_2 & -\gamma_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{45}$$

Some matrix algebra will show that these two rotation matrices are the same. At first glance, this result appears to contradict the uniqueness of Euler-angle factorizations mentioned in my document `EulerAngles.pdf` for the case of XYZ factorization. That factorization involved computing the inverse-sine function for the

y-angle, and the inversion is required to be in the interval $[-\pi/2, \pi/2]$ (the principal branch of the inverse-sine). If you relax the restriction and choose angles outside the interval, you can generate multiple Euler-angle factorizations.

Effectively, Equation(41) provides 8 equations in the 8 unknowns c_ℓ and s_ℓ,

$$\begin{aligned} c_0c_1c_2 - s_0s_1s_2 &= p_0, \qquad & c_0^2 + s_0^2 &= 1 \\ s_0c_1c_2 + c_0s_1s_2 &= p_1, \qquad & c_1^2 + s_1^2 &= 1 \\ c_0s_1c_2 - s_0c_1s_2 &= p_2, \qquad & c_2^2 + s_2^2 &= 1 \\ s_0s_1c_2 + c_0c_1s_2 &= p_3, \qquad & c_3^2 + s_3^2 &= 1 \end{aligned} \tag{46}$$

We may solve for s_0 and c_0 as follows. Multiplying the equations involving p_0 and p_1 by c_0 and s_0, and multiplying the equations involving p_2 and p_3 similarly,

$$\begin{aligned} c_0p_0 + s_0p_1 &= c_1c_2, \qquad & c_0p_2 + s_0p_3 &= s_1c_2 \\ -s_0p_2 + c_0p_3 &= c_1s_2, \qquad & -s_0p_0 + c_0p_1 &= s_1s_2 \end{aligned} \tag{47}$$

Combining these, we have

$$\begin{aligned} 0 &= (c_0p_0 + s_0p_1)(-s_0p_0 + c_0p_1) - (c_0p_2 + s_0p_3)(-s_0p_2 + c_0p_3) \\ &= (c_0^2 - s_0^2)(p_0p_1 - p_2p_3) - (s_0c_0)(p_0^2 - p_1^2 - p_2^2 + p_3^2) \\ &= \gamma_0(p_0p_1 - p_2p_3) - \sigma_0(p_0^2 - p_1^2 - p_2^2 + p_3^2)/2 \end{aligned} \tag{48}$$

where $\gamma_0 = \cos(2\theta_0)$ and $\sigma_0 = \sin(2\theta_0)$ with $\gamma_0^2 + \sigma_0^2 = 1$.

2.5.1 Degenerate Linear Equation

Equation (48) is degenerate when $p_0p_1 - p_2p_3 = 0$ and $p_0^2 - p_1^2 - p_2^2 + p_3^2 = 0$. The latter term combined with the unit-length condition $p_0^2 + p_1^2 + p_2^2 + p_3^2 = 1$ implies $p_0^2 + p_3^2 = p_1^2 + p_2^2 = 1/2$. The former term implies (p_0, p_3) and $(p_1, -p_2)$ are perpendicular, in which case (p_0, p_3) and (p_2, p_1) are parallel. The conclusions is that $(p_0, p_3) = \pm(p_2, p_1)$.

If $(p_0, p_3) = (p_2, p_1)$, then Equation (47) reduces to $c_0p_0 + s_0p_1 = c_1c_2 = s_1c_2$ and $-s_0p_0 + c_0p_1 = s_1s_2 = c_1s_2$. It is not possible to have simultaneously $c_2 = 0$ and $s_2 = 0$, so $s_1 = c_1 = \pm 1/\sqrt{2}$. Choosing a sign of -1 leads to the negation of two coordinate quaternions in the factorization, which does not generate a new rotation. It is sufficient to choose the sign $+1$, in which case

$$\begin{bmatrix} c_2 \\ s_2 \end{bmatrix} = \sqrt{2} \begin{bmatrix} c_0 & s_0 \\ -s_0 & c_0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \end{bmatrix} \tag{49}$$

We have no additional conditions, so for each choice of unit-length vector (c_0, s_0) we obtain a corresponding unit-length vector (c_2, s_2). There are infinitely many factorizations of $\hat{p}$, namely,

$$\begin{aligned} \hat{p} &= p_0 + p_1\hat{\imath} + p_0\hat{\jmath} + p_1\hat{k} \\ &= (c_0 + s_0\hat{\imath})\left(\tfrac{1}{\sqrt{2}} + \tfrac{1}{\sqrt{2}}\hat{\jmath}\right)\left(\sqrt{2}(c_0p_0 + s_0p_1) + \sqrt{2}(-s_0p_0 + c_0p_1)\hat{k}\right) \end{aligned} \tag{50}$$

If $(p_0, p_3) = -(p_2, p_1)$, a similar analysis leads to infinitely many factorizations,

$$\begin{aligned} \hat{p} &= p_0 + p_1\hat{\imath} - p_0\hat{\jmath} - p_1\hat{k} \\ &= (c_0 + s_0\hat{\imath})\left(\tfrac{1}{\sqrt{2}} - \tfrac{1}{\sqrt{2}}\hat{\jmath}\right)\left(\sqrt{2}(c_0p_0 + s_0p_1) + \sqrt{2}(s_0p_0 - c_0p_1)\hat{k}\right) \end{aligned} \tag{51}$$

2.5.2 Nondegenerate Linear Equation

Equation (48) is nondegenerate when $\alpha = p_0p_1 - p_2p_3$ and $\beta = (p_0^2 - p_1^2 - p_2^2 + p_3^2)/2$ are not both zero. The conclusion is that

$$(\gamma_0, \sigma_0) = \pm\frac{(\beta, \alpha)}{\sqrt{\beta^2 + \alpha^2}} = \pm\frac{((p_0^2 - p_1^2 - p_2^2 + p_3^2)/2, p_0p_1 - p_2p_3)}{\sqrt{([p_0^2 - p_1^2 - p_2^2 + p_3^2)/2]^2 + (p_0p_1 - p_2p_3)^2}} \tag{52}$$

The sign term is one degree of freedom. We may solve for θ_0 for a choice of (γ_0, σ_0) and compute (c_0, s_0). The half-angle formulas are

$$c_0^2 = \frac{1 + \gamma_0}{2}, \quad s_0^2 = \frac{1 - \gamma_0}{2}, \quad 2s_0c_0 = \sigma_0 \tag{53}$$

For numerical robustness, if $\gamma_0 \geq 0$, then

$$c_0 = \pm\sqrt{\frac{1 + \gamma_0}{2}}, \quad s_0 = \frac{\sigma_0}{2c_0} \tag{54}$$

When $\gamma_0 < 0$, then

$$s_0 = \pm\sqrt{\frac{1 - \gamma_0}{2}}, \quad c_0 = \frac{\sigma_0}{2s_0} \tag{55}$$

In either case, you have a sign term to select, which is one degree of freedom.

Now that (c_0, s_0) is known in Equation (47), observe that

$$\begin{aligned} c_1(c_2, s_2) &= (c_0p_0 + s_0p_1, -s_0p_2 + c_0p_3) \\ s_1(c_2, s_2) &= (c_0p_2 + s_0p_3, -s_0p_0 + c_0p_1) \end{aligned} \tag{56}$$

For numerical robustness, choose the 2-tuple of larger magnitude from the right-hand side to compute (c_2, s_2). If the first 2-tuple has the larger magnitude, then

$$(c_2, s_2) = \pm\frac{(c_0p_0 + s_0p_1, -s_0p_2 + c_0p_3)}{\sqrt{(c_0p_0 + s_0p_1)^2 + (-s_0p_2 + c_0p_3)^2}} \tag{57}$$

If the second 2-tuple has the larger magnitude, then

$$(c_2, s_2) = \pm\frac{(c_0p_2 + s_0p_3, -s_0p_0 + c_0p_1)}{\sqrt{(c_0p_2 + s_0p_3)^2 + (-s_0p_0 + c_0p_1)^2}} \tag{58}$$

The magnitude comparison that leads to choosing Equation (57) is

$$\begin{aligned} 0 &\leq (c_0p_0 + s_0p_1)^2 + (-s_0p_2 + c_0p_3)^2 - (c_0p_2 + s_0p_3)^2 - (-s_0p_0 + c_0p_3)^2 \\ &= (c_0^2 - s_0^2)(p_0^2 - p_1^2 - p_2^2 + p_3^2) + 4s_0c_0(p_0p_1 - p_2p_3) \\ &= 2(\gamma_0\beta + \sigma_0\alpha) \end{aligned} \tag{59}$$

If we chose $(\gamma_0, \sigma_0) = (\beta, \alpha)/\sqrt{\beta^2 + \alpha^2}$, then Equation (57) is used for (c_2, s_2); otherwise, Equation (58) is used for (c_2, s_2). In either equation, the sign term is a degree of freedom.

Equation (47) also leads to

$$\begin{aligned} c_2(c_1, s_1) &= (c_0 p_0 + s_0 p_1, c_0 p_2 + s_0 p_3) \\ s_2(c_1, s_1) &= (-s_0 p_2 + c_0 p_3, -s_0 p_0 + c_0 p_1) \end{aligned} \tag{60}$$

For numerical robustness, if $|c_2| \geq |s_2|$, then choose

$$(c_1, s_1) = \frac{1}{c_2} (c_0 p_0 + s_0 p_1, c_0 p_2 + s_0 p_3) \tag{61}$$

otherwise, choose

$$(c_1, s_1) = \frac{1}{s_2} (-s_0 p_2 + c_0 p_3, -s_0 p_0 + c_0 p_1) \tag{62}$$

The pseudocode uses a sign of $+1$ in Equation (52), signs of $+1$ in Equations (54) and (55), and a sign of $+1$ in Equation (57).

```
void FactorQuaternionXYZ(Quaternion p, float& cx, float& sx, float& cy, float& sy, float& cz, float& sz)
{
    float a = p[0]*p[1] - p[2]*p[3];
    float b = (p[0]*p[0] - p[1]*p[1] - p[2]*p[2] + p[3]*p[3])/2;
    float length = sqrt(a*a + b*b);
    if(length > 0)
    {
        float invLength = 1/length;
        float sigma0 = a * invLength;
        float gamma0 = b * invLength;
        if(gamma0 >= 0)
        {
            cx = sqrt((1 + gamma0)/2);
            sx = sigma0/(2*cx);
        }
        else
        {
            sx = sqrt((1 - gamma0)/2);
            cx = sigma0/(2*sx);
        }

        cz = cx*p[0] + sx*p[1];
        sz = cx*p[3] - sx*p[2];
        invLength = 1/sqrt(cz*cz + sz*sz);
        cz *= invLength;
        sz *= invLength;

        if(|cz| >= |sz|)
        {
            invLength = 1/cz;
            cy = (cx*p[0] + sx*p[1]) * invLength;
            sy = (cx*p[2] + sx*p[3]) * invLength;
        }
        else
        {
            invLength = 1/sz;
            cy = (cx*p[3] - sx*p[2]) * invLength;
            sy = (cx*p[1] - sx*p[0]) * invLength;
        }
    }
    else
    {
        // Infinitely many solutions.  Choose one of them.
```

```
        if(p[0]*p[2] + p[1]*p[3] > 0)
        {
            // p = (p0,p1,p0,p1)
            cx = 1;
            sx = 0;
            cy = 1/sqrt(2);
            sy = 1/sqrt(2);
            cz = sqrt(2)*p[0];
            sz = sqrt(2)*p[1];
        }
        else
        {
            // p = (p0,p1,-p0,-p1)
            cx = 1;
            sx = 0;
            cy = 1/sqrt(2);
            sy = -1/sqrt(2);
            cz = sqrt(2)*p[0];
            sz = -sqrt(2)*p[1];
        }
    }
}
```

2.6 Rotation XYZ, XZY, YZX, YXZ, ZXY, ZYX

The code for rotation XYZ can be reused for the other triples of rotations.

2.6.1 Rotation XZY

We want to factor

$$\begin{aligned} p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k} &= (c_0 + s_0\hat{\imath})(c_2 + s_2\hat{k})(c_1 + s_1\hat{\jmath}) \\ &= (c_0c_1c_2 + s_0s_1s_2) + (s_0c_1c_2 - c_0s_1s_2)\hat{\imath} \\ &\quad + (c_0s_1c_2 - s_0c_1s_2)\hat{\jmath} + (c_0c_1s_2 + s_0s_1c_2)\hat{k} \end{aligned} \tag{63}$$

Some algebraic manipulation may be used to verify that the following factorization is equivalent

$$\begin{aligned} p_0 + p_1\hat{\imath} + p_3\hat{\jmath} - p_2\hat{k} &= (c_0 + s_0\hat{\imath})(c_2 + s_2\hat{\jmath})(c_1 - s_1\hat{k}) \\ &= (c_0c_1c_2 + s_0s_1s_2) + (s_0c_1c_2 - c_0s_1s_2)\hat{\imath} \\ &\quad + (c_0c_1s_2 + s_0s_1c_2)\hat{\jmath} - (c_0s_1c_2 - s_0c_1s_2)\hat{k} \end{aligned} \tag{64}$$

We can use the XYZ factorization code as shown next.

```
void FactorQuaternionXZY(Quaternion p, float& cx, float& sx, float& cz, float& sz, float& cy, float& sy)
{
    FactorQuaternionXYZ(Quaternion(p[0], p[1], p[3], -p[2]), cx, sx, cz, sz, cy, sy);
    sy = -sy;
}
```

2.6.2 Rotation YZX

We want to factor

$$\begin{aligned} p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k} &= (c_1 + s_1\hat{\jmath})(c_2 + s_2\hat{k})(c_0 + s_0\hat{\imath}) \\ &= (c_0c_1c_2 - s_0s_1s_2) + (c_0s_1s_2 + s_0c_1c_2)\hat{\imath} \\ &\quad + (c_0s_1c_2 + s_0c_1s_2)\hat{\jmath} + (c_0c_1s_2 - s_0s_1c_2)\hat{k} \end{aligned} \tag{65}$$

Some algebraic manipulation may be used to verify that the following factorization is equivalent

$$\begin{aligned} p_0 - p_2\hat{\imath} + p_3\hat{\jmath} - p_1\hat{k} &= (c_1 - s_1\hat{\imath})(c_2 + s_2\hat{\jmath})(c_0 - s_0\hat{k}) \\ &= (c_0c_1c_2 - s_0s_1s_2) - (c_0s_1c_2 + s_0c_1s_2)\hat{\imath} \\ &\quad + (c_0c_1s_2 - s_0s_1c_2)\hat{\jmath} - (c_0s_1s_2 + s_0c_1c_2)\hat{k} \end{aligned} \tag{66}$$

We can use the XYZ factorization code as shown next.

```
void FactorQuaternionYZX(Quaternion p, float& cy, float& sy, float& cz, float& sz, float& cx, float& sx)
{
    FactorQuaternionXYZ(Quaternion(p[0], -p[2], p[3], -p[1]), cy, sy, cz, sz, cx, sx);
    sx = -sx;
    sy = -sy;
}
```

2.6.3 Rotation YXZ

We want to factor

$$\begin{aligned} p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k} &= (c_1 + s_1\hat{\jmath})(c_0 + s_0\hat{\imath})(c_2 + s_2\hat{k}) \\ &= (c_0c_1c_2 + s_0s_1s_2) + (c_0s_1s_2 + s_0c_1c_2)\hat{\imath} \\ &\quad + (c_0s_1c_2 - s_0c_1s_2)\hat{\jmath} + (c_0c_1s_2 - s_0s_1c_2)\hat{k} \end{aligned} \tag{67}$$

Some algebraic manipulation may be used to verify that the following factorization is equivalent

$$\begin{aligned} p_0 - p_2\hat{\imath} + p_1\hat{\jmath} + p_3\hat{k} &= (c_1 - s_1\hat{\imath})(c_0 + s_0\hat{\jmath})(c_2 + s_2\hat{k}) \\ &= (c_0c_1c_2 + s_0s_1s_2) - (c_0s_1c_2 - s_0c_1s_2)\hat{\imath} \\ &\quad + (c_0s_1s_2 + s_0c_1c_2)\hat{\jmath} + (c_0c_1s_2 - s_0s_1c_2)\hat{k} \end{aligned} \tag{68}$$

We can use the XYZ factorization code as shown next.

```
void FactorQuaternionYXZ(Quaternion p, float& cy, float& sy, float& cx, float& sx, float& cz, float& sz)
{
    FactorQuaternionXYZ(Quaternion(p[0], -p[2], p[1], p[3]), cy, sy, cx, sx, cz, sz);
    sy = -sy;
}
```

2.6.4 Rotation ZXY

We want to factor

$$\begin{aligned} p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k} &= (c_2 + s_2\hat{k})(c_0 + s_0\hat{\imath})(c_1 + s_1\hat{\jmath}) \\ &= (c_0c_1c_2 - s_0s_1s_2) + (s_0c_1c_2 - c_0s_1s_2)\hat{\imath} \\ &\quad + (c_0s_1c_2 + s_0c_1s_2)\hat{\jmath} + (c_0c_1s_2 + s_0s_1c_2)\hat{k} \end{aligned} \tag{69}$$

Some algebraic manipulation may be used to verify that the following factorization is equivalent

$$\begin{aligned} p_0 - p_3\hat{\imath} + p_1\hat{\jmath} - p_2\hat{k} &= (c_1 - s_1\hat{\imath})(c_0 + s_0\hat{\jmath})(c_2 - s_2\hat{k}) \\ &= (c_0c_1c_2 - s_0s_1s_2) - (c_0c_1s_2 + s_0s_1c_2)\hat{\imath} \\ &\quad + (s_0c_1c_2 - c_0s_1s_2)\hat{\jmath} - (c_0s_1c_2 + s_0c_1s_2)\hat{k} \end{aligned} \tag{70}$$

We can use the XYZ factorization code as shown next.

```
void FactorQuaternionZXY(Quaternion p, float& cz, float& sz, float& cx, float& sx, float& cy, float& sy)
{
    FactorQuaternionXYZ(Quaternion(p[0], -p[3], p[1], -p[2]), cz, sz, cx, sx, cy, sy);
    sy = -sy;
    sz = -sz;
}
```

2.6.5 Rotation ZYX

We want to factor

$$\begin{aligned} p_0 + p_1\hat{\imath} + p_2\hat{\jmath} + p_3\hat{k} &= (c_2 + s_2\hat{k})(c_1 + s_1\hat{\jmath})(c_0 + s_0\hat{\imath}) \\ &= (c_0c_1c_2 + s_0s_1s_2) + (s_0c_1c_2 - c_0s_1s_2)\hat{\imath} \\ &\quad + (c_0s_1c_2 + s_0c_1s_2)\hat{\jmath} + (c_0c_1s_2 - s_0s_1c_2)\hat{k} \end{aligned} \tag{71}$$

Some algebraic manipulation may be used to verify that the following factorization is equivalent

$$\begin{aligned} p_0 + p_3\hat{\imath} - p_2\hat{\jmath} + p_1\hat{k} &= (c_2 + s_2\hat{\imath})(c_1 - s_1\hat{\jmath})(c_0 + s_0\hat{k}) \\ &= (c_0c_1c_2 + s_0s_1s_2) + (c_0c_1s_2 - s_0s_1c_2)\hat{\imath} \\ &\quad - (c_0s_1c_2 + s_0c_1s_2)\hat{\jmath} + (s_0c_1c_2 - c_0s_1s_2)\hat{k} \end{aligned} \tag{72}$$

We can use the XYZ factorization code as shown next.

```
void FactorQuaternionZYX(Quaternion p, float& cz, float& sz, float& cy, float& sy, float& cx, float& sx)
{
    FactorQuaternionXYZ(Quaternion(p[0], p[3], -p[2], p[1]), cz, sz, cy, sy, cx, sx);
    sy = -sy;
}
```

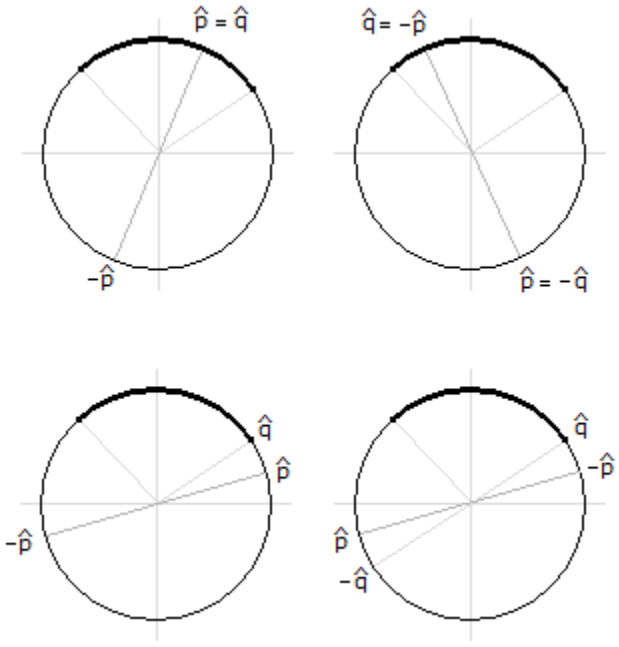
3 The Constrained Problem

3.1 Rotation X (The Analysis)

The unconstrained problem required an angle $\theta \in [-\pi, \pi]$, and may be viewed as the problem of finding the closest point on a circle to a specified point. An application might need to constrain the angle instead to $[\theta_{\min}, \theta_{\max}] \subseteq [-\pi, \pi]$. At first glance, you might convert this to the geometric problem of computing the closest point on a circular arc to a specified point. However, this is not correct in the context of quaternions as representations of rotations and orientations. The issue at hand is that $\hat{q}$ and $-\hat{q}$ represent the same rotation. We need to take this into account, effectively redefining what it means to be the closest quaternion.

Figure 3.1 illustrates the problem.

Figure 3.1 Upper left: Closest arc point **q** to a specified **p**. Upper right: Closest quaternion to a specified **p**, allowing for selection from a quaternion or its negative. The quaternion $-\mathbf{q}$ is closest, where **q** is on the arc (an interior point). Lower left: Closest arc point **q** to a specified **p**. Lower right: Closest quaternion is $-\mathbf{q}$, where **q** is an endpoint of the arc.



The algorithm must account for two things. First, it must process $-\mathbf{p}$ as well as **p**. Second, it must handle whether these quaternions are already on the **q**-arc or only near an endpoint of the arc.

Pseudocode for computing the closest quaternion is listed next. It takes into account the constrained angle.

```
Quaternion ClosestX (Quaternion p, float minAngle, float maxAngle)
```

```
{
    // Assert:  -PI <= minAngle < maxAngle <= PI

    Quaternion q;
    q[2] = 0;
    q[3] = 0;

    float sqrLength = p[0]*p[0] + p[1]*p[1];
    if (sqrLength > 0)
    {
        // Process p.
        float angle = atan2(p[1],p[0]);
        if (minAngle <= angle && angle <= maxAngle)
        {
            // p is interior to the q-arc.
            float invLength = 1/sqrt(sqrLength);
            q[0] = p[0]*invLength;
            q[1] = p[1]*invLength;
        }
        else
        {
            // Process -p.
            angle = atan2(-p[1],-p[0]);
            if (minAngle <= angle && angle <= maxAngle)
            {
                // -p is interior to the q-arc.
                float invLength = 1/sqrt(sqrLength);
                q[0] = p[0]*invLength;
                q[1] = p[1]*invLength;
            }
            else
            {
                // p and -p are exterior to the q-arc.

                // Compare p and -p to the minAngle endpoint.
                float csMin = cos(minAngle);
                float snMin = sin(minAngle);
                float dotMinAngle = p[0]*csMin + p[1]*snMin;
                if(dotMinAngle < 0)
                {
                    csMin = -csMin;
                    snMin = -snMin;
                    dotMinAngle = -dotMinAngle;
                }

                // Compare p and -p to the maxAngle endpoint.
                float csMax = cos(maxAngle);
                float snMax = sin(maxAngle);
                float dotMaxAngle = p[0]*csMax + p[1]*snMax;
                if(dotMaxAngle < 0)
                {
                    csMax = -csMax;
                    snMax = -snMax;
                    dotMaxAngle = -dotMaxAngle;
                }

                // Select q based on which endpoint led to the largest
                // dot product with p and -p.
                if(dotMaxAngle >= dotMinAngle)
                {
                    q[0] = csMax;
                    q[1] = snMax;
                }
                else
                {
                    q[0] = csMin;
                    q[1] = snMin;
                }
            }
        }
    }
    else
```

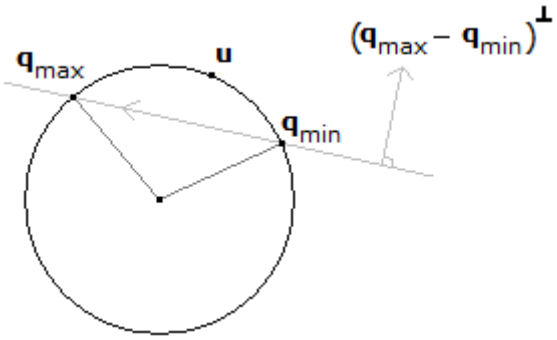
```
    {
        // Infinitely many solutions, arbitrarily choose the arc midpoint.
        float avrAngle = 0.5*(minAngle + maxAngle);
        q[0] = cos(avrAngle);
        q[1] = sin(avrAngle);
    }

    return q;
}
```

Some optimizations are possible. First, assuming `ClosestX` will be called multiple times for the same constrained angles, the sines and cosines may be precomputed for the minimum, maximum, and average angles and passed to the function.

Second, the atan2 calls can be avoided. The angle comparisons are used to determine whether $\mathbf{u} = (p_0, p_1)/|(p_0, p_1)|$ is on the circular arc with endpoints $\mathbf{q}_{\min}$ and $\mathbf{q}_{\max}$. A geometric alternative is illustrated in Figure 3.2.

Figure 3.2 An arc of a circle spanned counterclockwise from $\mathbf{q}_{\min}$ to $\mathbf{q}_{\max}$. The line containing $\mathbf{q}_{\min}$ and $\mathbf{q}_{\max}$ separates the circle into the arc itself and the remainder of the circle. $\mathbf{u}$ is on the arc since it is on the same side of the line as the arc.



Let the (non-unit-length) line direction be

$$\mathbf{D} = \mathbf{q}_{\max} - \mathbf{q}_{\min} \tag{73}$$

A line normal pointing to the side of the line containing the circular arc is

$$\mathbf{N} = (\mathbf{q}_{\max} - \mathbf{q}_{\min})^{\perp} \tag{74}$$

where the perpendicular of a vector is defined by $(d_0, d_1)^{\perp} = (d_1, -d_0)$. Because $\mathbf{D}$ and $\mathbf{N}$ are perpendicular, we may write

$$\mathbf{u} - \mathbf{q}_{\min} = \alpha\mathbf{D} + \beta\mathbf{N} \tag{75}$$

for some scalars α and β. When $\beta \geq 0$, $\mathbf{u}$ is on the circular arc, so

$$\beta = (\mathbf{u} - \mathbf{q}_{\min}) \cdot (\mathbf{q}_{\max} - \mathbf{q}_{\min})^{\perp} \geq 0 \tag{76}$$

is an algebraic test for whether the arc contains the specified point. This test requires fewer computational cycles than extracting the actual angle and comparing to the angle constraints. Equivalently, the test is

$$\beta = (\mathbf{p} - L\mathbf{q}_{\min}) \cdot (\mathbf{q}_{\max} - \mathbf{q}_{\min})^{\perp} \geq 0 \tag{77}$$

where $L = |\mathbf{p}|$ is the length of $\mathbf{p}$.

Pseudocode that uses the optimizations is listed next.

```
struct Constraints
{
    float minAngle;       // -PI <= minAngle < PI
    float maxAngle;       // minAngle < maxAngle <= PI
    float cosMinAngle;    // cos(minAngle)
    float sinMinAngle;    // sin(minAngle)
    float cosMaxAngle;    // cos(maxAngle)
    float sinMaxAngle;    // sin(maxAngle)
    float diffCosMaxMin;  // cos(maxAngle) - cos(minAngle)
    float diffSinMaxMin;  // sin(maxAngle) - sin(minAngle)
    float cosAvrAngle;    // cos((minAngle + maxAngle)/2)
    float sinAvrAngle;    // sin((minAngle + maxAngle)/2)
};

Quaternion ClosestX (Quaternion p, Constraints con)
{
    Quaternion q;
    q[2] = 0;
    q[3] = 0;

    float sqrLength = p[0]*p[0] + p[1]*p[1];
    if (sqrLength > 0)
    {
        float invLength = 1/sqrt(sqrLength);
        p[0] *= invLength;
        p[1] *= invLength;

        if ((p[0] - con.cosMinAngle)*con.diffSinMaxMin >= (p[1] - con.sinMinAngle)*con.diffCosMaxMin   // test p
        ||  (p[0] + con.cosMinAngle)*con.diffSinMaxMin <= (p[1] + con.sinMinAngle)*con.diffCosMaxMin)  // test -p
        {
            // p or -p is interior to the q-arc.
            q[0] = p[0];
            q[1] = p[1];
        }
        else
        {
            // p and -p are exterior to the q-arc.

            // Compare p and -p to the minAngle endpoint.
            float csMin = con.cosMinAngle;
            float snMin = con.sinMinAngle;
            float dotMinAngle = p[0]*csMin + p[1]*snMin;
            if(dotMinAngle < 0)
            {
                csMin = -csMin;
                snMin = -snMin;
                dotMinAngle = -dotMinAngle;
            }

            // Compare p and -p to the maxAngle endpoint.
            float csMax = con.cosMaxAngle;
            float snMax = con.sinMaxAngle;
            float dotMaxAngle = p[0]*csMax + p[1]*snMax;
            if(dotMaxAngle < 0)
            {
                csMax = -csMax;
                snMax = -snMax;
                dotMaxAngle = -dotMaxAngle;
            }
```

```
            // Select q based on which endpoint led to the largest
            // dot product with p and -p.
            if(dotMaxAngle >= dotMinAngle)
            {
                q[0] = csMax;
                q[1] = snMax;
            }
            else
            {
                q[0] = csMin;
                q[1] = snMin;
            }
        }
    }
    else
    {
        // Infinitely many solutions, arbitrarily choose the arc midpoint.
        q[0] = con.cosAvrAngle;
        q[1] = con.sinAvrAngle;
    }

    return q;
}
```

3.2 Rotation X, Y, or Z

Pseudocode is shown next for the X, Y, or Z cases. The `axis` value is 1 for X, 2 for Y, or 3 for Z.

```
Quaternion Closest (Quaternion p, int axis, Constraints con)
{
    // The appropriate nonzero components will be set later.
    Quaternion q(0,0,0,0);
    float p0 = p[0], p1 = p[axis];

    float sqrLength = p0*p0 + p1*p1;
    if (sqrLength > 0)
    {
        float invLength = 1/sqrt(sqrLength);
        p0 *= invLength;
        p1 *= invLength;

        if ((p0 - con.cosMinAngle)*con.diffSinMaxMin >= (p1 - con.sinMinAngle)*con.diffCosMaxMin   // test p
        ||  (p0 + con.cosMinAngle)*con.diffSinMaxMin <= (p1 + con.sinMinAngle)*con.diffCosMaxMin)  // test -p
        {
            // p or -p is interior to the q-arc.
            q[0] = p0;
            q[axis] = p1;
        }
        else
        {
            // p and -p are exterior to the q-arc.

            // Compare p and -p to the minAngle endpoint.
            float csMin = con.cosMinAngle;
            float snMin = con.sinMinAngle;
            float dotMinAngle = p0*csMin + p1*snMin;
            if(dotMinAngle < 0)
            {
                csMin = -csMin;
                snMin = -snMin;
                dotMinAngle = -dotMinAngle;
            }

            // Compare p and -p to the maxAngle endpoint.
            float csMax = con.cosMaxAngle;
            float snMax = con.sinMaxAngle;
            float dotMaxAngle = p0*csMax + p1*snMax;
            if(dotMaxAngle < 0)
```

```
            {
                csMax = -csMax;
                snMax = -snMax;
                dotMaxAngle = -dotMaxAngle;
            }

            // Select q based on which endpoint led to the largest
            // dot product with p and -p.
            if(dotMaxAngle >= dotMinAngle)
            {
                q[0] = csMax;
                q[axis] = snMax;
            }
            else
            {
                q[0] = csMin;
                q[axis] = snMin;
            }
        }
    }
    else
    {
        // Infinitely many solutions, choose one that satisfies the angle
        // constraints.
        q[0] = con.cosAvrAngle;
        q[axis] = con.sinAvrAngle;
    }

    return q;
}
```

3.3 Rotation XY

The pseudocode for the unconstrained case of a closest xy-quaternion is refined to allow for the Euler angle constraints. Immediately before that pseudocode, I made the observation that $(p_1, p_3) = (p_2, -p_0)$ implies $p_0p_3 - p_1p_2 = -(p_0^2 + p_1^2) < 0$ and $(p_1, p_3) = (-p_2, p_0)$ implies $p_0p_3 - p_1p_2 = p_0^2 + p_1^2 > 0$. These conditions occur in the pseudocode for the constrained case.

```
// Test whether the unit-length input point (x,y) is on the arc implied
// by the angle constraints.
bool Constraints::IsValid (float x, float y)
{
    // Test whether (x,y) satisfies the constraints.
    float xm = x - cosMinAngle;
    float ym = x - sinMinAngle;
    if (xm * diffSinMaxMin >= ym * diffCosMaxMin)
    {
        return true;
    }

    // Test whether (-x,-y) satisfies the constraints.
    float xp = x + cosMinAngle;
    float yp = y + sinMinAngle;
    if (xp * diffSinMaxMin <= yp * diffCosMaxMin)
    {
        return true;
    }

    return false;
}

void Constraints::SetAngles (Real inMinAngle, Real inMaxAngle)
{
    minAngle = inMinAngle;
    maxAngle = inMaxAngle;
    cosMinAngle = cos(minAngle);
```

```
    sinMinAngle = sin(minAngle);
    cosMaxAngle = cos(maxAngle);
    sinMaxAngle = sin(maxAngle);
    diffCosMaxMin = cosMaxAngle - cosMinAngle;
    diffSinMaxMin = sinMaxAngle - sinMinAngle;
    float avrAngle = (minAngle + maxAngle)/2;
    cosAvrAngle = cos(avrAngle);
    sinAvrAngle = sin(avrAngle);
}

Quaternion ClosestQuaternionXY (Quaternion p, Constraints xCon, Constraints yCon)
{
    Quaternion q, tmp;
    float c0, s0, c1, s1, invLength;

    float det = p[0]*p[3] - p[1]*p[2];
    if (|det| < 0.5)
    {
        float discr = sqrt(1 -det*det);
        float a = p[0]*p[1] + p[2]*p[3];
        float b = p[0]*p[0] - p[1]*p[1] + p[2]*p[2] - p[3]*p[3];

        if (b >= 0)
        {
            c0 = (discr + b)/2;
            s0 = a;
        }
        else
        {
            c0 = a;
            s0 = (discr - b)/2;
        }
        invLength = 1/sqrt(c0*c0 + s0*s0);
        c0 *= invLength;
        s0 *= invLength;

        c1 = p[0]*c0 + p[1]*s0;
        s1 = p[2]*c0 + p[3]*s0;
        invLength = 1/sqrt(c1*c1 + s1*s1);
        c1 *= invLength;
        s1 *= invLength;

        if (xCon.IsValid(c0,s0) && yCon.IsValid(c1,s1))
        {
            // The maximum occurs at an interior point.
            q[0] = c0*c1;
            q[1] = s0*c1;
            q[2] = c0*s1;
            q[3] = s0*s1;
        }
        else
        {
            // The maximum occurs at a boundary point.  Search the four
            // boundaries for the maximum.
            Quaternion r(xCon.cosMinAngle,xCon.sinMinAngle,0,0);
            Quaternion rinv(xCon.cosMinAngle,-xCon.sinMinAngle,0,0);
            Quaternion prod = rinv*p;
            tmp = GetClosest(prod,2,yCon);
            float dotOptAngle = prod.Dot(tmp);
            q = r*tmp;

            r = Quaternion(xCon.cosMaxAngle,xCon.sinMaxAngle,0,0);
            rinv = Quaternion(xCon.cosMaxAngle,-xCon.sinMaxAngle,0,0);
            prod = rinv*p;
            tmp = GetClosest(prod,2,yCon);
            float dotAngle = prod.Dot(tmp);
            if (dotAngle > dotOptAngle)
            {
                q = r*tmp;
                dotOptAngle = dotAngle;
            }
```

```
        r = Quaternion(yCon.cosMinAngle,0,yCon.sinMinAngle,0);
        rinv = Quaternion(yCon.cosMinAngle,0,-yCon.sinMinAngle,0);
        prod = p*rinv;
        tmp = GetClosest(prod,1,xCon);
        dotAngle = prod.Dot(tmp);
        if (dotAngle > dotOptAngle)
        {
            q = tmp*r;
            dotOptAngle = dotAngle;
        }

        r = Quaternion(yCon.cosMaxAngle,0,yCon.sinMaxAngle,0);
        rinv = Quaternion(yCon.cosMaxAngle,0,-yCon.SinMaxAngle,0);
        prod = p*rinv;
        tmp = GetClosest(prod,1,xCon);
        dotAngle = prod.Dot(tmp);
        if (dotAngle > dotOptAngle)
        {
            q = tmp*r;
            dotOptAngle = dotAngle;
        }
    }
}
else
{
    // Infinitely many solutions, choose one that satisfies the angle
    // constraints.
    Real minAngle, maxAngle, angle;
    Constraints con;

    if (det > 0)
    {
        minAngle = xCon.minAngle - yCon.maxAngle;
        maxAngle = xCon.maxAngle - yCon.minAngle;
        con.SetAngles(minAngle,maxAngle);

        tmp = GetClosest(p,1,kCon);

        angle = atan2(tmp[1],tmp[0]);
        if (angle < minAngle || angle > maxAngle)
        {
            angle -= (tmp[1] >= 0 ? PI : -PI);
            // assert(minAngle <= angle && angle <= maxAngle);
        }

        if (angle <= xCon.maxAngle - yCon.maxAngle)
        {
            c1 = yCon.cosMaxAngle;
            s1 = yCon.sinMaxAngle;
            angle = yCon.maxAngle + angle;
            c0 = cos(angle);
            s0 = sin(angle);
        }
        else
        {
            c0 = xCon.cosMaxAngle;
            s0 = xCon.sinMaxAngle;
            angle = xCon.maxAngle - angle;
            c1 = cos(angle);
            s1 = sin(angle);
        }
    }
    else
    {
        minAngle = xCon.minAngle + yCon.minAngle;
        maxAngle = xCon.maxAngle + yCon.maxAngle;
        con.SetAngles(minAngle,maxAngle);

        tmp = GetClosest(p,1,kCon);

        angle = atan2(tmp[1],tmp[0]);
        if (angle < minAngle || angle > maxAngle)
```

```
            {
                angle -= (tmp[1] >= 0 ? PI : -PI);
                // assert(minAngle <= angle && angle <= maxAngle);
            }

            if (angle >= xCon.minAngle + yCon.maxAngle)
            {
                c1 = yCon.cosMaxAngle;
                s1 = yCon.sinMaxAngle;
                angle = angle - yCon.maxAngle;
                c0 = cos(angle);
                s0 = sin(angle);
            }
            else
            {
                c0 = xCon.cosMaxAngle;
                s0 = xCon.sinMaxAngle;
                angle = angle - xCon.maxAngle;
                c1 = cos(angle);
                s1 = sin(angle);
            }
        }

        q[0] = c0*c1;
        q[1] = s0*c1;
        q[2] = c0*s1;
        q[3] = s0*s1;

        // Ensure that the angle between p and q is acute.
        if (p.Dot(q) < 0)
        {
            q = -q;
        }
    }

    return q;
}
```

3.4 Rotation YX

Similar to the unconstrained case, we may use the XY code to solve the YX problem.

```
Quaternion ClosestQuaternionYX (Quaternion p, Constraints yCon, Constraints xCon)
{
    Quaternion q = ClosestQuaternionXY(Quaternion(p[0], p[1], p[2], -p[3]), xCon, yCon);
    q[3] = -q[3];
    return q;
}
```

3.5 Rotation ZX, XZ, YZ, ZY

The analysis of the ZX order is similar to that of the XY order. The reversed order XZ is handled similarly to that of the reversed order YX. The same is true for handling the YZ and ZY cases. Refer to the actual Wild Magic Quaternion code for the implementations.

4 The Test Code

The following code is what was used to test the correctness of the code for computing the closest quaternion.

```
//----------------------------------------------------------------------------
// Unconstrained
//----------------------------------------------------------------------------
void TestXUnconstrained ()
{
    OutputDebugString("TestXUnconstrained\n");
    char message[256];

    const int maxSample = 1024;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestX();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 2048;
        int imax = -1;
        for (int i = 0; i <= numIterations; i++)
        {
            float angle = -Mathf::PI + i*Mathf::TWO_PI/(float)numIterations;
            angle *= 0.5f;
            float dot = fabsf(p[0]*cosf(angle) + p[1]*sinf(angle));
            if (dot > maxDot)
            {
                maxDot = dot;
                imax = i;
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYUnconstrained ()
{
    OutputDebugString("TestYUnconstrained\n");
    char message[256];

    const int maxSample = 1024;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestY();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
```

```
        const int numIterations = 2048;
        int imax = -1;
        for (int i = 0; i <= numIterations; i++)
        {
            float angle = -Mathf::PI + i*Mathf::TWO_PI/(float)numIterations;
            angle *= 0.5f;
            float dot = fabsf(p[0]*cosf(angle) + p[2]*sinf(angle));
            if (dot > maxDot)
            {
                maxDot = dot;
                imax = i;
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZUnconstrained ()
{
    OutputDebugString("TestZUnconstrained\n");
    char message[256];

    const int maxSample = 1024;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestZ();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 2048;
        int imax = -1;
        for (int i = 0; i <= numIterations; i++)
        {
            float angle = -Mathf::PI + i*Mathf::TWO_PI/(float)numIterations;
            angle *= 0.5f;
            float dot = fabsf(p[0]*cosf(angle) + p[3]*sinf(angle));
            if (dot > maxDot)
            {
                maxDot = dot;
                imax = i;
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }
```

```
    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXYUnconstrained ()
{
    OutputDebugString("TestXYUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestXY();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float xAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            xAngle *= 0.5f;
            Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = rx*ry;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXYUnconstrainedSpecial ()
{
    OutputDebugString("TestXYUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
```

```
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[1];
        p[3] = -sign*p[0];
        p.Normalize();

        Quaternionf q = p.GetClosestXY();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float xAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            xAngle *= 0.5f;
            Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = rx*ry;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYXUnconstrained ()
{
    OutputDebugString("TestYXUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestYX();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float xAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
```

```
            xAngle *= 0.5f;
            Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = ry*rx;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYXUnconstrainedSpecial ()
{
    OutputDebugString("TestYXUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[1];
        p[3] = sign*p[0];
        p.Normalize();

        Quaternionf q = p.GetClosestYX();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float xAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            xAngle *= 0.5f;
            Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = ry*rx;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
```

```
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZXUnconstrained ()
{
    OutputDebugString("TestZXUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestZX();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float xAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                xAngle *= 0.5f;
                Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
                Quaternionf tmp = rz*rx;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
```

```
}
//----------------------------------------------------------------------------
void TestZXUnconstrainedSpecial ()
{
    OutputDebugString("TestZXUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[0];
        p[3] = -sign*p[1];
        p.Normalize();

        Quaternionf q = p.GetClosestZX();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float xAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                xAngle *= 0.5f;
                Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
                Quaternionf tmp = rz*rx;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXZUnconstrained ()
{
    OutputDebugString("TestXZUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
```

```
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestXZ();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float xAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                xAngle *= 0.5f;
                Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
                Quaternionf tmp = rx*rz;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXZUnconstrainedSpecial ()
{
    OutputDebugString("TestXZUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[0];
        p[3] = sign*p[1];
        p.Normalize();

        Quaternionf q = p.GetClosestXZ();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
```

```
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float xAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                xAngle *= 0.5f;
                Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
                Quaternionf tmp = rx*rz;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYZUnconstrained ()
{
    OutputDebugString("TestYZUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestYZ();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = ry*rz;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
```

```
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYZUnconstrainedSpecial ()
{
    OutputDebugString("TestYZUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[1] = sign*p[0];
        p[3] = -sign*p[2];
        p.Normalize();

        Quaternionf q = p.GetClosestYZ();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = ry*rz;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
```

```
}
//----------------------------------------------------------------------------
void TestZYUnconstrained ()
{
    OutputDebugString("TestZYUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        Quaternionf q = p.GetClosestZY();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = rz*ry;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZYUnconstrainedSpecial ()
{
    OutputDebugString("TestZYUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
```

```
        p[2] = Mathf::SymmetricRandom();
        p[1] = sign*p[0];
        p[3] = sign*p[2];
        p.Normalize();

        Quaternionf q = p.GetClosestZY();

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = -Mathf::PI + i1*Mathf::TWO_PI/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = -Mathf::PI + i0*Mathf::TWO_PI/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = rz*ry;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXYZUnconstrained()
{
    OutputDebugString("TestXYZUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorXYZ(cx, sx, cy, sy, cz, sz);
        Quaternionf q = Quaternionf(cx,sx,0.0f,0.0f)*
            Quaternionf(cy,0.0f,sy,0.0f)*Quaternionf(cz,0.0f,0.0f,sz);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
```

```
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXYZUnconstrainedSpecial()
{
    OutputDebugString("TestXYZUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[0];
        p[3] = sign*p[1];
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorXYZ(cx, sx, cy, sy, cz, sz);
        Quaternionf q = Quaternionf(cx,sx,0.0f,0.0f)*
            Quaternionf(cy,0.0f,sy,0.0f)*Quaternionf(cz,0.0f,0.0f,sz);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXZYUnconstrained()
{
    OutputDebugString("TestXZYUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorXZY(cx, sx, cz, sz, cy, sy);
        Quaternionf q =
            Quaternionf(cx,sx,0.0f,0.0f) *
            Quaternionf(cz,0.0f,0.0f,sz) *
            Quaternionf(cy,0.0f,sy,0.0f);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
```

```
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXZYUnconstrainedSpecial()
{
    OutputDebugString("TestXZYUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[1];
        p[3] = -sign*p[0];
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorXZY(cx, sx, cz, sz, cy, sy);
        Quaternionf q =
            Quaternionf(cx,sx,0.0f,0.0f) *
            Quaternionf(cz,0.0f,0.0f,sz) *
            Quaternionf(cy,0.0f,sy,0.0f);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYXZUnconstrained()
{
    OutputDebugString("TestYXZUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorYXZ(cy, sy, cx, sx, cz, sz);
        Quaternionf q =
            Quaternionf(cy,0.0f,sy,0.0f) *
```

```
            Quaternionf(cx,sx,0.0f,0.0f) *
            Quaternionf(cz,0.0f,0.0f,sz);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYXZUnconstrainedSpecial()
{
    OutputDebugString("TestYXZUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[1] = sign*p[0];
        p[3] = -sign*p[2];
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorYXZ(cy, sy, cx, sx, cz, sz);
        Quaternionf q =
            Quaternionf(cy,0.0f,sy,0.0f) *
            Quaternionf(cx,sx,0.0f,0.0f) *
            Quaternionf(cz,0.0f,0.0f,sz);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYZXUnconstrained()
{
    OutputDebugString("TestYZXUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();
```

```
        float cx, sx, cy, sy, cz, sz;
        p.FactorYZX(cy, sy, cz, sz, cx, sx);
        Quaternionf q =
            Quaternionf(cy,0.0f,sy,0.0f) *
            Quaternionf(cz,0.0f,0.0f,sz) *
            Quaternionf(cx,sx,0.0f,0.0f);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYZXUnconstrainedSpecial()
{
    OutputDebugString("TestYZXUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[1];
        p[3] = sign*p[0];
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorYZX(cy, sy, cz, sz, cx, sx);
        Quaternionf q =
            Quaternionf(cy,0.0f,sy,0.0f) *
            Quaternionf(cz,0.0f,0.0f,sz) *
            Quaternionf(cx,sx,0.0f,0.0f);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZXYUnconstrained()
{
    OutputDebugString("TestZXYUnconstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
```

```
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorZXY(cz, sz, cx, sx, cy, sy);
        Quaternionf q =
            Quaternionf(cz,0.0f,0.0f,sz) *
            Quaternionf(cx,sx,0.0f,0.0f) *
            Quaternionf(cy,0.0f,sy,0.0f);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZXYUnconstrainedSpecial()
{
    OutputDebugString("TestZXYUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[1] = sign*p[0];
        p[3] = sign*p[2];
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorZXY(cz, sz, cx, sx, cy, sy);
        Quaternionf q =
            Quaternionf(cz,0.0f,0.0f,sz) *
            Quaternionf(cx,sx,0.0f,0.0f) *
            Quaternionf(cy,0.0f,sy,0.0f);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZYXUnconstrained()
{
    OutputDebugString("TestZYXUnconstrained\n");
    char message[256];

    const int maxSample = 128;
```

```
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorZYX(cz, sz, cy, sy, cx, sx);
        Quaternionf q =
            Quaternionf(cz,0.0f,0.0f,sz) *
            Quaternionf(cy,0.0f,sy,0.0f) *
            Quaternionf(cx,sx,0.0f,0.0f);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZYXUnconstrainedSpecial()
{
    OutputDebugString("TestZYXUnconstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = -sign*p[0];
        p[3] = sign*p[1];
        p.Normalize();

        float cx, sx, cy, sy, cz, sz;
        p.FactorZYX(cz, sz, cy, sy, cx, sx);
        Quaternionf q =
            Quaternionf(cz,0.0f,0.0f,sz) *
            Quaternionf(cy,0.0f,sy,0.0f) *
            Quaternionf(cx,sx,0.0f,0.0f);

        float dotPQ = p.Dot(q);
        float error = fabsf(1.0f - dotPQ);
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
```

```
//----------------------------------------------------------------------------
// Constrained
//----------------------------------------------------------------------------
void TestXConstrained ()
{
    OutputDebugString("TestXConstrained\n");
    char message[256];

    const int maxSample = 1024;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float minAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxAngle = Mathf::IntervalRandom(minAngle, Mathf::PI);
        Quaternionf::Constraints con(0.5f*minAngle, 0.5f*maxAngle);

        Quaternionf q = p.GetClosestX(con);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 2048;
        int imax = -1;
        for (int i = 0; i <= numIterations; i++)
        {
            float angle = minAngle + i*(maxAngle - minAngle)/(float)numIterations;
            angle *= 0.5f;
            float dot = fabsf(p[0]*cosf(angle) + p[1]*sinf(angle));
            if (dot > maxDot)
            {
                maxDot = dot;
                imax = i;
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYConstrained ()
{
    OutputDebugString("TestYConstrained\n");
    char message[256];

    const int maxSample = 1024;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();
```

```
        float minAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxAngle = Mathf::IntervalRandom(minAngle, Mathf::PI);
        Quaternionf::Constraints con(0.5f*minAngle, 0.5f*maxAngle);

        Quaternionf q = p.GetClosestY(con);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 2048;
        int imax = -1;
        for (int i = 0; i <= numIterations; i++)
        {
            float angle = minAngle + i*(maxAngle - minAngle)/(float)numIterations;
            angle *= 0.5f;
            float dot = fabsf(p[0]*cosf(angle) + p[2]*sinf(angle));
            if (dot > maxDot)
            {
                maxDot = dot;
                imax = i;
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZConstrained ()
{
    OutputDebugString("TestZConstrained\n");
    char message[256];

    const int maxSample = 1024;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float minAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxAngle = Mathf::IntervalRandom(minAngle, Mathf::PI);
        Quaternionf::Constraints con(0.5f*minAngle, 0.5f*maxAngle);

        Quaternionf q = p.GetClosestZ(con);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 2048;
        int imax = -1;
        for (int i = 0; i <= numIterations; i++)
        {
            float angle = minAngle + i*(maxAngle - minAngle)/(float)numIterations;
            angle *= 0.5f;
            float dot = fabsf(p[0]*cosf(angle) + p[3]*sinf(angle));
            if (dot > maxDot)
            {
                maxDot = dot;
                imax = i;
            }
```

```
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXYConstrained ()
{
    OutputDebugString("TestXYConstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float minXAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxXAngle = Mathf::IntervalRandom(minXAngle, Mathf::PI);
        Quaternionf::Constraints xCon(0.5f*minXAngle, 0.5f*maxXAngle);

        float minYAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxYAngle = Mathf::IntervalRandom(minYAngle, Mathf::PI);
        Quaternionf::Constraints yCon(0.5f*minYAngle, 0.5f*maxYAngle);

        Quaternionf q = p.GetClosestXY(xCon, yCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float xAngle = minXAngle + i1*(maxXAngle - minXAngle)/(float)numIterations;
            xAngle *= 0.5f;
            Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = minYAngle + i0*(maxYAngle - minYAngle)/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = rx*ry;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
```

```
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXYConstrainedSpecial ()
{
    OutputDebugString("TestXYConstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[1];
        p[3] = -sign*p[0];
        p.Normalize();

        float minXAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxXAngle = Mathf::IntervalRandom(minXAngle, Mathf::PI);
        Quaternionf::Constraints xCon(0.5f*minXAngle, 0.5f*maxXAngle);

        float minYAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxYAngle = Mathf::IntervalRandom(minYAngle, Mathf::PI);
        Quaternionf::Constraints yCon(0.5f*minYAngle, 0.5f*maxYAngle);

        Quaternionf q = p.GetClosestXY(xCon, yCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float xAngle = minXAngle + i1*(maxXAngle - minXAngle)/(float)numIterations;
            xAngle *= 0.5f;
            Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = minYAngle + i0*(maxYAngle - minYAngle)/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = rx*ry;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
```

```
}
//----------------------------------------------------------------------------
void TestYXConstrained ()
{
    OutputDebugString("TestYXConstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float minXAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxXAngle = Mathf::IntervalRandom(minXAngle, Mathf::PI);
        Quaternionf::Constraints xCon(0.5f*minXAngle, 0.5f*maxXAngle);

        float minYAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxYAngle = Mathf::IntervalRandom(minYAngle, Mathf::PI);
        Quaternionf::Constraints yCon(0.5f*minYAngle, 0.5f*maxYAngle);

        Quaternionf q = p.GetClosestYX(yCon, xCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float xAngle = minXAngle + i1*(maxXAngle - minXAngle)/(float)numIterations;
            xAngle *= 0.5f;
            Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = minYAngle + i0*(maxYAngle - minYAngle)/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = ry*rx;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYXConstrainedSpecial ()
{
    OutputDebugString("TestYXConstrainedSpecial\n");
    char message[256];
```

```
    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[1];
        p[3] = sign*p[0];
        p.Normalize();

        float minXAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxXAngle = Mathf::IntervalRandom(minXAngle, Mathf::PI);
        Quaternionf::Constraints xCon(0.5f*minXAngle, 0.5f*maxXAngle);

        float minYAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxYAngle = Mathf::IntervalRandom(minYAngle, Mathf::PI);
        Quaternionf::Constraints yCon(0.5f*minYAngle, 0.5f*maxYAngle);

        Quaternionf q = p.GetClosestYX(yCon, xCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float xAngle = minXAngle + i1*(maxXAngle - minXAngle)/(float)numIterations;
            xAngle *= 0.5f;
            Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = minYAngle + i0*(maxYAngle - minYAngle)/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = ry*rx;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZXConstrained ()
{
    OutputDebugString("TestZXConstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
```

```
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float minXAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxXAngle = Mathf::IntervalRandom(minXAngle, Mathf::PI);
        Quaternionf::Constraints xCon(0.5f*minXAngle, 0.5f*maxXAngle);

        float minZAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxZAngle = Mathf::IntervalRandom(minZAngle, Mathf::PI);
        Quaternionf::Constraints zCon(0.5f*minZAngle, 0.5f*maxZAngle);

        Quaternionf q = p.GetClosestZX(zCon, xCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = minZAngle + i1*(maxZAngle - minZAngle)/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float xAngle = minXAngle + i0*(maxXAngle - minXAngle)/(float)numIterations;
                xAngle *= 0.5f;
                Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
                Quaternionf tmp = rz*rx;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZXConstrainedSpecial ()
{
    OutputDebugString("TestZXConstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[0];
        p[3] = -sign*p[1];
        p.Normalize();
```

```
        float minZAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxZAngle = Mathf::IntervalRandom(minZAngle, Mathf::PI);
        Quaternionf::Constraints zCon(0.5f*minZAngle, 0.5f*maxZAngle);

        float minXAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxXAngle = Mathf::IntervalRandom(minXAngle, Mathf::PI);
        Quaternionf::Constraints xCon(0.5f*minXAngle, 0.5f*maxXAngle);

        Quaternionf q = p.GetClosestZX(zCon, xCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = minZAngle + i1*(maxZAngle - minZAngle)/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float xAngle = minXAngle + i0*(maxXAngle - minXAngle)/(float)numIterations;
                xAngle *= 0.5f;
                Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
                Quaternionf tmp = rz*rx;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXZConstrained ()
{
    OutputDebugString("TestXZConstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float minXAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxXAngle = Mathf::IntervalRandom(minXAngle, Mathf::PI);
        Quaternionf::Constraints xCon(0.5f*minXAngle, 0.5f*maxXAngle);

        float minZAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxZAngle = Mathf::IntervalRandom(minZAngle, Mathf::PI);
        Quaternionf::Constraints zCon(0.5f*minZAngle, 0.5f*maxZAngle);
```

```
        Quaternionf q = p.GetClosestXZ(xCon, zCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = minZAngle + i1*(maxZAngle - minZAngle)/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float xAngle = minXAngle + i0*(maxXAngle - minXAngle)/(float)numIterations;
                xAngle *= 0.5f;
                Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
                Quaternionf tmp = rx*rz;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestXZConstrainedSpecial ()
{
    OutputDebugString("TestXZConstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = sign*p[0];
        p[3] = sign*p[1];
        p.Normalize();

        float minZAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxZAngle = Mathf::IntervalRandom(minZAngle, Mathf::PI);
        Quaternionf::Constraints zCon(0.5f*minZAngle, 0.5f*maxZAngle);

        float minXAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxXAngle = Mathf::IntervalRandom(minXAngle, Mathf::PI);
        Quaternionf::Constraints xCon(0.5f*minXAngle, 0.5f*maxXAngle);

        Quaternionf q = p.GetClosestXZ(xCon, zCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
```

```
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = minZAngle + i1*(maxZAngle - minZAngle)/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float xAngle = minXAngle + i0*(maxXAngle - minXAngle)/(float)numIterations;
                xAngle *= 0.5f;
                Quaternionf rx(cosf(xAngle), sinf(xAngle), 0.0f, 0.0f);
                Quaternionf tmp = rx*rz;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZYConstrained ()
{
    OutputDebugString("TestZYConstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float minZAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxZAngle = Mathf::IntervalRandom(minZAngle, Mathf::PI);
        Quaternionf::Constraints zCon(0.5f*minZAngle, 0.5f*maxZAngle);

        float minYAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxYAngle = Mathf::IntervalRandom(minYAngle, Mathf::PI);
        Quaternionf::Constraints yCon(0.5f*minYAngle, 0.5f*maxYAngle);

        Quaternionf q = p.GetClosestZY(zCon, yCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = minZAngle + i1*(maxZAngle - minZAngle)/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
```

```
            {
                float yAngle = minYAngle + i0*(maxYAngle - minYAngle)/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = rz*ry;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestZYConstrainedSpecial ()
{
    OutputDebugString("TestZYConstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[1] = sign*p[0];
        p[3] = sign*p[2];
        p.Normalize();

        float minZAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxZAngle = Mathf::IntervalRandom(minZAngle, Mathf::PI);
        Quaternionf::Constraints zCon(0.5f*minZAngle, 0.5f*maxZAngle);

        float minYAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxYAngle = Mathf::IntervalRandom(minYAngle, Mathf::PI);
        Quaternionf::Constraints yCon(0.5f*minYAngle, 0.5f*maxYAngle);

        Quaternionf q = p.GetClosestZY(zCon, yCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = minZAngle + i1*(maxZAngle - minZAngle)/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = minYAngle + i0*(maxYAngle - minYAngle)/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = rz*ry;
                float dot = fabsf(p.Dot(tmp));
```

```
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYZConstrained ()
{
    OutputDebugString("TestYZConstrained\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[1] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[3] = Mathf::SymmetricRandom();
        p.Normalize();

        float minZAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxZAngle = Mathf::IntervalRandom(minZAngle, Mathf::PI);
        Quaternionf::Constraints zCon(0.5f*minZAngle, 0.5f*maxZAngle);

        float minYAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxYAngle = Mathf::IntervalRandom(minYAngle, Mathf::PI);
        Quaternionf::Constraints yCon(0.5f*minYAngle, 0.5f*maxYAngle);

        Quaternionf q = p.GetClosestYZ(yCon, zCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = minZAngle + i1*(maxZAngle - minZAngle)/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = minYAngle + i0*(maxYAngle - minYAngle)/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = ry*rz;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
```

```
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//----------------------------------------------------------------------------
void TestYZConstrainedSpecial ()
{
    OutputDebugString("TestYZConstrainedSpecial\n");
    char message[256];

    const int maxSample = 128;
    float maxError = -Mathf::MAX_REAL;
    int maxIndex = -1;
    for (int s = 0; s <= maxSample; s++)
    {
        float sign = (Mathf::SymmetricRandom() >= 0.0f ? 1.0f : -1.0f);
        Quaternionf p;
        p[0] = Mathf::SymmetricRandom();
        p[2] = Mathf::SymmetricRandom();
        p[1] = sign*p[0];
        p[3] = -sign*p[2];
        p.Normalize();

        float minZAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxZAngle = Mathf::IntervalRandom(minZAngle, Mathf::PI);
        Quaternionf::Constraints zCon(0.5f*minZAngle, 0.5f*maxZAngle);

        float minYAngle = Mathf::IntervalRandom(-Mathf::PI, Mathf::PI);
        float maxYAngle = Mathf::IntervalRandom(minYAngle, Mathf::PI);
        Quaternionf::Constraints yCon(0.5f*minYAngle, 0.5f*maxYAngle);

        Quaternionf q = p.GetClosestYZ(yCon, zCon);

        float dotPQ = p.Dot(q);
        float maxDot = -1.0f;
        const int numIterations = 512;
        int i0Max = -1, i1Max = -1;
        for (int i1 = 0; i1 <= numIterations; i1++)
        {
            float zAngle = minZAngle + i1*(maxZAngle - minZAngle)/(float)numIterations;
            zAngle *= 0.5f;
            Quaternionf rz(cosf(zAngle), 0.0f, 0.0f, sinf(zAngle));
            for (int i0 = 0; i0 <= numIterations; i0++)
            {
                float yAngle = minYAngle + i0*(maxYAngle - minYAngle)/(float)numIterations;
                yAngle *= 0.5f;
                Quaternionf ry(cosf(yAngle), 0.0f, sinf(yAngle), 0.0f);
                Quaternionf tmp = ry*rz;
                float dot = fabsf(p.Dot(tmp));
                if (dot > maxDot)
                {
                    maxDot = dot;
                    i0Max = i0;
                    i1Max = i1;
                }
            }
        }

        float error = maxDot - dotPQ;
        if(error > maxError)
        {
            maxError = error;
```

```
            maxIndex = s;
            sprintf_s(message,256,"s = %d, error = %g\n",maxIndex,maxError);
            OutputDebugString(message);
        }
    }

    OutputDebugString("\n");
}
//---------------------------------------------------------------------------
int main (int, char**)
{
    TestXUnconstrained();
    TestYUnconstrained();
    TestZUnconstrained();
    TestXYUnconstrained();
    TestXYUnconstrainedSpecial();
    TestYXUnconstrained();
    TestYXUnconstrainedSpecial();
    TestZXUnconstrained();
    TestZXUnconstrainedSpecial();
    TestXZUnconstrained();
    TestXZUnconstrainedSpecial();
    TestYZUnconstrained();
    TestYZUnconstrainedSpecial();
    TestZYUnconstrained();
    TestZYUnconstrainedSpecial();
    TestXYZUnconstrained();
    TestXYZUnconstrainedSpecial();
    TestXZYUnconstrained();
    TestXZYUnconstrainedSpecial();
    TestYXZUnconstrained();
    TestYXZUnconstrainedSpecial();
    TestYZXUnconstrained();
    TestYZXUnconstrainedSpecial();
    TestZXYUnconstrained();
    TestZXYUnconstrainedSpecial();
    TestZYXUnconstrained();
    TestZYXUnconstrainedSpecial();

    TestXConstrained();
    TestYConstrained();
    TestZConstrained();
    TestXYConstrained();
    TestXYConstrainedSpecial();
    TestYXConstrained();
    TestYXConstrainedSpecial();
    TestZXConstrained();
    TestZXConstrainedSpecial();
    TestXZConstrained();
    TestXZConstrainedSpecial();
    TestZYConstrained();
    TestZYConstrainedSpecial();
    TestYZConstrained();
    TestYZConstrainedSpecial();
    return 0;
}
//---------------------------------------------------------------------------
```