Interpolation of Rigid Motions in 3D

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: October 15, 2017

Contents

1 Introduction

This document describes how to intepolate between two rigid transformations, each involving rotation and translation; reflections are not considered here. Moreover, the interpolating function is a curve that is required to be a shortest-distance path between the transformations, in which case the path is geodesic. Naturally, the concept of geodesic path requires an underlying mathematical framework. In the specific case at hand, the constructions involve *Lie groups*, *Lie algebras* and exponential maps. In the context of interpolation, the mathematical construction involves square matrices A and computing formally the power series $e^A = \sum_{n=0}^{\infty} A^n/n!$. A good summary of computing various quantities of interest for some standard Lie groups and Lie algebras is found in Ethan Eade's online paper [1]. His site also has related papers of interest.

2 Interpolation of Translations

The simplest form of a rigid motion involves only translation. Given two translations $\mathbf{T}_0$ and $\mathbf{T}_1$, each a 3×1 point in space, the shortest path connecting them is parameterized as

$$\mathbf{F}(t; \mathbf{T}_0, \mathbf{T}_1) = (1-t)\mathbf{T}_0 + t\mathbf{T}_1, \quad t \in [0,1] \tag{1}$$

The underlying geometry is Euclidean 3-space, and it is well known that in Euclidean n-space the shortest path between two points is a line segment.

A line segment can be parameterized in infinitely many ways. The most interesting one is *parameterization by arc length*. If $L = |\mathbf{T}_1 - \mathbf{T}_0|$, define $s = Lt$ and

$$\mathbf{G}(s; \mathbf{T}_0, \mathbf{T}_1) = \left(1 - \frac{s}{L}\right)\mathbf{T}_0 + \frac{s}{L}\,\mathbf{T}_1, \quad s \in [0, L] \tag{2}$$

For a parameter value s in its domain, the point $\mathbf{G}(s)$ is that point s units of distance along the curve from the endpoint $\mathbf{G}(0)$. If you think of a particle traveling along the curve, the derivative of the curve corresponds to the velocity of the particle and the length of the derivative is the speed of the particle. For parameterization by arc length, the derivative is

$$\frac{d\mathbf{G}(s)}{ds} = \frac{|\mathbf{T}_1 - \mathbf{T}_0|}{L} = 1 \tag{3}$$

Therefore, the particle travels along the curve with unit speed. The derivative of the parameterization of equation (1) is

$$\frac{d\mathbf{F}(t)}{dt} = |\mathbf{T}_1 - \mathbf{T}_0| = L \tag{4}$$

In this parameterization, the particle travels along the curve with constant speed (that is not necessarily unit speed). For the purpose of this document, it is sufficient to produce interpolations where the derivatives are independent of t; the particles travel along a curve with constant speed. For a more detailed discussion of constant-speed particles, see [2].

3 Interpolation of Rotations

Computing a shortest-distance path between two rotations is a topic of interest in geometry-related fields such as computer graphics, computer vision and robotics. The formulation made popular in computer

graphics by Ken Shoemake is *spherical linear interpolation (SLERP)* that involves representing the rotations as unit-length quaternions; [4].

3.1 Interpolation using a Quaternion Representation

Let R_0 and R_1 be 3×3 rotation matrices with the convention that they are applied to a 3×1 vector $\mathbf{X}$ as $R_i\mathbf{X}$. Let q_0 and q_1 be unit-length quaternions that represent the rotations. See [3] for details about the relationship between rotation matrices and quaternions. The SLERP of two quaternions is

$$q(t) = \frac{\sin((1-t)\phi)}{\sin(\phi)}\, q_0 + \frac{\sin(t\phi)}{\sin(\phi)}\, q_1, \quad t \in [0,1] \tag{5}$$

where ϕ is the angle between q_0 and q_1 treated as 4-tuple vectors, in which case $q_0 \cdot q_1 = \cos(\phi)$. The quaternions q and $-q$ represent the same rotation, so usually q_0 and q_1 are chosen so that ϕ is in $[0, \pi/2]$.

The unit-length quaternions live on the 3-dimensional unit sphere centered at the origin in 4-space. Equation (5) is a parameterization of the geodesic path on the hypersphere with q_0 and q_1 the endpoints. If you were to visualize this in 3D, say w-components are set to 0, the geodesic path is the great-circle arc connecting the two points. Notice that when $\phi = \pi$, the two points are antipodal, in which case there are infinitely many great-circle arcs connecting the two points. SLERP is not used in these cases.

The derivative of the slerp equation with respect to parameter t is

$$q'(t) = \frac{-\phi\cos((1-t)\phi)}{\sin(\phi)}\, q_0 + \frac{\phi\cos(t\phi)}{\sin(\phi)}\, q_1 \tag{6}$$

Treating the quaternions as 4-tuple vectors, the squared length is

$$q'(t) \cdot q'(t) = [\phi^2/\sin^2(\phi)]\left[\cos^2((1-t)\phi) + \cos^2(t\phi) - 2\cos((1-t)\phi)\cos(t\phi)\cos(\phi)\right] = \phi^2 \tag{7}$$

where the first equality uses $q_0 \cdot q_1 = \cos(\phi)$ and where the second equality requires expanding $\cos((1-t)\phi) = \cos(\phi)\cos(t\phi) + \sin(\phi)\sin(t\phi)$ and using some trigonometric identities and algebraic factoring. A particle travels along the curve with constant speed $|q'(t)| = \phi$.

3.2 Interpolation using a Matrix Representation

Computing the geodesic path connecting two rotations does not require a quaternion representation. It is possible to compute it using a matrix representation and the exponential map for rotations as the Lie Group $SO(3)$.

Let S be a skew-symmetric matrix,

$$S = \begin{bmatrix} 0 & -s_2 & s_1 \\ s_2 & 0 & -s_0 \\ -s_1 & s_0 & 0 \end{bmatrix} \tag{8}$$

and let $\mathbf{s} = (s_0, s_1, s_2)$. If $\mathbf{s}$ is not the zero vector, then the angle of rotation is $\theta = |\mathbf{s}|$ and the axis of rotation has a unit-length direction $\mathbf{s}/\theta$. The corresponding rotation matrix R can be computed using the

Taylor series for e^x, where x is replaced by the matrix S,

$$R = \exp(S) = \sum_{n=0}^{\infty} \frac{S^n}{n!} = I + \left(\frac{\sin\theta}{\theta}\right) S + \left(\frac{1-\cos\theta}{\theta^2}\right) S^2 \tag{9}$$

The infinite sum reduces to a finite sum because of the identity $S^3 = -\theta^2 S$. For practical purposes, constrain $\theta \in [0, \pi]$.

We may think of applying a logarithm to extract S given R, say, $S = \log(R)$. Equation (9) can be used to solve for S. When $0 < \theta < \pi$, observe that $S^{\mathsf{T}} = -S$ and

$$R - R^{\mathsf{T}} = 2\left(\frac{\sin\theta}{\theta}\right) S \tag{10}$$

in which case

$$S = \frac{\theta}{2\sin\theta}\left(R - R^{\mathsf{T}}\right) = \log(R) \tag{11}$$

When $\theta = 0$, special handling is required to solve for S. Note that $\lim_{\theta\to 0}(\sin\theta)/\theta = 1$ and equation (11) still applies.

Numerically, for θ nearly 0, an approximation to $(\sin\theta)/\theta$ should be used. The reference [1] suggests using a Taylor polynomial approximation. Generally, it is better to use minimax approximations to obtain good accuracy over the entire range of inputs for the same (or smaller) computational cost. In the case at hand, these approximations are of the form $(\sin\theta)/\theta = \sum_{k=0}^{n} p_k \theta^{2k}$.

Special handling must also be used when $\theta = \pi$. In this case, equation (11) does not apply because of the implied division by zero. When $\theta = \pi$, $R = R^{\mathsf{T}}$; that is, the rotation matrix is symmetric and $R = I + (2/\pi^2)S^2$. Observe that generally, $S^2 = \mathbf{s}\mathbf{s}^{\mathsf{T}} - \theta^2 I$. In our special case, this leads to $\mathbf{s}\mathbf{s}^{\mathsf{T}} = (\pi^2/2)(R+I)$. For numerical robustness, locate the row of $(\pi^2/2)(R+I)$ that has the largest length and normalize it to obtain $\mathbf{s}$, which in turn determines S. One may just as well use $-\mathbf{s}$, but this leads to the same rotation matrix.

Given two rotation matrices R_0 and R_1, from the theory of Lie groups, the geodesic path connecting them is

$$F(t; R_0, R_1) = \exp(t\log(R_1 R_0^{-1}))R_0, \quad t \in [0,1] \tag{12}$$

It is easy to verify that $F(0) = R_0$ and $F(1) = R_1$ using $\exp(0) = I$, where in the exponent 0 is the zero matrix, and $\exp(\log(R_1 R_0^{-1})) = R_1 R_0^{-1}$. Because R_0 is a rotation matrix, we can replace R_0^{-1} by R_0^{T}.

The steps in evaluating $F(t)$ are as follows. Compute the matrix product $R = R_1 R_0^{\mathsf{T}}$. Compute $S = \log(R)$ using equation (11) and note that

$$R = R_1 R_0^{\mathsf{T}} = I + \left(\frac{\sin(\theta)}{\theta}\right) S + \left(\frac{1-\cos(\theta)}{\theta^2}\right) S^2 \tag{13}$$

Compute equation (9) with S replaced by tS and θ replaced by $t\theta$ and then multiply the result by R_0,

$$F(t) = \left[I + \left(\frac{\sin(t\theta)}{(t\theta)}\right)(tS) + \left(\frac{1-\cos(t\theta)}{(t\theta)^2}\right)(tS)^2\right] R_0 = \left[I + \left(\frac{\sin(t\theta)}{\theta}\right) S + \left(\frac{1-\cos(t\theta)}{\theta^2}\right) S^2\right] R_0 \tag{14}$$

It is easily verified that $F(0) = R_0$ and $F(1) = R_1$ and it is evident that $F(t)$ is a rotation matrix for all t.

The interpolation for translations is a curve of 3-tuples and the interpolation for quaternions is a curve of 4-tuples. In both cases, we verified the constant-speed condition by computing t-derivatives and observing that the lengths of the derivatives are constant. In the case of rotation matrices, we have a curve of 3×3 matrices. A natural measure for size of the derivative is a *matrix norm*. I choose to use the Frobenius norm, which is the generalization of the Euclidean length of a vector to matrices. The Frobenius norm is the square root of the sum of squares of the matrix entries. Equivalently, the Frobenius norm is the square root of the sum of the squared lengths of the matrix columns.

Given a square matrix M, the Frobenius norm is denoted $\|M\|$. It is easily proved that if Q is an orthogonal matrix, then $\|QM\| = \|M\| = \|MQ\|$.

The derivative for the geodesic path of rotations is

$$F'(t) = \exp(t \log(R_1 R_0^{-1})) \log(R_1 R_0^{-1}) R_0 = R(t) \log(R_1 R_0^{-1}) R_0 \tag{15}$$

where $R(t)$ is a rotation matrix and is defined by the last equality. The Frobenius norm is

$$\|F'(t)\| = \|R(t) \log(R_1 R_0^{-1}) R_0\| = \|\log(R_1 R_0^{-1}) R_0\| = \|\log(R_1 R_0^{-1})\| \tag{16}$$

The right-hand side is constant with respect to t, so a particle traverses the curve of rotations with constant speed.

4 Interpolation of Rotations and Translations

Consider rigid transformations that involve both rotations and translations. Let R_0 and R_1 be 3×3 rotation matrices and let $\mathbf{T}_0$ and $\mathbf{T}_1$ be 3×1 translation vectors. These may be combined into 4×4 homogeneous matrices

$$H_i = \begin{bmatrix} R_i & \mathbf{T}_i \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix}, \quad i = 0, 1 \tag{17}$$

The inverses of the matrices are

$$H_i^{-1} = \begin{bmatrix} R_i^\mathsf{T} & -R^\mathsf{T} \mathbf{T}_i \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix}, \quad i = 0, 1 \tag{18}$$

The problem now is to compute the geodesic path connecting H_0 and H_1. At first glance, your intuition might say that the path involves independently interpolating the translation terms using equation (1) and interpolating the rotation terms using equation (12). It turns out this is not correct.

A rigid transformation H corresponding to rotation matrix R and translation $\mathbf{T}$ can be written as an exponential,

$$\begin{bmatrix} R & \mathbf{T} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} = \exp\left(\begin{bmatrix} S & \mathbf{U} \\ \mathbf{0}^\mathsf{T} & 0 \end{bmatrix} \right) = \begin{bmatrix} \exp(S) & V\mathbf{U} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \tag{19}$$

where S is a skew-symmetric matrix and $\mathbf{U}$ is a 3×1 vector. The exponent has 6 independent parameters, 3 for rotation and 3 for translation. The rotation matrix is $R = \exp(S)$ and has the finite sum representation of equation (9). The matrix V and its inverse are

$$V = I + \left(\frac{1 - \cos\theta}{\theta^2} \right) S + \left(\frac{\theta - \sin\theta}{\theta^3} \right) S^2, \quad V^{-1} = I - \frac{1}{2} S + \frac{1}{\theta^2} \left(1 - \frac{\theta \sin\theta}{2(1 - \cos\theta)} \right) S^2 \tag{20}$$

The translation vector is $\mathbf{T} = V\mathbf{U}$. Therefore, given S and $\mathbf{U}$, we can compute the exponential map to produce R and $\mathbf{T}$. Similarly, given R and $\mathbf{T}$, we can compute the logarithm by extracting S for R and computing $\mathbf{U} = V^{-1}\mathbf{T}$.

The geodesic path connecting H_0 and H_1 is

$$F(t; H_0, H_1) = \exp(t \log(H_1 H_0^{-1}))H_0, \quad t \in [0, 1] \tag{21}$$

Compute

$$H_1 H_0^{-1} = \begin{bmatrix} R_1 & \mathbf{T}_1 \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \begin{bmatrix} R_0^\mathsf{T} & -R_0^\mathsf{T}\mathbf{T}_0 \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} = \begin{bmatrix} R_1 R_0^\mathsf{T} & \mathbf{T}_1 - R_1 R_0^\mathsf{T}\mathbf{T}_0 \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} = \exp\left(\begin{bmatrix} S & \mathbf{U} \\ \mathbf{0}^\mathsf{T} & 0 \end{bmatrix}\right) \tag{22}$$

where S and $\mathbf{U}$ are extracted as described in the previous paragraph using $R = R_1 R_0^\mathsf{T}$ and $\mathbf{T} = \mathbf{T}_1 - R\mathbf{T}_0$. The exponential map part of the geodesic path is

$$\exp(t \log(H_1 H_0^{-1})) = \exp\left(\begin{bmatrix} tS & t\mathbf{U} \\ \mathbf{0}^\mathsf{T} & 0 \end{bmatrix}\right) = \begin{bmatrix} \exp(tS) & tV(t)\mathbf{U} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \tag{23}$$

where

$$V(t) = I + \left(\frac{1 - \cos(t\theta)}{(t\theta)^2}\right)(tS) + \left(\frac{t\theta - \sin(t\theta)}{(t\theta)^3}\right)(tS)^2 \tag{24}$$

Observe that $\mathbf{T} = V(1)\mathbf{U}$, $\mathbf{U} = V(1)^{-1}\mathbf{T}$ and

$$tV(t) = tI + \left(\frac{1 - \cos(t\theta)}{\theta^2}\right)S + \left(\frac{t\theta - \sin(t\theta)}{\theta^3}\right)S^2 \tag{25}$$

and

$$\frac{d}{dt}(tV(t)) = I + \left(\frac{\sin(t\theta)}{\theta}\right)S + \left(\frac{1 - \cos(t\theta)}{\theta^2}\right)S^2 = \exp(tS) \tag{26}$$

4.1 Geodesic Path in Terms of Matrices

The geodesic path in terms of matrices is

$$\begin{aligned} F(t; H_0, H_1) &= \exp(t \log(H_1 H_0^{-1}))H_0 \\ &= \begin{bmatrix} \exp(tS)R_0 & \exp(tS)\mathbf{T}_0 + tV(t)\mathbf{U} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \\ &= \begin{bmatrix} G(t) & G(t)R_0^\mathsf{T}\mathbf{T}_0 + tV(t)\mathbf{U} \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \\ &= \begin{bmatrix} G(t) & \mathbf{P}(t) \\ \mathbf{0}^\mathsf{T} & 1 \end{bmatrix} \end{aligned} \tag{27}$$

where $G(t) = \exp(tS)R_0 = \exp(t\log(R_1R_0^{\mathsf{T}}))R_0$ is the geodesic path connecting the rotation matrices. The last equality defines the translation term $\mathbf{P}(t)$.

To verify the constant-speed condition of the geodesic path,

$$F'(t) = \begin{bmatrix} SG(t) & \mathbf{P}'(t) \\ \mathbf{0}^{\mathsf{T}} & 0 \end{bmatrix} = \begin{bmatrix} SG(t) & \exp(tS)S\mathbf{T}_0 + \exp(tS)\mathbf{U} \\ \mathbf{0}^{\mathsf{T}} & 0 \end{bmatrix} = \begin{bmatrix} SG(t) & \exp(tS)(S\mathbf{T}_0 + \mathbf{U}) \\ \mathbf{0}^{\mathsf{T}} & 0 \end{bmatrix} \tag{28}$$

Using the fact that the square of the Frobenius norm of a matrix is the sum of the squared lengths of the columns,

$$\|F'(t)\|^2 = \|SG(t)\|^2 + |\exp(tS)(S\mathbf{T}_0 + \mathbf{U})|^2 \tag{29}$$

The Frobenius norm on the left-hand side is applied to a 4×4 matrix. The Frobenius norm on the right-hand side is applied to a 3×3 matrix. We know that $\|SG(t)\|^2 = \|S\|^2$ because $G(t)$ is orthogonal. We also know that a rotation matrix $\exp(tS)$ does not change the length of the vectors it rotates, so $|\exp(tS)(S\mathbf{T}_0 + \mathbf{U})|^2 = |S\mathbf{T}_0 + \mathbf{U}|^2$; therefore,

$$\|F'(t)\|^2 = \|S\|^2 + |S\mathbf{T}_0 + \mathbf{U}|^2 \tag{30}$$

where the right-hand side is a constant.

The rotational component of the geodesic path connecting H_0 and H_1 can be computed as the slerp of quaternions representing the rotation matrices. The translational component is

$$\mathbf{P}(t) = \exp(tS)\mathbf{T}_0 + tV(t)\mathbf{U} \tag{31}$$

with derivative

$$\mathbf{P}'(t) = \exp(tS)(S\mathbf{T}_0 + \mathbf{U}) \tag{32}$$

Although the length of $\mathbf{P}'(t)$ is constant, the derivative itself is not constant. In fact, the derivative vector lives on a circle of revolution about the rotation axis, so $\mathbf{P}(t)$ cannot be a line segment.

Observe that when $R_0 = R_1 = I$, $F(t)$ of equation (21) reduces to that of equation (1). Similarly, when $\mathbf{T}_0 = \mathbf{T}_1 = \mathbf{0}$, $F(t)$ of equation (21) reduces to that of equation (12).

4.2 Geodesic Path in Terms of Dual Quaternions

Given a rigid transformation $\mathbf{Y} = R\mathbf{X} + \mathbf{T}$, where R is a rotation matrix and $\mathbf{T}$ is a translation vector, the dual quaternion that represents the transformation is $d = q + (pq/2)e$, where q is a unit-length quaternion representing R and where p is a quaternion with w-component zero and whose other components provided by $\mathbf{T}$. The symbolic term e is not zero, but in a product we use $e^2 = 0$ (dual quaternions are based on the concept of dual numbers).

In the context of the geodesic path connecting rigid transformations, define q_0 and q_1 to be the unit-length quaternions representing R_0 and R_1. Define $q(t)$ to be the unit-length quaternion representing the slerp of q_0 and q_1, namely, the quaternion representing the rotation $G(t) = \exp(t\log(R_1R_0^{\mathsf{T}}))R_0$ that occurs in equation (27). Let $p(t)$ be the quaternion with w-component zero and whose other components are provided by $\mathbf{P}(t)$ in equation (27). Define $b(t)$ to be the quaternion whose w-component is zero and whose other components are determined by $tV(t)\mathbf{U}$ in equation (27). The other part of the translation term in that equation is $G(t)R_0^{\mathsf{T}}\mathbf{T}_0$. Define quaternion a to have w-component of zero and whose other components are determined by $R_0^{\mathsf{T}}\mathbf{T}_0$; then $p(t) = q(t)\,a\,q^*(t) + b(t)$, where $q^*(t)$ is the conjugate of $q(t)$.

The dual quaternion that represents the geodesic path is

$$d(t) = q(t) + (p(t)q(t)/2)e = q(t) + [(q(t)\,a\,q^*(t) + b(t))q(t)/2]e = q(t) + [(q(t)a + b(t)q(t))/2]e \qquad (33)$$

where I have used $q^*(t)q(t) = 1$.

4.3 Implementation for the Matrix Representation

Pseudocode for the algorithm is provided in the next listing. I assume that you have available a basic matrix and vector algebra library for computing standard arithmetic operations on matrices and vectors.

```
struct RigidTransformation
{
    Matrix3x3 R;
    Vector3 T;
}

RigidTransformation InverseRigid(RigidTransformation H)
{
    RigidTransformation invH;
    invH.R = Transpose(H.R);
    invH.T = -invH.R * H.T;
    return invH;
}

Matrix3x3 Exp(Real t, Real theta, Matrix3x3 S)
{
    Real angle = t * theta, thetaSqr = theta * theta;
    return I + (sin(angle)/theta) * S + ((1 - cos(angle)) / thetaSqr) * S * S;
}

Matrix3x3 Log(Matrix3x3 R)
{
    Matrix3x3 S;

    Real arg = 0.5 * (R(0,0) + R(1,1) + R(2,2) - 1);  // in [-1,1]
    if (arg > -1)
    {
        if (arg < 1)
        {
            // 0 < angle < pi
            Real angle = acos(arg);
            Real sinAngle = sin(angle);
            Real c = 0.5 * angle / sinAngle;
            S = c * (R - Transpose(R));
        }
        else  // arg = 1, angle = 0
        {
            // R is the identity matrix and S is the zero matrix.
            S = 0;
        }
    }
    else  // arg = -1, angle = pi
    {
        // Knowing R+I is symmetric and wanting to avoid bias, we use
        // (R(i,j)+R(j,i))/2 for the off-diagonal entries rather than
        // R(i,j).
        Real s[3];
        if (R(0, 0) >= R(1, 1))
        {
            if (R(0, 0) >= R(2, 2))
            {
                // r00 is maximum diagonal term
                s[0] = R(0, 0) + 1;
                s[1] = 0.5 * (R(0, 1) + R(1, 0));
                s[2] = 0.5 * (R(0, 2) + R(2, 0));
            }
```

```
            else
            {
                // r22 is maximum diagonal term
                s[0] = 0.5 * (R(2, 0) + R(0, 2));
                s[1] = 0.5 * (R(2, 1) + R(1, 2));
                s[2] = R(2, 2) + 1;
            }
        }
        else
        {
            if (R(1, 1) >= R(2, 2))
            {
                // r11 is maximum diagonal term
                s[0] = 0.5 * (R(1, 0) + R(0, 1));
                s[1] = R(1, 1) + 1;
                s[2] = 0.5 * (R(1, 2) + R(2, 1));
            }
            else
            {
                // r22 is maximum diagonal term
                s[0] = 0.5 * (R(2, 0) + R(0, 2));
                s[1] = 0.5 * (R(2, 1) + R(1, 2));
                s[2] = R(2, 2) + 1;
            }
        }

        Real length = sqrt(s[0] * s[0] + s[1] * s[1] + s[2] * s[2]);
        if (length > 0)
        {
            Real adjust = pi * sqrt(0.5) / length;
            s[0] *= adjust;
            s[1] *= adjust;
            s[2] *= adjust;
        }
        else
        {
            s[0] = 0;
            s[1] = 0;
            s[2] = 0;
        }

        S(0, 0) = 0;      S(0, 1) = -s[2];  S(0, 2) = +s[1];
        S(1, 0) = +s[2];  S(1, 1) = 0;      S(1, 2) = -s[0];
        S(2, 0) = -s[1];  S(2, 1) = +s[0];  S(2, 2) = 0;
    }

    return S;
}

Matrix3x3 ComputeTTimesV(Real t, Real theta, Matrix3x3 S)
{
    if (theta > 0)
    {
        Real angle = t * theta, thetaSqr = theta * theta, thetaCub = theta * thetaSqr;
        Real c0 = (1 - cos(angle)) / thetaSqr;
        Real c1 = (angle - sin(angle)) / thetaCub;
        return t * I + c0 * S + c1 * S * S;
    }
    else
    {
        return t * I;
    }
}

Matrix3x3 ComputeInverseV1(Real theta, Matrix3x3 S)
{
    if (theta > 0)
    {
        Real thetaSqr = theta * theta;
        Real c = (1 - (theta * sin(theta))/(2 * (1 - cos(theta))) / thetaSqr;
        return I - 0.5 * S + c * S * S;
    }
```

```
    else
    {
        return I;
    }
}

RigidTransformation GeodesicPath(Real t, RigidTransformation H0, RigidTransformation H1)
{
    // If you plan on calling GeodesicPath for the same H0 and H1 but for multiple
    // t-values, the following terms can be precomputed and cached for use by the
    // last block of code.
    RigidTransformation H = H1 * InverseRigid(H0);
    Matrix3x3 S = Log(H.R);
    Real s0 = S(2, 1), s1 = S(0, 2), s2 = S(1, 0);
    Real theta = sqrt(s0 * s0 + s1 * s1 + s2 * s2);
    Matrix3x3 invV1 = ComputeInverseV1(theta, S);
    Vector3 U = invV1 * H.T;

    Matrix3x3 interpR = Exp(t, theta, S);
    Matrix3x3 interpTTimesV = ComputeTTimesV(t, theta, S);
    RigidTransformation interpH;
    interpH.R = interpR * H0.R;
    interpH.T = interpR * H0.T + interpTTimesV * U;
    return interpH;
}
```

References

[1] *Lie Groups for Computer Vision*,
http://ethaneade.com/lie_groups.pdf

[2] *Moving Along a Curve with Specified Speed*, 24 June 2010,
https://www.geometrictools.com/Documentation/MovingAlongCurveSpecifiedSpeed.pdf

[3] *Quaternion Algebra and Calculus*, 18 August 2010,
https://www.geometrictools.com/Documentation/Quaternions.pdf

[4] Ken Shoemake, *Animating rotation with quaternion calculus*, ACM SIGGRAPH 1987, Course Notes 10, Computer Animation: 3-D Motion, Specification, and Control.
https://dl.acm.org/citation.cfm?id=325242