A Robust Eigensolver for 3×3 Symmetric Matrices

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

Created: December 6, 2014
Last Modified: October 9, 2018

Contents

1 Introduction

Let A be a 3×3 symmetric matrix of real numbers. From linear algebra, we know that A has all real-valued eigenvalues and a full basis of eigenvectors. Let $D = \text{Diagonal}(\lambda_0, \lambda_1, \lambda_2)$ be the diagonal matrix whose diagonal entries are the eigenvalues. The eigenvalues are not necessarily distinct. Let $R = [\mathbf{U}_0\ \mathbf{U}_1\ \mathbf{U}_2]$ be an orthogonal matrix whose columns are linearly independent eigenvectors, ordered consistently with the diagonal entries of D. That is, $A\mathbf{U}_i = \lambda_i \mathbf{U}_i$. The eigendecomposition is $A = RDR^{\mathsf{T}}$.

The typical presentation in a linear algebra class shows that the eigenvalues are the roots of the cubic polynomial $\det(A - \lambda I) = 0$, where I is the 3×3 identity matrix. The left-hand side of the equation is the determinant of the matrix $A - \lambda I$. Closed-form equations exist for the roots of a cubic polynomial, so in theory one could compute the roots and for each one solve the equation $(A - \lambda I)\mathbf{U} = \mathbf{0}$ for nonzero vectors $\mathbf{U}$. Although correct theoretically, computing roots of the cubic polynomial using the closed-form equations is known to be a non-robust algorithm (generally).

2 An Iterative Algorithm

A matrix M is specified by $M = [m_{ij}]$ for $0 \le i \le 2$ and $0 \le 2$. The classical numerical approach is to use a Householder reflection matrix H to compute $B = H^{\mathsf{T}} A H$ so that $b_{02} = 0$; that is, B is a tridiagonal matrix. The matrix H is a reflection, so $H^{\mathsf{T}} = H$. A sequence of Givens rotations G_k are used to drive the superdiagonal entries to zero. This is an iterative process for which a termination condition is required. If n rotations are applied, we obtain

$$G_{n-1}^{\mathsf{T}} \cdots G_0^{\mathsf{T}} H^{\mathsf{T}} A H G_0 \cdots G_{n-1} = D' = D + E$$

where D' is a tridiagonal matrix. The matrix D is diagonal and the matrix E has entries that are sufficiently small that the diagonal entries of D are reasonable approximations to the eigenvalues of A. The orthogonal matrix $R' = HG_0 \cdots G_{n-1}$ has columns that are reasonable approximations to the eigenvectors of A.

The source code that implements this algorithm is in class SymmetricEigensolver found in the files

GeometricTools/GTEngine/Include/GteSymmetricEigensolver.{h, inl}

and is an implementation of Algorithm 8.2.3 (Symmetric QR Algorithm) described in *Matrix Computations, 2nd edition*, by G. H. Golub and C. F. Van Loan, The Johns Hopkins University Press, Baltimore MD, Fourth Printing 1993. Algorithm 8.2.1 (Householder Tridiagonalization) is used to reduce matrix A to tridiagonal D'. Algorithm 8.2.2 (Implicit Symmetric QR Step with Wilkinson Shift) is used for the iterative reduction from tridiagonal to diagonal. Numerically, we have errors $E = R^T AR - D$. Algorithm 8.2.3 mentions that one expects $|E|$ is approximately $\mu|A|$, where $|M|$ denotes the Frobenius norm of M and where μ is the unit roundoff for the floating-point arithmetic: 2^{-23} for float, which is FLT_EPSILON = 1.192092896e-7f, and 2^{-52} for double, which is DBL_EPSILON = 2.2204460492503131e-16.

The book uses the condition $|a(i,i+1)| \le \varepsilon |a(i,i) + a(i+1,i+1)|$ to determine when the reduction decouples to smaller problems. That is, when a superdiagonal term is effectively zero, the iterations may be applied separately to two tridiagonal submatrices. Our source code is implemented instead to deal with floating-point numbers,

```
sum = |a(i,i)| + |a(i+1,i+1)|;
if (sum + |a(i,i+1)| == sum)
{
    // The superdiagonal term a(i,i+1) is effectively zero.
}
```

That is, the superdiagonal term is small relative to its diagonal neighbors, and so it is effectively zero. The unit tests have shown that this interpretation of decoupling is effective.

3 A Variation of the Iterative Algorithm

The variation uses the Householder transformation to compute $B = H^{\mathsf{T}}AH$ where $b_{02} = 0$. Let $c = \cos\theta$ and $s = \sin\theta$ for some angle θ. The right-hand side is

$$
\begin{aligned}
H^{\mathsf{T}}AH &= \begin{bmatrix} c & s & 0 \\ s & -c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{01} & a_{11} & a_{12} \\ a_{02} & a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} c & s & 0 \\ s & -c & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} c^2 a_{00} + 2sca_{01} + s^2 a_{11} & sc(a_{00} - a_{11}) + (s^2 - c^2)a_{01} & ca_{02} + sa_{12} \\ sc(a_{00} - a_{11}) + (s^2 - c^2)a_{01} & c^2 a_{11} - 2sca_{01} + s^2 a_{00} & sa_{02} - ca_{12} \\ ca_{02} + sa_{12} & sa_{02} - ca_{12} & a_{22} \end{bmatrix} \\
&= \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{01} & b_{11} & b_{12} \\ b_{02} & b_{12} & b_{22} \end{bmatrix}
\end{aligned}
$$

We require $0 = b_{02} = ca_{02} + sa_{12} = (c, s) \cdot (a_{02}, a_{12})$, which occurs when $(c, s) = (a_{12}, -a_{02}) \,/\, \sqrt{a_{02}^2 + a_{12}^2}$.

Rather than using Givens rotations for the iterations, we may instead use reflection matrices. Suppose that $|b_{12}| \le |b_{01}|$. We will choose a sequence of reflection matrices to drive b_{12} to zero. Choose a reflection matrix

$$
G_1 = \begin{bmatrix} c_1 & 0 & -s_1 \\ s_1 & 0 & c_1 \\ 0 & 1 & 0 \end{bmatrix}
$$

where $c_1 = \cos\theta_1$ and $s_1 = \sin\theta_1$ for some angle θ_1. Consider the product

$$
P_1 = G_1^{\mathsf{T}} B G_1 = \begin{bmatrix} c_1^2 b_{00} + 2s_1c_1b_{01} + s_1^2 b_{11} & s_1 b_{12} & s_1c_1(b_{11} - b_{00}) + (c_1^2 - s_1^2)b_{01} \\ s_1 b_{12} & b_{22} & c_1 b_{12} \\ s_1c_1(b_{11} - b_{00}) + (c_1^2 - s_1^2)b_{01} & c_1 b_{12} & c_1^2 b_{11} - 2s_1c_1b_{01} + s_1^2 b_{00} \end{bmatrix}
$$

We need P_1 to be tridiagonal, which requires

$$0 = s_1 c_1 (b_{11} - b_{00}) + (c_1^2 - s_1^2) b_{01} = \sin(2\theta_1)(b_{11} - b_{00})/2 + \cos(2\theta_1) b_{01}$$

leading us to two possible choices

$$(\cos(2\theta_1), \sin(2\theta_1)) = \pm \frac{(b_{00} - b_{11}, 2b_{01})}{\sqrt{(b_{00} - b_{11})^2 + 4b_{01}^2}}$$

We must extract $c_1 = \cos\theta_1$ and $s_1 = \sin\theta$ from whichever solution we choose. In fact, we will choose $\cos(2\theta_1) \le 0$ so that $|c_1| \le |s_1| = s_1$. Let $\sigma = \mathrm{Sign}(b_{00} - b_{11})$; then

$$(\cos(2\theta_1), \sin(2\theta_1)) = \frac{(-|b_{00} - b_{11}|, -2\sigma b_{01})}{\sqrt{(b_{00} - b_{11})^2 + 4b_{01}^2}}, \quad \sin\theta_1 = \sqrt{\frac{1 - \cos(2\theta_1)}{2}}, \quad \cos\theta_1 = \frac{\sin(2\theta_1)}{2\sin\theta_1}$$

Notice that

$$|\cos\theta_1| = \sqrt{(1 + \cos(2\theta_1))/2} \le \sqrt{(1 - \cos(2\theta_1))/2} = \sin\theta_1$$

because we chose $\cos(2\theta_1) \le 0$.

The previous construction guarantees that $|\cos\theta_0| \le 1/\sqrt{2} < 1$. Let $P_1 = [p_{ij}^{(1)}]$; we now know $p_{12}^{(1)} = c_0 b_{12}$, so $|p_{12}^{(1)}| \le |b_{12}|/\sqrt{2}$. Our precondition for computing P_1 from B was that $|b_{12}| \le |b_{01}|$. A postcondition is that $|p_{12}^{(1)}| = |c_0 b_{12}| \le |s_0 b_{12}| = |p_{01}^{(1)}|$, which is the precondition if we construct and multiply by another reflection G_2 of the same form as G_1.

Define $P_0 = B$ and let $P_{i+1} = G_{i+1}^{\mathsf{T}} P_i G_{i+1}$ be the iterative process. The conclusion is that the $(1,2)$-entry of the output matrix is smaller than the $(1,2)$-entry of the input matrix, so indeed repeated iterations will drive the $(1,2)$-entry to zero. If $c_i = \cos\theta_i$ and $s_i = \sin\theta_i$, we have

$$\left|p_{12}^{(i+1)}\right| = \left|c_{i+1} p_{12}^{(i)}\right| \le 2^{-1/2} \left|p_{12}^{(i)}\right|$$

which implies

$$\left|p_{12}^{(i)}\right| \le 2^{-i/2} \left|b_{12}\right|, \quad i \ge 1$$

In the limit as $i \to \infty$, the $(1,2)$-entry is forced to zero. In fact the bounds here allow you to select the final i so that $2^{-i/2}|b_{12}|$ is a number whose nearest floating-point representation is zero. The number of iterations of the algorithm is limited by this i.

If instead we find that $|b_{01}| \le |b_{12}|$, we can choose the reflections to be of the form

$$G_i = \begin{bmatrix} 0 & 1 & 0 \\ c_1 & 0 & s_1 \\ -s_1 & 0 & c_1 \end{bmatrix}$$

Consider the product

$$P_1 = G_1^{\mathsf{T}} B G_1 = \begin{bmatrix} c_1^2 b_{11} - 2s_1c_1b_{12} + s_1^2 b_{22} & c_1 b_{01} & s_1c_1(b_{11} - b_{22}) + (c_1^2 - s_1^2) b_{12} \\ c_1 b_{01} & b_{00} & s_1 b_{01} \\ s_1c_1(b_{11} - b_{22}) + (c_1^2 - s_1^2) b_{12} & s_1 b_{01} & s_1^2 b_{11} + 2s_1c_1b_{12} + c_1^2 b_{22} \end{bmatrix}$$

Define $\sigma = \text{Sign}(b_{22} - b_{11})$ and choose

$$(\cos(2\theta_1), \sin(2\theta_1)) = \frac{(-|b_{22} - b_{11}|, -2\sigma b_{12})}{\sqrt{(b_{22} - b_{11})^2 + 4b_{12}^2}}, \quad \sin\theta_1 = \sqrt{\frac{1 - \cos(2\theta_1)}{2}}, \quad \cos\theta_1 = \frac{\sin(2\theta_1)}{2\sin\theta_1}$$

so that $|\cos\theta_1| = \sqrt{(1 + \cos(2\theta_1))/2} \leq \sqrt{(1 - \cos(2\theta_1))/2} = \sin\theta_1$. We can apply the G_i to drive the $(0,1)$-entry to zero.

A less aggressive approach is to use the condition mentioned previously that compares the superdiagonal entry to the diagonal entries using floating-point arithmetic.

4 Implementation of the Iterative Algorithm

The source code that implements the iterative algorithm for symmetric 3×3 matrices is in class SymmetricEigensolver3x3 found in the file

GeometricTools/GTEngine/Include/GteSymmetricEigensolver3x3.h

The code was written not to use any GTEngine object. The class interface is

```
#include <algorithm>
#include <array>
#include <cmath>
#include <limits>

template <typename Real>
class SymmetricEigensolver3x3
{
public:
    // The input matrix must be symmetric, so only the unique elements must
    // be specified: a00, a01, a02, a11, a12, and a22.
    //
    // If 'aggressive' is 'true', the iterations occur until a superdiagonal
    // entry is exactly zero.  If 'aggressive' is 'false', the iterations
    // occur until a superdiagonal entry is effectively zero compared to the
    // sum of magnitudes of its diagonal neighbors.  Generally, the
    // nonaggressive convergence is acceptable.
    //
    // The order of the eigenvalues is specified by sortType: -1 (decreasing),
    // 0 (no sorting), or +1 (increasing).  When sorted, the eigenvectors are
    // ordered accordingly, and {evec[0], evec[1], evec[2]} is guaranteed to
    // be a right-handed orthonormal set.  The return value is the number of
    // iterations used by the algorithm.

    int operator()(Real a00, Real a01, Real a02, Real a11, Real a12, Real a22,
        bool aggressive, int sortType, std::array<Real, 3>& eval,
        std::array<std::array<Real, 3>, 3>& evec) const;

private:
    // Update Q = Q*G in-place using G = {{c,0,-s},{s,0,c},{0,0,1}}.
    void Update0(Real Q[3][3], Real c, Real s) const;

    // Update Q = Q*G in-place using G = {{0,1,0},{c,0,s},{-s,0,c}}.
    void Update1(Real Q[3][3], Real c, Real s) const;

    // Update Q = Q*H in-place using H = {{c,s,0},{s,-c,0},{0,0,1}}.
    void Update2(Real Q[3][3], Real c, Real s) const;

    // Update Q = Q*H in-place using H = {{1,0,0},{0,c,s},{0,s,-c}}.
    void Update3(Real Q[3][3], Real c, Real s) const;
```

```
    // Normalize (u,v) robustly, avoiding floating-point overflow in the sqrt
    // call.  The normalized pair is (cs,sn) with cs <= 0.  If (u,v) = (0,0),
    // the function returns (cs,sn) = (-1,0).  When used to generate a
    // Householder reflection, it does not matter whether (cs,sn) or (-cs,-sn)
    // is used.  When generating a Givens reflection, cs = cos(2*theta) and
    // sn = sin(2*theta).  Having a negative cosine for the double-angle
    // term ensures that the single-angle terms c = cos(theta) and
    // s = sin(theta) satisfy |c| <= |s|.
    void GetCosSin(Real u, Real v, Real& cs, Real& sn) const;

    // The convergence test.  When 'aggressive' is 'true', the superdiagonal
    // test is "bSuper == 0".  When 'aggressive' is 'false', the superdiagonal
    // test is "|bDiag0| + |bDiag1| + |bSuper| == |bDiag0| + |bDiag1|, which
    // means bSuper is effectively zero compared to the sizes of the diagonal
    // entries.
    bool Converged(bool aggressive, Real bDiag0, Real bDiag1,
        Real bSuper) const;

    // Support for sorting the eigenvalues and eigenvectors.  The output
    // (i0,i1,i2) is a permutation of (0,1,2) so that d[i0] <= d[i1] <= d[i2].
    // The 'bool' return indicates whether the permutation is odd.  If it is
    // not, the handedness of the Q matrix must be adjusted.
    bool Sort(std::array<Real, 3> const& d, int& i0, int& i1, int& i2) const;
};
```

and the implementation is

```
template <typename Real>
int SymmetricEigensolver3x3<Real>::operator()(Real a00, Real a01,
    Real a02, Real a11, Real a12, Real a22, bool aggressive, int sortType,
    std::array<Real, 3>& eval, std::array<std::array<Real, 3>, 3>& evec) const
{
    // Compute the Householder reflection H and B = H*A*H, where b02 = 0.
    Real const zero = (Real)0, one = (Real)1, half = (Real)0.5;
    bool isRotation = false;
    Real c, s;
    GetCosSin(a12, -a02, c, s);
    Real Q[3][3] = { { c, s, zero }, { s, -c, zero }, { zero, zero, one } };
    Real term0 = c * a00 + s * a01;
    Real term1 = c * a01 + s * a11;
    Real b00 = c * term0 + s * term1;
    Real b01 = s * term0 - c * term1;
    term0 = s * a00 - c * a01;
    term1 = s * a01 - c * a11;
    Real b11 = s * term0 - c * term1;
    Real b12 = s * a02 - c * a12;
    Real b22 = a22;

    // Givens reflections, B' = G^T*B*G, preserve tridiagonal matrices.
    int const maxIteration = 2 * (1 + std::numeric_limits<Real>::digits -
        std::numeric_limits<Real>::min_exponent);
    int iteration;
    Real c2, s2;

    if (std::abs(b12) <= std::abs(b01))
    {
        Real saveB00, saveB01, saveB11;
        for (iteration = 0; iteration < maxIteration; ++iteration)
        {
            // Compute the Givens reflection.
            GetCosSin(half * (b00 - b11), b01, c2, s2);
            s = sqrt(half * (one - c2));  // >= 1/sqrt(2)
            c = half * s2 / s;

            // Update Q by the Givens reflection.
            Update0(Q, c, s);
            isRotation = !isRotation;

            // Update B <- Q^T*B*Q, ensuring that b02 is zero and |b12| has
            // strictly decreased.
```

```
        saveB00 = b00;
        saveB01 = b01;
        saveB11 = b11;
        term0 = c * saveB00 + s * saveB01;
        term1 = c * saveB01 + s * saveB11;
        b00 = c * term0 + s * term1;
        b11 = b22;
        term0 = c * saveB01 - s * saveB00;
        term1 = c * saveB11 - s * saveB01;
        b22 = c * term1 - s * term0;
        b01 = s * b12;
        b12 = c * b12;

        if (Converged(aggressive, b00, b11, b01))
        {
            // Compute the Householder reflection.
            GetCosSin(half * (b00 - b11), b01, c2, s2);
            s = sqrt(half * (one - c2));
            c = half * s2 / s;  // >= 1/sqrt(2)

            // Update Q by the Householder reflection.
            Update2(Q, c, s);
            isRotation = !isRotation;

            // Update D = Q^T*B*Q.
            saveB00 = b00;
            saveB01 = b01;
            saveB11 = b11;
            term0 = c * saveB00 + s * saveB01;
            term1 = c * saveB01 + s * saveB11;
            b00 = c * term0 + s * term1;
            term0 = s * saveB00 - c * saveB01;
            term1 = s * saveB01 - c * saveB11;
            b11 = s * term0 - c * term1;
            break;
        }
    }
}
else
{
    Real saveB11, saveB12, saveB22;
    for (iteration = 0; iteration < maxIteration; ++iteration)
    {
        // Compute the Givens reflection.
        GetCosSin(half * (b22 - b11), b12, c2, s2);
        s = sqrt(half * (one - c2));  // >= 1/sqrt(2)
        c = half * s2 / s;

        // Update Q by the Givens reflection.
        Update1(Q, c, s);
        isRotation = !isRotation;

        // Update B <- Q^T*B*Q, ensuring that b02 is zero and |b12| has
        // strictly decreased.  MODIFY...
        saveB11 = b11;
        saveB12 = b12;
        saveB22 = b22;
        term0 = c * saveB22 + s * saveB12;
        term1 = c * saveB12 + s * saveB11;
        b22 = c * term0 + s * term1;
        b11 = b00;
        term0 = c * saveB12 - s * saveB22;
        term1 = c * saveB11 - s * saveB12;
        b00 = c * term1 - s * term0;
        b12 = s * b01;
        b01 = c * b01;

        if (Converged(aggressive, b11, b22, b12))
        {
            // Compute the Householder reflection.
            GetCosSin(half * (b11 - b22), b12, c2, s2);
            s = sqrt(half * (one - c2));
```

```
            c = half * s2 / s;  // >= 1/sqrt(2)

            // Update Q by the Householder reflection.
            Update3(Q, c, s);
            isRotation = !isRotation;

            // Update D = Q^T*B*Q.
            saveB11 = b11;
            saveB12 = b12;
            saveB22 = b22;
            term0 = c * saveB11 + s * saveB12;
            term1 = c * saveB12 + s * saveB22;
            b11 = c * term0 + s * term1;
            term0 = s * saveB11 - c * saveB12;
            term1 = s * saveB12 - c * saveB22;
            b22 = s * term0 - c * term1;
            break;
        }
    }
}

std::array<Real, 3> diagonal = { b00, b11, b22 };
int i0, i1, i2;
if (sortType >= 1)
{
    // diagonal[i0] <= diagonal[i1] <= diagonal[i2]
    bool isOdd = Sort(diagonal, i0, i1, i2);
    if (!isOdd)
    {
        isRotation = !isRotation;
    }
}
else if (sortType <= -1)
{
    // diagonal[i0] >= diagonal[i1] >= diagonal[i2]
    bool isOdd = Sort(diagonal, i0, i1, i2);
    std::swap(i0, i2);  // (i0,i1,i2)->(i2,i1,i0) is odd
    if (isOdd)
    {
        isRotation = !isRotation;
    }
}
else
{
    i0 = 0;
    i1 = 1;
    i2 = 2;
}

eval[0] = diagonal[i0];
eval[1] = diagonal[i1];
eval[2] = diagonal[i2];
evec[0][0] = Q[0][i0];
evec[0][1] = Q[1][i0];
evec[0][2] = Q[2][i0];
evec[1][0] = Q[0][i1];
evec[1][1] = Q[1][i1];
evec[1][2] = Q[2][i1];
evec[2][0] = Q[0][i2];
evec[2][1] = Q[1][i2];
evec[2][2] = Q[2][i2];

// Ensure the columns of Q form a right-handed set.
if (!isRotation)
{
    for (int j = 0; j < 3; ++j)
    {
        evec[2][j] = -evec[2][j];
    }
}

return iteration;
```

```
}

template <typename Real>
void SymmetricEigensolver3x3<Real>::Update0(Real Q[3][3], Real c, Real s) const
{
    for (int r = 0; r < 3; ++r)
    {
        Real tmp0 = c * Q[r][0] + s * Q[r][1];
        Real tmp1 = Q[r][2];
        Real tmp2 = c * Q[r][1] - s * Q[r][0];
        Q[r][0] = tmp0;
        Q[r][1] = tmp1;
        Q[r][2] = tmp2;
    }
}

template <typename Real>
void SymmetricEigensolver3x3<Real>::Update1(Real Q[3][3], Real c, Real s) const
{
    for (int r = 0; r < 3; ++r)
    {
        Real tmp0 = c * Q[r][1] - s * Q[r][2];
        Real tmp1 = Q[r][0];
        Real tmp2 = c * Q[r][2] + s * Q[r][1];
        Q[r][0] = tmp0;
        Q[r][1] = tmp1;
        Q[r][2] = tmp2;
    }
}

template <typename Real>
void SymmetricEigensolver3x3<Real>::Update2(Real Q[3][3], Real c, Real s) const
{
    for (int r = 0; r < 3; ++r)
    {
        Real tmp0 = c * Q[r][0] + s * Q[r][1];
        Real tmp1 = s * Q[r][0] - c * Q[r][1];
        Q[r][0] = tmp0;
        Q[r][1] = tmp1;
    }
}

template <typename Real>
void SymmetricEigensolver3x3<Real>::Update3(Real Q[3][3], Real c, Real s) const
{
    for (int r = 0; r < 3; ++r)
    {
        Real tmp0 = c * Q[r][1] + s * Q[r][2];
        Real tmp1 = s * Q[r][1] - c * Q[r][2];
        Q[r][1] = tmp0;
        Q[r][2] = tmp1;
    }
}

template <typename Real>
void SymmetricEigensolver3x3<Real>::GetCosSin(Real u, Real v, Real& cs, Real& sn) const
{
    Real maxAbsComp = std::max(std::abs(u), std::abs(v));
    if (maxAbsComp > (Real)0)
    {
        u /= maxAbsComp;  // in [-1,1]
        v /= maxAbsComp;  // in [-1,1]
        Real length = sqrt(u*u + v*v);
        cs = u / length;
        sn = v / length;
        if (cs > (Real)0)
        {
            cs = -cs;
            sn = -sn;
        }
    }
    else
```

```
    {
        cs = (Real)-1;
        sn = (Real)0;
    }
}

template <typename Real>
bool SymmetricEigensolver3x3<Real>::Converged(bool aggressive, Real bDiag0, Real bDiag1, Real bSuper) const
{
    if (aggressive)
    {
        return bSuper == (Real)0;
    }
    else
    {
        Real sum = std::abs(bDiag0) + std::abs(bDiag1);
        return sum + std::abs(bSuper) == sum;
    }
}

template <typename Real>
bool SymmetricEigensolver3x3<Real>::Sort(std::array<Real, 3> const& d, int& i0, int& i1, int& i2) const
{
    bool odd;
    if (d[0] < d[1])
    {
        if (d[2] < d[0])
        {
            i0 = 2; i1 = 0; i2 = 1; odd = true;
        }
        else if (d[2] < d[1])
        {
            i0 = 0; i1 = 2; i2 = 1; odd = false;
        }
        else
        {
            i0 = 0; i1 = 1; i2 = 2; odd = true;
        }
    }
    else
    {
        if (d[2] < d[1])
        {
            i0 = 2; i1 = 1; i2 = 0; odd = false;
        }
        else if (d[2] < d[0])
        {
            i0 = 1; i1 = 2; i2 = 0; odd = true;
        }
        else
        {
            i0 = 1; i1 = 0; i2 = 2; odd = false;
        }
    }
    return odd;
}
```

5 A Noniterative Algorithm

An algorithm that is reasonably robust in practice is provided at the Wikipedia page Eigenvalue Algorithm in the section entitled **3 × 3 matrices**. The algorithm involves converting a cubic polynomial to a form that allows the application of a trigonometric identity in order to obtain closed-form expressions for the roots. This topic is quite old; the Wikipedia page Cubic Function summarizes the history.[1] Although the topic is old, the current-day emphasis is usually the robustness of the algorithm when using floating-point arithmetic. Knowing that real-valued symmetric matrices have only real-valued eigenvalues, we know that the characteristic cubic polynomial has roots that are all real valued. The Wikipedia discussion includes pseudocode and a reference to a 1961 paper in the *Communications of the ACM* entitled *Eigenvalues of a symmetric* 3×3 *matrix*. In my opinion, important details are omitted from the Wikipedia page. In particular, it is helpful to visualize the graph of the cubic polynomial $\det(\beta I - B) = \beta^3 - 3\beta - \det(B)$ in order to understand the location of the polynomial roots. This information is essential to compute eigenvectors when a real-valued root is repeated. The Wikipedia page mentions briefly the mathematical details for generating the eigenvectors, but the discussion does not make it clear how to do so robustly in a computer program.

I will use the notation that occurs at the Wikipedia page, except that the pseudocode will use 0-based indexing of vectors and matrices rather than 1-based indexing. Let A be a real-valued symmetric 3×3 matrix,

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{01} & a_{11} & a_{12} \\ a_{02} & a_{12} & a_{22} \end{bmatrix} \tag{1}$$

The *trace* of a matrix M, denoted $\mathrm{tr}(M)$, is the sum of the diagonal entries of M. The determinant of M is denoted $\det(M)$. The characteristic polynomial for A is

$$f(\alpha) = \det(\alpha I - A) = \alpha^3 - c_2\alpha^2 + c_1\alpha - c_0 \tag{2}$$

where I is the 3×3 identity matrix and where

$$\begin{aligned} c_2 &= a_{00} + a_{11} + a_{22} = \mathrm{tr}(A) \\ c_1 &= (a_{00}a_{11} - a_{01}^2) + (a_{00}a_{22} - a_{02}^2) + (a_{11}a_{22} - a_{12}^2) = \left(\mathrm{tr}^2(A) - \mathrm{tr}(A^2)\right)/2 \\ c_0 &= a_{00}a_{11}a_{22} + 2a_{01}a_{02}a_{12} - a_{00}a_{12}^2 - a_{11}a_{02}^2 - a_{22}a_{01}^2 = \det(A) \end{aligned} \tag{3}$$

Given scalars $p > 0$ and q, define the matrix B by $A = pB + qI$. It is the case that A and B have the same eigenvectors. If $\mathbf{v}$ is an eigenvector of A, then by definition $A\mathbf{v} = \alpha\mathbf{v}$ where α is an eigenvalue of A; moreover,

$$\alpha\mathbf{v} = A\mathbf{v} = pB\mathbf{v} + q\mathbf{v} \tag{4}$$

which implies $B\mathbf{v} = ((\alpha - q)/p)\mathbf{v}$. Thus, $\mathbf{v}$ is an eigenvector of B with corresponding eigenvalue $\beta = (\alpha - q)/p$, or equivalently $\alpha = p\beta + q$. If we compute the eigenvalues and eigenvectors of B, we can then obtain the eigenvalues and eigenvectors of B.

[1] My first exposure to the trigonometric approach was in high school when reading parts of my first-purchased mathematics book *CRC Standard Mathematic Tables*, the 19th edition published in 1971 by The Chemical Rubber Company. You might recognize the CRC acronym as part of *CRC Press*, a current-day publisher of books. The book title hides the fact that there is quite a bit of (old) mathematics in the book. However, the book does have many tables of numbers because at the time the common tools for computing were (1) pencil and paper, (2) table lookups, and (3) a slide rule. Yes, I had my very own slide rule.

5.1 Computing the Eigenvalues

Define $q = \mathrm{tr}(A)/3$ and $p = \sqrt{\mathrm{tr}\left((A - qI)^2\right)/6}$ so that $B = (A - qI)/p$. The Wikipedia discussion does not point out that this is defined only when $p \neq 0$. If A is a scalar multiple of the identity, then $p = 0$. On the other hand, the pseudocode at the Wikipedia page has special handling for when A is a diagonal matrix, which happens to include the case $p = 0$. The remainder of the construction here assumes $p \neq 0$. Some algebraic manipulation will show that $\mathrm{tr}(B) = 0$ and the characteristic polynomial is

$$g(\beta) = \det(\beta I - B) = \beta^3 - 3\beta - \det(B) \tag{5}$$

Choosing $\beta = 2\cos(\theta)$, the characteristic polynomial becomes

$$g(\theta) = 2\left(4\cos^3(\theta) - 3\cos(\theta)\right) - \det(B) \tag{6}$$

Using the trigonometric identity $\cos(3\theta) = 4\cos^3(\theta) - 3\cos(\theta)$, we obtain

$$g(\theta) = 2\cos(3\theta) - \det(B) \tag{7}$$

The roots of g are obtained by solving for θ in the equation $\cos(3\theta) = \det(B)/2$. Knowing that B has only real-valued roots, it must be that θ is real-valued which implies $|\cos(3\theta)| \leq 1$.[2] This additionally implies that $|\det(B)| \leq 2$. The real-valued roots of $g(\theta)$ are

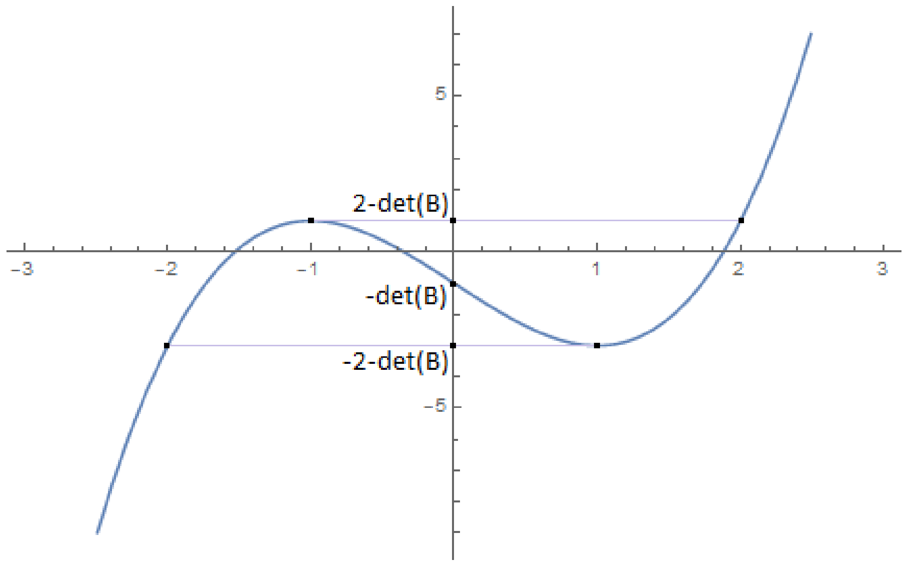
$$\beta = 2\cos\left(\theta + \frac{2\pi k}{3}\right), \quad k = 0, 1, 2 \tag{8}$$

where $\theta = \arccos(\det(B)/2)/3$, and all roots are in the interval $[-2, 2]$.

Let us now examine in greater detail the function $g(\beta)$. The first derivative is $g'(\beta) = 3\beta^2 - 3$, which is zero when $\beta = \pm 1$. The second derivative is $g''(\beta) = 3\beta$, which is not zero when $g'(\beta) = 0$. Therefore, $\beta = -1$ produces a local maximum of g and $\beta = +1$ produces a local minimum of g. The graph of g has an inflection point at $\beta = 0$ because $g''(0) = 0$. A typical graph is shown in Figure 1.

[2] When working with complex numbers, the magnitude of the cosine function can exceed 1.

Figure 1. The graph of $g(\beta) = \beta^3 - 3\beta - 1$ where $\det(B) = 1$. The graph was drawn using Mathematica (Wolfram Research, Inc., Mathematica, Version 11.0, Champaign, IL (2016)) but with manual modifications to emphasize some key graph features.



The roots of $g(\beta)$ are indeed in the interval $[-2, 2]$. Generally, $g(-2) = g(1) = -2 - \det(B)$, $g(-1) = g(2) = 2 - \det(B)$, and $g(0) = -\det(B)$. Table 1 shows bounds for the roots. The roots are named so that $\beta_0 \le \beta_1 \le \beta_2$.

Table 1. Bounds on the roots of $g(\beta)$. The roots are ordered by $\beta_0 \le \beta_1 \le \beta_2$.

$\det(B) > 0$	$\theta + 2\pi/3 \in (2\pi/3, 5\pi/6)$	$\cos(\theta + 2\pi/3) \in (-\sqrt{3}/2, -1/2)$	$\beta_0 \in (-\sqrt{3}, -1)$
	$\theta + 4\pi/3 \in (4\pi/3, 3\pi/2)$	$\cos(\theta + 4\pi/3) \in (-1/2, 0)$	$\beta_1 \in (-1, 0)$
	$\theta \in (0, \pi/6)$	$\cos(\theta) \in (\sqrt{3}/2, 1)$	$\beta_2 \in (\sqrt{3}, 2)$
$\det(B) < 0$	$\theta + 2\pi/3 \in (5\pi/6, \pi)$	$\cos(\theta + 2\pi/3) \in (-1, -\sqrt{3}/2)$	$\beta_0 \in (-2, -\sqrt{3})$
	$\theta + 4\pi/3 \in (3\pi/2, 5\pi/3)$	$\cos(\theta + 4\pi/3) \in (0, 1/2)$	$\beta_1 \in (0, 1)$
	$\theta \in (\pi/6, \pi/3)$	$\cos(\theta) \in (1/2, \sqrt{3}/2)$	$\beta_2 \in (1, \sqrt{3})$
$\det(B) = 0$	$\theta + 2\pi/3 = 5\pi/6$	$\cos(\theta + 2\pi/3) = -\sqrt{3}/2$	$\beta_0 = -\sqrt{3}$
	$\theta + 4\pi/3 = 3\pi/2$	$\cos(\theta + 4\pi/3) = 0$	$\beta_1 = 0$
	$\theta = \pi/6$	$\cos(\theta) = \sqrt{3}/2$	$\beta_2 = \sqrt{3}$

Because $\mathrm{tr}(B) = 0$, we also know $\beta_0 + \beta_1 + \beta_2 = 0$.

Computing the eigenvectors for distinct *and well-separated* eigenvalues is generally robust. However, if a root is repeated or two roots are nearly the same numerically, issues can occur when computing the 2-dimensional eigenspace. Fortunately, the special form of $g(\beta)$ leads to a robust algorithm.

5.2 Computing the Eigenvectors

The discussion here is based on constructing an eigenvector for β_0 when $\beta_0 < 0 < \beta_1 \le \beta_2$. The implementation must also handle the construction of an eigenvector for β_2 when $\beta_0 \le \beta_1 < 0 < \beta_2$. The corresponding eigenvalues of A are α_0, α_1, and α_2, ordered as $\alpha_0 \le \alpha_1 \le \alpha_2$. The eigenvectors for α_0 are solutions to $(A - \alpha_0 I)\mathbf{v} = \mathbf{0}$. The matrix $A - \alpha_0 I$ is singular (by definition of eigenvalue) and in this case has rank 2; that is, two rows are linearly independent. Write the system of equations as shown next,

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0} = (A - \alpha_0 I)\mathbf{v} = \begin{bmatrix} \mathbf{r}_0^\mathsf{T} \\ \mathbf{r}_1^\mathsf{T} \\ \mathbf{r}_2^\mathsf{T} \end{bmatrix} \mathbf{v} = \begin{bmatrix} \mathbf{r}_0 \cdot \mathbf{v} \\ \mathbf{r}_1 \cdot \mathbf{v} \\ \mathbf{r}_2 \cdot \mathbf{v} \end{bmatrix} \tag{9}$$

where $\mathbf{r}_i$ are the 3×1 vectors whose transposes are the rows of the matrix $A - \alpha_0 I$. Assuming the first two rows are linearly independent, the conditions $\mathbf{r}_0 \cdot \mathbf{v} = 0$ and $\mathbf{r}_1 \cdot \mathbf{v} = 0$ imply $\mathbf{v}$ is perpendicular to both rows. Consequently, $\mathbf{v}$ is parallel to the cross product $\mathbf{r}_0 \times \mathbf{r}_1$. We can normalize the cross product to obtain a unit-length eigenvector.

It is unknown which two rows of the matrix are linearly independent, so we must figure out which two rows to choose. If two rows are nearly parallel, the cross product will have length nearly zero. For numerical robustness, this suggests that we look at the three possible cross products of rows and choose the pair of rows for which the cross product has largest length. Alternatively, we could apply elementary row and column operations to reduce the matrix so that the last row is (numerically) the zero vector; this effectively is the algorithm of using Gaussian elimination with full pivoting. In the pseudocode shown in Listing 1, the cross product approach is used.

Listing 1. Pseudocode for computing the eigenvector corresponding to the eigenvalue α_0 of A that was generated from the root β_0 of the cubic polynomial for B that has multiplicity 1.

```
void ComputeEigenvector0(Matrix3x3 A, Real eigenvalue0, Vector3& eigenvector0)
{
    Vector3 row0( A(0,0) - eigenvalue0, A(0,1), A(0,2) );
    Vector3 row1( A(0,1), A(1,1) - eigenvalue0, A(1,2) );
    Vector3 row2( A(0,2), A(1,2), A(2,2) - eigenvalue0 );
    Vector3 r0xr1 = Cross(row0, row1);
    Vector3 r0xr2 = Cross(row0, row2);
    Vector3 r1xr2 = Cross(row1, row2);
    Real d0 = Dot(r0xr1, r0xr1);
    Real d1 = Dot(r0xr2, r0xr2);
    Real d2 = Dot(r1xr2, r1xr2);
    Real dmax = d0;
    int imax = 0;
    if (d1 > dmax) { dmax = d1; imax = 1; }
    if (d2 > dmax) { imax = 2; }
    if (imax == 0)
    {
        eigenvector0 = r0xr1 / sqrt(d0);
    }
    else if (imax == 1)
```

```
    {
        eigenvector0 = r0×r2 / sqrt(d1);
    }
    else
    {
        eigenvector0 = r1×r2 / sqrt(d2);
    }
}
```

If the other two eigenvalues are well separated, the algorithm of Listing 1 may be used for each eigenvalue. However, if the two eigenvalues are nearly equal, Listing 1 can have numerical issues. The problem is that theoretically when the eigenvalue is repeated ($\alpha_1 = \alpha_2$), the rank of $A - \alpha_1 I$ is 1. The cross products of any pair of rows is the zero vector. Determining the rank of a matrix numerically is problematic.

A different approach is used to compute an eigenvector of $A - \alpha_1 I$. We know that the eigenvectors corresponding to α_1 and α_2 are perpendicular to the eigenvector $\mathbf{W}$ of α_0. Compute unit-length vectors $\mathbf{U}$ and $\mathbf{V}$ such that $\{\mathbf{U}, \mathbf{V}, \mathbf{W}\}$ is a right-handed orthonormal set; that is, the vectors are all unit length, mutually perpendicular, and $\mathbf{W} = \mathbf{U} \times \mathbf{V}$. The computations can be done robustly when using floating-point arithmetic, as shown in Listing 2.

Listing 2. Robust computation of $\mathbf{U}$ and $\mathbf{V}$ for a specified $\mathbf{W}$.

```
void ComputeOrthogonalComplement(Vector3 W, Vector3& U, Vector3& V)
{
    Real invLength;
    if (fabs(W[0]) > fabs(W[1]))
    {
        // The component of maximum absolute value is either W[0] or W[2].
        invLength = 1 / sqrt(W[0] * W[0] + W[2] * W[2]);
        U = Vector3(-W[2] * invLength, 0, +W[0] * invLength);
    }
    else
    {
        // The component of maximum absolute value is either W[1] or W[2].
        invLength = 1 / sqrt(W[1] * W[1] + W[2] * W[2]);
        U = Vector3(0, +W[2] * invLength, -W[1] * invLength);
    }
    Vector3 V = Cross(W, U);
}
```

A unit-length eigenvector $\mathbf{E}$ of $A - \alpha_1 I$ must be a circular combination of $\mathbf{U}$ and $\mathbf{V}$; that is, $\mathbf{E} = x_0\mathbf{U} + x_1\mathbf{V}$ for some choice of x_0 and x_1 with $x_0^2 + x_1^2 = 1$. There are exactly two such unit-length eigenvectors when $\alpha_1 \neq \alpha_2$ but infinitely many when $\alpha_1 = \alpha_2$.

Define the 3×2 matrix $J = [\mathbf{U}\ \mathbf{V}]$ whose columns are the specified vectors. Define the 2×2 symmetric matrix $M = J^{\mathsf{T}}(A - \alpha_1 I)J$. Define the 2×1 vector $\mathbf{X} = J\mathbf{E}$ whose rows are x_0 and x_1. The 3×3 linear system $(A - \alpha_1 I)\mathbf{E} = \mathbf{0}$ reduces to the 2×2 linear system $M\mathbf{X} = \mathbf{0}$. M has rank 1 when $\alpha_1 \neq \alpha_2$, in which case M is not the zero matrix, or rank 0 when $\alpha_1 = \alpha_2$, in which case M is the zero matrix. Numerically, we need to trap the cases properly.

In the event M is not the zero matrix, we can select the row of M that has largest length and normalize it. A solution $\mathbf{X}$ is perpendicular to the normalized row, and we can choose $\mathbf{X}$ so that it is unit length. The normalization first factors out the largest component of the row and discards it to avoid floating-point underflow or overflow when computing the length of the row. If M is the zero matrix, then any choice of unit-length $\mathbf{X}$ suffices. Listing 3 contains pseudocode for computing a unit-length eigenvector corresponding to α_1.

Listing 3. Pseudocode for computing the eigenvector corresponding to the eigenvalue α_1 of A that was generated from the root β_1 of the cubic polynomial for B that potentially has multiplicity 2.

```
void ComputeEigenvector1(Matrix3x3 A, Vector3 eigenvector0, Real eigenvalue1, Vector3& eigenvector1)
{
    Vector3 AU = A*U, AV = A*V;
    float m00 = Dot(U, AU) - eigenvalue1, m01 = Dot(U, AV), m11 = Dot(V, AV) - eigenvalue1;
    float absM00 = fabs(m00), absM01 = fabs(m01), absM11 = fabs(m11);
    if (absM00 >= absM11)
    {
        float maxAbsComp = max(absM00, absM01);
        if (maxAbsComp > 0)
        {
            if (absM00 >= absM01)
            {
                m01 /= m00;  m00 = 1 / sqrt(1 + m01 * m01);  m01 *= m00;
            }
            else
            {
                m00 /= m01;  m01 = 1 / sqrt(1 + m00 * m00);  m00 *= m01;
            }
            eigenvector1 = m01 * U - m00 * V;
        }
        else
        {
            eigenvector1 = U;
        }
    }
    else
    {
        float maxAbsComp = max(absM11, absM01);
        if (maxAbsComp > 0)
        {
            if (absM11 >= absM01)
            {
                m01 /= m11;  m11 = 1 / sqrt(1 + m01 * m01);  m01 *= m11;
            }
            else
            {
                m11 /= m01;  m01 = 1 / sqrt(1 + m11 * m11);  m11 *= m01;
            }
            eigenvector1 = m11 * U - m01 * V;
        }
        else
        {
            eigenvector1 = U;
        }
    }
}
```

The remaining eigenvector must be perpendicular to the first two computed eigenvectors as shown in Listing 4.

Listing 4. The remaining eigenvector is simply a cross product of the first two computed eigenvectors, and it is guaranteed to be unit length (within numerical tolerance).

```
ComputeEigenvector0(A, eigenvalue0, eigenvector0);
ComputeEigenvector1(A, eigenvector0, eigenvalue1, eigenvector1);
eigenvector2 = Cross(eigenvector0, eigenvector1);
```

6 Implementation of the Noniterative Algorithm

The source code that implements the iterative algorithm for symmetric 3×3 matrices is in class NISymmetricEigensolver3x3 found in the file

GeometricTools/GTEngine/Include/GteSymmetricEigensolver3x3.h

The code was written not to use any GTEngine object. The class interface is

```
#include <algorithm>
#include <array>
#include <cmath>
#include <limits>

template <typename Real>
class NISymmetricEigensolver3x3
{
public:
    // The input matrix must be symmetric, so only the unique elements must
    // be specified: a00, a01, a02, a11, a12, and a22.  The eigenvalues are
    // sorted in ascending order: eval0 <= eval1 <= eval2.

    void operator()(Real a00, Real a01, Real a02, Real a11, Real a12, Real a22,
        std::array<Real, 3>& eval, std::array<std::array<Real, 3>, 3>& evec) const;

private:
    static std::array<Real, 3> Multiply(Real s, std::array<Real, 3> const& U);
    static std::array<Real, 3> Subtract(std::array<Real, 3> const& U, std::array<Real, 3> const& V);
    static std::array<Real, 3> Divide(std::array<Real, 3> const& U, Real s);
    static Real Dot(std::array<Real, 3> const& U, std::array<Real, 3> const& V);
    static std::array<Real, 3> Cross(std::array<Real, 3> const& U, std::array<Real, 3> const& V);

    void ComputeOrthogonalComplement(std::array<Real, 3> const& W,
        std::array<Real, 3>& U, std::array<Real, 3>& V) const;

    void ComputeEigenvector0(Real a00, Real a01, Real a02, Real a11, Real a12, Real a22,
        Real& eval0, std::array<Real, 3>& evec0) const;

    void ComputeEigenvector1(Real a00, Real a01, Real a02, Real a11, Real a12, Real a22,
        std::array<Real, 3> const& evec0, Real& eval1, std::array<Real, 3>& evec1) const;
};
```

and the implementation is

```
template <typename Real>
void NISymmetricEigensolver3x3<Real>::operator() (Real a00, Real a01, Real a02,
    Real a11, Real a12, Real a22, std::array<Real, 3>& eval,
    std::array<std::array<Real, 3>, 3>& evec) const
{
```

```
// Precondition the matrix by factoring out the maximum absolute value
// of the components. This guards against floating-point overflow when
// computing the eigenvalues.
Real max0 = std::max(fabs(a00), fabs(a01));
Real max1 = std::max(fabs(a02), fabs(a11));
Real max2 = std::max(fabs(a12), fabs(a22));
Real maxAbsElement = std::max(std::max(max0, max1), max2);
if (maxAbsElement == (Real)0)
{
    // A is the zero matrix.
    eval[0] = (Real)0;
    eval[1] = (Real)0;
    eval[2] = (Real)0;
    evec[0] = { (Real)1, (Real)0, (Real)0 };
    evec[1] = { (Real)0, (Real)1, (Real)0 };
    evec[2] = { (Real)0, (Real)0, (Real)1 };
    return;
}

Real invMaxAbsElement = (Real)1 / maxAbsElement;
a00 *= invMaxAbsElement;
a01 *= invMaxAbsElement;
a02 *= invMaxAbsElement;
a11 *= invMaxAbsElement;
a12 *= invMaxAbsElement;
a22 *= invMaxAbsElement;

Real norm = a01 * a01 + a02 * a02 + a12 * a12;
if (norm > (Real)0)
{
    // Compute the eigenvalues of A.

    // In the PDF mentioned previously, B = (A - q*I)/p, where q = tr(A)/3
    // with tr(A) the trace of A (sum of the diagonal entries of A) and where
    // p = sqrt(tr((A - q*I)^2)/6).
    Real q = (a00 + a11 + a22) / (Real)3;

    // The matrix A - q*I is represented by the following, where b00, b11 and
    // b22 are computed after these comments,
    //   +-           -+
    //   | b00 a01 a02 |
    //   | a01 b11 a12 |
    //   | a02 a12 b22 |
    //   +-           -+
    Real b00 = a00 - q;
    Real b11 = a11 - q;
    Real b22 = a22 - q;

    // The is the variable p mentioned in the PDF.
    Real p = std::sqrt((b00 * b00 + b11 * b11 + b22 * b22 + norm * (Real)2) / (Real)6);

    // We need det(B) = det((A - q*I)/p) = det(A - q*I)/p^3. The value
    // det(A - q*I) is computed using a cofactor expansion by the first
    // row of A - q*I. The cofactors are c00, c01 and c02 and the
    // determinant is b00*c00 - a01*c01 + a02*c02. The det(B) is then
    // computed finally by the division with p^3.
    Real c00 = b11 * b22 - a12 * a12;
    Real c01 = a01 * b22 - a12 * a02;
    Real c02 = a01 * a12 - b11 * a02;
    Real det = (b00 * c00 - a01 * c01 + a02 * c02) / (p * p * p);

    // The halfDet value is cos(3*theta) mentioned in the PDF. The acos(z)
    // function requires |z| <= 1, but will fail silently and return NaN
    // if the input is larger than 1 in magnitude. To avoid this problem
    // due to rounding errors, the halfDet/ value is clamped to [-1,1].
    Real halfDet = det * (Real)0.5;
    halfDet = std::min(std::max(halfDet, (Real)-1), (Real)1);

    // The eigenvalues of B are ordered as beta0 <= beta1 <= beta2. The
    // number of digits in twoThirdsPi is chosen so that, whether float or
    // double, the floating-point number is the closest to theoretical 2*pi/3.
    Real angle = std::acos(halfDet) / (Real)3;
```

```
        Real const twoThirdsPi = (Real)2.09439510239319549;
        Real beta2 = std::cos(angle) * (Real)2;
        Real beta0 = std::cos(angle + twoThirdsPi) * (Real)2;
        Real beta1 = -(beta0 + beta2);

        // The eigenvalues of A are ordered as alpha0 <= alpha1 <= alpha2.
        eval[0] = q + p * beta0;
        eval[1] = q + p * beta1;
        eval[2] = q + p * beta2;

        // Compute the eigenvectors so that the set {evec[0], evec[1], evec[2]}
        // is right handed and orthonormal.
        if (halfDet >= (Real)0)
        {
            ComputeEigenvector0(a00, a01, a02, a11, a12, a22, eval[2], evec[2]);
            ComputeEigenvector1(a00, a01, a02, a11, a12, a22, evec[2], eval[1], evec[1]);
            evec[0] = Cross(evec[1], evec[2]);
        }
        else
        {
            ComputeEigenvector0(a00, a01, a02, a11, a12, a22, eval[0], evec[0]);
            ComputeEigenvector1(a00, a01, a02, a11, a12, a22, evec[0], eval[1], evec[1]);
            evec[2] = Cross(evec[0], evec[1]);
        }
    }
    else
    {
        // The matrix is diagonal.
        eval[0] = a00;
        eval[1] = a11;
        eval[2] = a22;
        evec[0] = { (Real)1, (Real)0, (Real)0 };
        evec[1] = { (Real)0, (Real)1, (Real)0 };
        evec[2] = { (Real)0, (Real)0, (Real)1 };
    }

    // The preconditioning scaled the matrix A, which scales the eigenvalues.
    // Revert the scaling.
    eval[0] *= maxAbsElement;
    eval[1] *= maxAbsElement;
    eval[2] *= maxAbsElement;
}

template <typename Real>
std::array<Real, 3> NISymmetricEigensolver3x3<Real>::Multiply(
    Real s, std::array<Real, 3> const& U)
{
    std::array<Real, 3> product = { s * U[0], s * U[1], s * U[2] };
    return product;
}

template <typename Real>
std::array<Real, 3> NISymmetricEigensolver3x3<Real>::Subtract(
    std::array<Real, 3> const& U, std::array<Real, 3> const& V)
{
    std::array<float, 3> difference = { U[0] - V[0], U[1] - V[1], U[2] - V[2] };
    return difference;
}

template <typename Real>
std::array<Real, 3> NISymmetricEigensolver3x3<Real>::Divide(
    std::array<Real, 3> const& U, Real s)
{
    Real invS = (Real)1 / s;
    std::array<Real, 3> division = { U[0] * invS, U[1] * invS, U[2] * invS };
    return division;
}

template <typename Real>
Real NISymmetricEigensolver3x3<Real>::Dot(std::array<Real, 3> const& U,
    std::array<Real, 3> const& V)
{
```

```
    Real dot = U[0] * V[0] + U[1] * V[1] + U[2] * V[2];
    return dot;
}

template <typename Real>
std::array<Real, 3> NISymmetricEigensolver3x3<Real>::Cross(std::array<Real, 3> const& U,
    std::array<Real, 3> const& V)
{
    std::array<Real, 3> cross =
    {
        U[1] * V[2] - U[2] * V[1],
        U[2] * V[0] - U[0] * V[2],
        U[0] * V[1] - U[1] * V[0]
    };
    return cross;
}

template <typename Real>
void NISymmetricEigensolver3x3<Real>::ComputeOrthogonalComplement(
    std::array<Real, 3> const& W, std::array<Real, 3>& U, std::array<Real, 3>& V) const
{
    // Robustly compute a right-handed orthonormal set { U, V, W }.  The
    // vector W is guaranteed to be unit-length, in which case there is no
    // need to worry about a division by zero when computing invLength.
    Real invLength;
    if (fabs(W[0]) > fabs(W[1]))
    {
        // The component of maximum absolute value is either W[0] or W[2].
        invLength = (Real)1 / sqrt(W[0] * W[0] + W[2] * W[2]);
        U = { -W[2] * invLength, (Real)0, +W[0] * invLength };
    }
    else
    {
        // The component of maximum absolute value is either W[1] or W[2].
        invLength = (Real)1 / sqrt(W[1] * W[1] + W[2] * W[2]);
        U = { (Real)0, +W[2] * invLength, -W[1] * invLength };
    }
    V = Cross(W, U);
}

template <typename Real>
void NISymmetricEigensolver3x3<Real>::ComputeEigenvector0(Real a00, Real a01,
    Real a02, Real a11, Real a12, Real a22, Real& eval0, std::array<Real, 3>& evec0) const
{
    // Compute a unit-length eigenvector for eigenvalue[i0].  The matrix is
    // rank 2, so two of the rows are linearly independent.  For a robust
    // computation of the eigenvector, select the two rows whose cross product
    // has largest length of all pairs of rows.
    std::array<Real, 3> row0 = { a00 - eval0, a01, a02 };
    std::array<Real, 3> row1 = { a01, a11 - eval0, a12 };
    std::array<Real, 3> row2 = { a02, a12, a22 - eval0 };
    std::array<Real, 3>  r0xr1 = Cross(row0, row1);
    std::array<Real, 3>  r0xr2 = Cross(row0, row2);
    std::array<Real, 3>  r1xr2 = Cross(row1, row2);
    Real d0 = Dot(r0xr1, r0xr1);
    Real d1 = Dot(r0xr2, r0xr2);
    Real d2 = Dot(r1xr2, r1xr2);

    Real dmax = d0;
    int imax = 0;
    if (d1 > dmax)
    {
        dmax = d1;
        imax = 1;
    }
    if (d2 > dmax)
    {
        imax = 2;
    }

    if (imax == 0)
    {
```

```
            evec0 = Divide(r0xr1, sqrt(d0));
        }
        else if (imax == 1)
        {
            evec0 = Divide(r0xr2, sqrt(d1));
        }
        else
        {
            evec0 = Divide(r1xr2, sqrt(d2));
        }
    }

    template <typename Real>
    void NISymmetricEigensolver3x3<Real>::ComputeEigenvector1(Real a00, Real a01,
        Real a02, Real a11, Real a12, Real a22, std::array<Real, 3> const& evec0,
        Real& eval1, std::array<Real, 3>& evec1) const
    {
        // Robustly compute a right-handed orthonormal set { U, V, evec0 }.
        std::array<Real, 3> U, V;
        ComputeOrthogonalComplement(evec0, U, V);

        // Let e be eval1 and let E be a corresponding eigenvector which is a
        // solution to the linear system (A - e*I)*E = 0.  The matrix (A - e*I)
        // is 3x3, not invertible (so infinitely many solutions), and has rank 2
        // when eval1 and eval are different.  It has rank 1 when eval1 and eval2
        // are equal.  Numerically, it is difficult to compute robustly the rank
        // of a matrix.  Instead, the 3x3 linear system is reduced to a 2x2 system
        // as follows.  Define the 3x2 matrix J = [U V] whose columns are the U
        // and V computed previously.  Define the 2x1 vector X = J*E.  The 2x2
        // system is 0 = M * X = (J^T * (A - e*I) * J) * X where J^T is the
        // transpose of J and M = J^T * (A - e*I) * J is a 2x2 matrix.  The system
        // may be written as
        //     +-                        -++-  -+       +-  -+
        //     | U^T*A*U - e  U^T*A*V      || x0 | = e * | x0 |
        //     | V^T*A*U      V^T*A*V - e  || x1 |       | x1 |
        //     +-                        -++   -+       +-  -+
        // where X has row entries x0 and x1.

        std::array<Real, 3> AU =
        {
            a00 * U[0] + a01 * U[1] + a02 * U[2],
            a01 * U[0] + a11 * U[1] + a12 * U[2],
            a02 * U[0] + a12 * U[1] + a22 * U[2]
        };

        std::array<Real, 3> AV =
        {
            a00 * V[0] + a01 * V[1] + a02 * V[2],
            a01 * V[0] + a11 * V[1] + a12 * V[2],
            a02 * V[0] + a12 * V[1] + a22 * V[2]
        };

        Real m00 = U[0] * AU[0] + U[1] * AU[1] + U[2] * AU[2] - eval1;
        Real m01 = U[0] * AV[0] + U[1] * AV[1] + U[2] * AV[2];
        Real m11 = V[0] * AV[0] + V[1] * AV[1] + V[2] * AV[2] - eval1;

        // For robustness, choose the largest-length row of M to compute the
        // eigenvector.  The 2-tuple of coefficients of U and V in the
        // assignments to eigenvector[1] lies on a circle, and U and V are
        // unit length and perpendicular, so eigenvector[1] is unit length
        // (within numerical tolerance).
        Real absM00 = fabs(m00);
        Real absM01 = fabs(m01);
        Real absM11 = fabs(m11);
        Real maxAbsComp;
        if (absM00 >= absM11)
        {
            maxAbsComp = std::max(absM00, absM01);
            if (maxAbsComp > (Real)0)
            {
                if (absM00 >= absM01)
                {
```

```
                m01 /= m00;
                m00 = (Real)1 / sqrt((Real)1 + m01 * m01);
                m01 *= m00;
            }
            else
            {
                m00 /= m01;
                m01 = (Real)1 / sqrt((Real)1 + m00 * m00);
                m00 *= m01;
            }
            evec1 = Subtract(Multiply(m01, U), Multiply(m00, V));
        }
        else
        {
            evec1 = U;
        }
    }
    else
    {
        maxAbsComp = std::max(absM11, absM01);
        if (maxAbsComp > (Real)0)
        {
            if (absM11 >= absM01)
            {
                m01 /= m11;
                m11 = (Real)1 / sqrt((Real)1 + m01 * m01);
                m01 *= m11;
            }
            else
            {
                m11 /= m01;
                m01 = (Real)1 / sqrt((Real)1 + m11 * m11);
                m11 *= m01;
            }
            evec1 = Subtract(Multiply(m11, U), Multiply(m01, V));
        }
        else
        {
            evec1 = U;
        }
    }
}
```