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: August 24, 2021

Contents

1 Introduction

Let A be a 3×3 symmetric matrix of real numbers. From linear algebra, 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 $R = [\boldsymbol{U}_0\, \boldsymbol{U}_1\, \boldsymbol{U}_2]$ be an orthogonal matrix whose columns are linearly independent eigenvectors, ordered consistently with the diagonal entries of D. That is, $A\boldsymbol{U}_i = \lambda_i \boldsymbol{U}_i$. 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 root, solve the equation $(A - \lambda I)\boldsymbol{U} = \boldsymbol{0}$ for nonzero vectors $\boldsymbol{U}$. Although theoretically correct, computing roots of the cubic polynomial using the closed-form equations and floating-point arithmetic can lead to inaccurate results.

2 An Iterative Algorithm

The matrix notation in this document uses 0-based indexing; that is, if M is a 3×3 matrix, the element in row r and column c is m_{rc} where $0 \le r \le 2$ and $0 \le c \le 2$. The *superdiagonal elements* are m_{01} and m_{12} and *subdiagonal elements* are m_{10} and m_{21}. For symmetric M, the two lists are the same. The numerical algorithms for computing the eigenvalues and eigenvectors modify elements m_{rc} for $c \ge r$. The elements m_{rc} for $c < r$ are not stored because they are known by symmetry.

The classical numerical algorithm for computing the eigenvalues and eigenvectors of A initially uses a Householder reflection matrix H to compute $B = H^{\mathsf{T}}AH$ so that $b_{02} = 0$. This makes B a symmetric 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 elements to zero. This is an iterative process for which a termination condition is required. If ℓ rotations are applied, the reductions are

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

where D' is a symmetric tridiagonal matrix. The matrix D is diagonal and the matrix E is symmetric tridiagonal with diagonal elements 0 and superdiagonal elements that are sufficiently smaller than the diagonal elements of D adjacent to them. The diagonal elements of D are reasonable approximations to the eigenvalues of A. The orthogonal matrix $R' = HG_0 \cdots G_{\ell-1}$ has columns that are reasonable approximations to the eigenvectors of A.

The source code that implements this algorithm (for any size symmetric matrix) is in SymmetricEigensolver.h and is an implementation of Algorithm 8.3.3 (Symmetric QR Algorithm) described in [1]. Algorithm 8.3.1 (Householder Tridiagonalization) reduces a symmetric $n \times n$ matrix A to a tridiagonal matrix T using $n - 2$ Householder reflections. Algorithm 8.3.2 (Implicit Symmetric QR Step with Wilkinson Shift) reduces T to the (nearly) diagonal matrix D' of equation (1) using a finite number of Givens rotations, the number depending on a termination condition. The numerical errors are the elements of $E = R^{\mathsf{T}}AR - D$. Algorithm 8.3.3 mentions that one expects $|E| \doteq \mu|A|$, where $|A|$ denotes the Frobenius norm of A 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`.

If C is the current tridiagonal matrix obtained from the Givens rotations, the book uses the condition

$|c_{i,i+1}| \le \varepsilon|c_{i,i} + c_{i+1,i+1}|$ to determine when the reduction decouples to smaller problems. The value of $\varepsilon > 0$ is chosen to be small. That is, when a superdiagonal term is effectively zero, the iterations may be applied separately to two tridiagonal submatrices. The first submatrix is the upper-left square block with rows and column indices from 0 to i and the second submatrix is the lower-right square block with row and column indices from $i+1$ to $n-1$. The two submatrices are processed independently. The Geometric Tools source code is implemented instead to provide a termination condition when floating-point arithmetic is used,

Listing 1. The condition used to decompose the current tridiagonal matrix into two tridiagonal submatrices. The matrix element $c_{i,j}$ is written in code as $c(i,j)$.

```
bool Converged(bool aggressive, Real diagonal0, Real diagonal1, Real superdiagonal)
{
    if (aggressive)
    {
        Test whether the superdiagonal term c_{i,i+1} is zero.
        return superdiagonal == 0;
    }
    else
    {
        // Test whether the superdiagonal term c_{i,i+1} is effectively zero
        // compared to its diagonal neighbors.
        Real sum = |diagonal0| + |diagonal1|;
        return sum + |superdiagonal| == sum;
    }
}
```

Using the nonaggressive condition, the superdiagonal elements are effectively zero relative to their diagonal neighbors. 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

$$H^{\mathsf{T}}AH = \begin{bmatrix} c & s & 0 \\ s & -c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{01} & a_{11} & a_{12} \\ a_{02} & a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} c & s & 0 \\ s & -c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c(ca_{00}+sa_{01})+s(ca_{01}+sa_{11}) & s(ca_{00}+sa_{01})-c(ca_{01}+sa_{11}) & ca_{02}+sa_{12} \\ s(ca_{00}+sa_{01})-c(ca_{01}+sa_{11}) & s(sa_{00}-ca_{01})-c(sa_{01}-ca_{11}) & sa_{02}-ca_{12} \\ ca_{02}+sa_{12} & sa_{02}-ca_{12} & a_{22} \end{bmatrix} \tag{2}$$

$$= \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{01} & b_{11} & b_{12} \\ b_{02} & b_{12} & b_{22} \end{bmatrix}$$

Require $0 = b_{02} = ca_{02} + sa_{12} = (c, s) \cdot (a_{02}, a_{12})$, which implies (c, s) is perpendicular to (a_{02}, a_{12}). Equivalently, (c, s) is parallel to $(a_{12}, -a_{02})$. The vector (c, s) is obtained by normalizing $(a_{12}, -a_{02})$, say, $(c, s) = \sigma(a_{12}, -a_{02})/\sqrt{a_{02}^2 + a_{12}^2}$ for sign $\sigma \in \{-1, +1\}$. Either choice of sign is allowed, but the Geometric Tools source code chooses σ so that $\cos\theta \le 0$, which allows some source code to be shared with the Givens reductions.

Generally, given an expression $(\cos\phi, \sin\phi) \cdot (-v, u) = 0$ where $(-v, u) \neq (0, 0)$, the vector $(\cos\phi, \sin\phi)$ is parallel to (u, v) and is obtained by normalizing (u, v), say $(\cos\phi, \sin\phi) = \sigma(u, v)/\sqrt{u^2 + v^2}$ for $\sigma \in \{-1, +1\}$. The source code uses $\sigma = -\operatorname{Sign}(u)$,

$$(\cos\phi, \sin\phi) = -\operatorname{Sign}(u)\,\frac{(u, v)}{\sqrt{u^2 + v^2}} = \frac{(-|u|, -\operatorname{Sign}(u)v)}{\sqrt{u^2 + v^2}} \tag{3}$$

Pseudocode for the computation is shown in Listing 2.

Listing 2. Given a pair (u, v), the pseudocode solves $(\cos\phi, \sin\phi) \cdot (-v, u) = 0$ for $(\cos\phi, \sin\phi)$.

```
void GetCosSin(Real u, Real v, Real& cosPhi, Real& sinPhi)
{
    Real length = sqrt(u * u + v * v);
    if (length > 0)
    {
        cosPhi = u / length;
        sinPhi = v / length;
        if (cosPhi > 0)
        {
            cosPhi = -cosPhi;
            sinPhi = -sinPhi;
        }
    }
    else
    {
        cosPhi = -1;
        sinPhi = 0;
    }
}
```

The pair $(c, s) = (\cos\phi, \sin\phi)$ is constructed using GetCosSin(a12, -a02, c, s) or GetCosSin(-a12, a02, c, s).

Reflection matrices are used rather than Givens rotations to force the superdiagonal terms to zero. The next two subsections illustrate this.

3.1 Case $|b_{12}| \le |b_{01}|$

Let $|b_{12}| \le |b_{01}|$. Choose a sequence of reflection matrices to drive b_{12} to zero; the first one is

$$G_1 = \begin{bmatrix} c_1 & 0 & -s_1 \\ s_1 & 0 & c_1 \\ 0 & 1 & 0 \end{bmatrix} \tag{4}$$

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

$$P_1 = G_1^{\mathsf{T}} B G_1 = \begin{bmatrix} c_1(c_1 b_{00} + s_1 b_{01}) + s_1(c_1 b_{01} + s_1 b_{11}) & s_1 b_{12} & s_1 c_1(b_{11} - b_{00}) + (c_1^2 - s_1^2) b_{01} \\ s_1 b_{12} & b_{22} & c_1 b_{12} \\ s_1 c_1(b_{11} - b_{00}) + (c_1^2 - s_1^2) b_{01} & c_1 b_{12} & s_1(s_1 b_{00} - c_1 b_{01}) - c_1(s_1 b_{01} - c_1 b_{11}) \end{bmatrix} \tag{5}$$

P_1 must be tridiagonal, which requires

$$0 = s_1 c_1(b_{11} - b_{00}) + (c_1^2 - s_1^2) b_{01} = (\cos(2\theta_1), \sin(2\theta_1)) \cdot (b_{01}, (b_{11} - b_{00})/2) \tag{6}$$

Use equation (3) with $u = (b_{11} - b_{00})/2$ and $v = -b_{01}$ to compute $(\cos(2\theta_1), \sin(2\theta_1))$ with $\cos(2\theta_1) \le 0$. The numbers $c_1 = \cos\theta_1$ and $s_1 = \sin\theta$ are extracted from $\cos(2\theta_1)$ and $\sin(2\theta_1)$ as

$$\sin\theta_1 = \sqrt{\frac{1 - \cos(2\theta_1)}{2}}, \quad \cos\theta_1 = \frac{\sin(2\theta_1)}{2\sin\theta_1} = \sqrt{\frac{1 + \cos(2\theta_1)}{2}} \tag{7}$$

Notice that

$$|\cos\theta_1| = \sqrt{\frac{1 + \cos(2\theta_1)}{2}} \le \sqrt{\frac{1 - \cos(2\theta_1)}{2}} = \sin\theta_1 = |\sin\theta_1| \tag{8}$$

where the inequality occurs because $\cos(2\theta_1) \le 0$. Equality of $|\cos\theta_1|$ and $|\sin\theta_1|$ occurs only when $\theta_1 = 0$ which in turn occurs only when $b_{01} = 0$. The precondition $|b_{12}| \le |b_{01}|$ implies that $b_{12} = 0$. The matrix B is diagonal, so Givens reduction is not needed. In the following discusssion assume that $\theta_1 > 0$, which implies $|\cos\theta_1| < 1/\sqrt{2} < |\sin\theta_1|$, $b_{01} \ne 0$ and $b_{12} \ne 0$.

Let P_1 have elements $p_{ij}^{(1)}$. The superdiagonal elements are $p_{01}^{(1)} = s_1 b_{12}$ and $p_{12}^{(1)} = c_1 b_{12}$. The condition $|c_1| < 1/\sqrt{2}$ implies $\left|p_{12}^{(1)}\right| < |b_{12}|/\sqrt{2} < |b_{12}|$. The precondition for computing P_1 from B was $|b_{12}| \le |b_{01}|$. The postcondition is

$$\left|p_{12}^{(1)}\right| = |c_1 b_{12}| = |c_1||b_{12}| < |s_1||b_{12}| = |s_1 b_{12}| = \left|p_{01}^{(1)}\right| \tag{9}$$

which is the precondition for multiplying P_1 by another Givens reflection G_2 to obtain $P_2 = G_2^{\mathsf{T}} P_1 G_2 = (G_1 G_2)^{\mathsf{T}} B (G_1 G_2)$.

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 $(1,2)$-element of the output matrix P_{i+1} is strictly smaller than the $(1,2)$-element of the input matrix P_i, so repeated iterations will drive the $(1,2)$-element to zero. If $c_i = \cos\theta_i$ and $s_i = \sin\theta_i$, then $|c_i| < 1/\sqrt{2} < |s_i|$ and

$$\left|p_{12}^{(i+1)}\right| = \left|c_{i+1} p_{12}^{(i)}\right| < 2^{-1/2} \left|p_{12}^{(i)}\right| \tag{10}$$

which implies

$$\left|p_{12}^{(i)}\right| < 2^{-i/2} |b_{12}|, \quad i \ge 1 \tag{11}$$

In the limit as $i \to \infty$, the $(1,2)$-element is forced to zero. In fact the bounds here determine 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.

Let the smallest subnormal number be $2^{-\alpha}$. The value α is 149 for float and 1074 for double. Let $i_{\max}$ be the largest positive integer for which $2^{-i_{\max}/2}|b_{12}| < 2^{-\alpha-1}$. Using the default rounding mode (round-to-nearest-ties-to-even), the left-hand real-valued number rounds to zero. Represent $|b_{12}| = x \cdot 2^e$ for some $x \in [1/2, 1)$

and for some integer power e; then $x < 2^{i_{\max}/2-e-\alpha-1}$. Knowing $x < 1$, we can choose the exponent on the right-hand side to be 0, in which case

$$i_{\max} = 2(e + \alpha + 1) \tag{12}$$

Once the decoupling condition of Listing 1 is satisfied, the final P-matrix is of the form

$$P = \begin{bmatrix} p_{00} & p_{01} & 0 \\ p_{01} & p_{11} & 0 \\ 0 & 0 & p_{22} \end{bmatrix} \tag{13}$$

Another Householder reflection is applied to diagonalize the upper-left 2×2 block,

$$H' = \begin{bmatrix} c & s & 0 \\ s & -c & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{14}$$

for some angle θ with $c = \cos\theta$ and $s = \sin\theta$. Applying the reflection,

$$P' = H'PH' = \begin{bmatrix} c(cp_{00} + sp_{01}) + s(cp_{01} + sp_{11}) & sc(p_{00} - p_{11}) + (c^2 - s^2)(-p_{01}) & 0 \\ sc(p_{00} - p_{11}) + (c^2 - s^2)(-p_{01}) & s(sp_{00} - cp_{01}) - c(sp_{01} - cp_{11}) & 0 \\ 0 & 0 & p_{22} \end{bmatrix} \tag{15}$$

P' must be tridiagonal, which requires

$$0 = sc(p_{00} - p_{11}) + (c^2 - s^2)(-p_{01}) = (\cos(2\theta), \sin(2\theta)) \cdot (-p_{01}, (p_{00} - p_{11})/2) \tag{16}$$

Use equation (3) with $u = (p_{00} - p_{11})/2$ and $v = p_{01}$ to compute $(\cos(2\theta_1), \sin(2\theta_1))$. Extract $\sin\theta = \sqrt{(1 - \cos(2\theta))/2}$ and $\cos\theta = \sin(2\theta)/(2\sin\theta)$. The estimates for the eigenvalues are

$$\begin{aligned} \lambda_0 &= p'_{00} = c(cp_{00} + sp_{01}) + s(cp_{01} + sp_{11}) \\ \lambda_1 &= p'_{11} = s(sp_{00} - cp_{01}) - c(sp_{01} - cp_{11}) \\ \lambda_2 &= p'_{22} = p_{22} \end{aligned} \tag{17}$$

3.2 Case $|b_{01}| \le |b_{12}|$

Let $|b_{01}| \le |b_{12}|$. Choose a sequence of reflection matrices to drive b_{01} to zero; the first one is

$$G_1 = \begin{bmatrix} 0 & 1 & 0 \\ c_1 & 0 & -s_1 \\ s_1 & 0 & c_1 \end{bmatrix} \tag{18}$$

Define the product

$$P_1 = G_1^{\mathsf{T}} B G_1 = \begin{bmatrix} c_1(c_1 b_{11} + s_1 b_{12}) + s_1(c_1 b_{12} + s_1 b_{22}) & c_1 b_{01} & s_1 c_1(b_{22} - b_{11}) + (c_1^2 - s_1^2) b_{12} \\ c_1 b_{01} & b_{00} & -s_1 b_{01} \\ s_1 c_1(b_{22} - b_{11}) + (c_1^2 - s_1^2) b_{12} & -s_1 b_{01} & s_1(s_1 b_{11} - c_1 b_{12}) - c_1(s_1 b_{12} - c_1 b_{22}) \end{bmatrix} \tag{19}$$

P_1 must be tridiagonal, which requires

$$0 = s_1 c_1(b_{22} - b_{11}) + (c_1^2 - s_1^2) b_{12} = (\cos(2\theta_1), \sin(2\theta_1)) \cdot (b_{12}, (b_{22} - b_{11})/2) \tag{20}$$

Use equation (3) with $u = (b_{22} - b_{11})/2$ and $v = -b_{12}$ to compute $(\cos(2\theta_1), \sin(2\theta_1))$ with $\cos(2\theta_1) \leq 0$. The numbers $c_1 = \cos\theta_1$ and $s_1 = \sin\theta$ are extracted from $\cos(2\theta_1)$ and $\sin(2\theta_1)$ as

$$\sin\theta_1 = \sqrt{\frac{1 - \cos(2\theta_1)}{2}}, \quad \cos\theta_1 = \frac{\sin(2\theta_1)}{2\sin\theta_1} = \sqrt{\frac{1 + \cos(2\theta_1)}{2}} \tag{21}$$

Notice that

$$|\cos\theta_1| = \sqrt{\frac{1 + \cos(2\theta_1)}{2}} \leq \sqrt{\frac{1 - \cos(2\theta_1)}{2}} = \sin\theta_1 = |\sin\theta_1| \tag{22}$$

where the inequality occurs because $\cos(2\theta_1) \leq 0$. Equality of $|\cos\theta_1|$ and $|\sin\theta_1|$ occurs only when $\theta_1 = 0$ which in turn occurs only when $b_{12} = 0$. The precondition $|b_{01}| \leq |b_{12}|$ implies that $b_{01} = 0$. The matrix B is diagonal, so Givens reduction is not needed. In the following discusssion assume that $\theta_1 > 0$, which implies $|\cos\theta_1| < 1/\sqrt{2} < |\sin\theta_1|$, $b_{01} \neq 0$ and $b_{12} \neq 0$.

Let P_1 have elements $p_{ij}^{(1)}$. The superdiagonal elements are $p_{01}^{(1)} = c_1 b_{01}$ and $p_{12}^{(1)} = -s_1 b_{01}$. The condition $|c_1| < 1/\sqrt{2}$ implies $\left|p_{01}^{(1)}\right| < |b_{01}|/\sqrt{2} < |b_{01}|$. The precondition for computing P_1 from B was $|b_{01}| \leq |b_{12}|$. The postcondition is

$$\left|p_{01}^{(1)}\right| = |c_1 b_{01}| = |c_1||b_{01}| < |s_1||b_{01}| = |s_1 b_{01}| = \left|p_{01}^{(1)}\right| \tag{23}$$

which is the precondition for multiplying P_1 by another Givens reflection G_2 to obtain $P_2 = G_2^{\mathsf{T}} P_1 G_2 = (G_1 G_2)^{\mathsf{T}} B (G_1 G_2)$.

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 $(0,1)$-element of the output matrix P_{i+1} is strictly smaller than the $(0,1)$-element of the input matrix P_i, so repeated iterations will drive the $(0,1)$-element to zero. If $c_i = \cos\theta_i$ and $s_i = \sin\theta_i$, then $|c_i| < 1/\sqrt{2} < |s_i|$ and

$$\left|p_{01}^{(i+1)}\right| = \left|c_{i+1} p_{01}^{(i)}\right| < 2^{-1/2} \left|p_{01}^{(i)}\right| \tag{24}$$

which implies

$$\left|p_{01}^{(i)}\right| < 2^{-i/2} |b_{01}|, \quad i \geq 1 \tag{25}$$

In the limit as $i \to \infty$, the $(0,1)$-element is forced to zero. In fact the bounds here determine the final i so that $2^{-i/2}|b_{01}|$ is a number whose nearest floating-point representation is zero. The number of iterations of the algorithm is limited by this i. The construction for the maximum iterations shown in equation (12) applies as well in this case.

Once the decoupling condition of Listing 1 is satisfied, the final P-matrix is of the form

$$P = \begin{bmatrix} p_{00} & 0 & 0 \\ 0 & p_{11} & p_{12} \\ 0 & p_{12} & p_{22} \end{bmatrix} \tag{26}$$

Another Householder reflection is applied to diagonalize the lower-right 2×2 block,

$$H' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & s & -c \end{bmatrix} \tag{27}$$

for some angle θ with $c = \cos\theta$ and $s = \sin\theta$. Applying the reflection,

$$P' = H'PH' = \begin{bmatrix} p_{00} & 0 & 0 \\ 0 & c(cp_{11} + sp_{12}) + s(cp_{12} + sp_{22}) & sc(p_{11} - p_{22}) + (c^2 - s^2)(-p_{12}) \\ 0 & sc(p_{11} - p_{22}) + (c^2 - s^2)(-p_{12}) & s(sp_{11} - cp_{12}) - c(sp_{12} - cp_{22}) \end{bmatrix} \tag{28}$$

P' must be tridiagonal, which requires

$$0 = sc(p_{11} - p_{22}) + (c^2 - s^2)(-p_{12}) = (\cos(2\theta), \sin(2\theta)) \cdot (-p_{12}, (p_{11} - p_{22})/2) \tag{29}$$

Use equation (3) with $u = (p_{11} - p_{22})/2$ and $v = p_{12}$ to compute $(\cos(2\theta_1), \sin(2\theta_1))$. Extract $\sin\theta = \sqrt{(1 - \cos(2\theta))/2}$ and $\cos\theta = \sin(2\theta)/(2\sin\theta)$. The estimates for the eigenvalues are

$$\begin{aligned} \lambda_0 &= p_{00} \\ \lambda_1 &= c(cp_{11} + sp_{12}) + s(cp_{12} + sp_{22}) \\ \lambda_2 &= s(sp_{11} - cp_{12}) - c(sp_{12} - cp_{22}) \end{aligned} \tag{30}$$

3.3 Estimating the Eigenvectors

Let H_0 and H_1 be the first and last Householder reflections applied to A. Let G_i be the ℓ Givens reflections applied to A, where $0 \le i < \ell$. The reflections are accumulated as a product of matrices $Q = H_0 G_0 \cdots G_{\ell-1} H_1$. The columns of Q are the eigenvector estimates.

4 Implementation of the Iterative Algorithm

Source code is provided in this section for the iterative algorithm, as shown in Listing 3. The actual C++ source code that implements the algorithm is found in the file SymmetricEigensolver3x3.h. The code was written not to use any GTE objects, removing a dependency on GTE for this algorithm. The source code has additional logic for sorting the eigenvalues and eigenvectors. The sorting algorithm is relatively simple and is not included in the pseudocode.

Listing 3. Source code for the iterative algorithm. The eigenvalue/eigenvector sorting used in the actual code is not included. Pseudocode for the Converged function is in Listing1. Source code for the GetCosSin function is in Listing 2.

```
// The function size_t return value is the number of Givens reflections in the eigensolver.
int SymmetricEigensolver3x3(Real a00, Real a01, Real a02, Real a11, Real a12, Real a22, bool aggressive,
    std::array<Real,3>& eigenvalues, std::array<std::array<Real,3>,3>& eigenvectors)
{
    // Compute the Householder reflection H_0 and B = H_0 A H_0, where b_02 = 0. H_0 = {{c,s,0},{s,-c,0},{0,0,1}}
    // with each inner triple a row of H_0.
    bool isRotation = false;
    Real c, s;
    GetCosSin(a12, -a02, c, s);
    Real term0 = c * a00 + s * a01;
    Real term1 = c * a01 + s * a11;
    Real term2 = s * a00 - c * a01;
    Real term3 = s * a01 - c * a11;
    // Real b02 = c * a02 + s * a12; // theoretically zero
    Real b00 = c * term0 + s * term1;
    Real b01 = s * term0 - c * term1;
    Real b11 = s * term2 - c * term3;
    Real b12 = s * a02 - c * a12;
    Real b22 = a22;

    // Maintain Q as the product of the reflections. Initially, Q = H_0. Updates by Givens reflections G are Q <- QG.
    // The columns of the final Q are the estimates for the eigenvectors.
    std::array<std::array<Real, 3>, 3> Q =
    {{
        { c,  s, 0 },
        { s, -c, 0 },
        { 0,  0, 1 }
    }};

    The smallest floating-point number of type Real is 2^{-alpha}. The code here uses C++ functions.
    int alpha = std::numeric_limits<T>::digits - std::numeric_limits<T>::min_exponent;
    int i = 0, imax = 0, power = 0;
    T c2, s2;

    if (std::fabs(b12) <= std::fabs(b01))
    {
        The maximum number of iterations i_max can be computed from Equation (12).
        std::frexp(b12, &power);)
        imax = 2 * (power + alpha + 1);
        for (i = 0; i < imax; ++i)
        {
            // Compute the Givens reflection G = {{c,0,-s},{s,0,c},{0,1,0}} where each inner triple is a row of G.
            GetCosSin(half * (b00 - b11), b01, c2, s2);
            s = std::sqrt(half * (one - c2));
            c = s2 / (2 * s);

            // Update Q <- QG.
            for (int r = 0; r < 3; ++r)
            {
                term0 = c * Q[r][0] + s * Q[r][1];
                term1 = Q[r][2];
                term2 = c * Q[r][1] - s * Q[r][0];
                Q[r][0] = term0;
                Q[r][1] = term1;
                Q[r][2] = term2;
            }
            isRotation = !isRotation;

            // Update B <- Q^T BQ, ensuring that b_02 is zero and |b_12| has strictly decreased.
            term0 = c * b00 + s * b01;
            term1 = c * b01 + s * b11;
            term2 = s * b00 - c * b01;
            term3 = s * b01 - c * b11;
            // b02 = s * c * (b11 - b00) + (c * c - s * s) * b01; // theoretically zero
            b00 = c * term0 + s * term1;
            b01 = s * b12;
            b11 = b22;
            b12 = c * b12;
            b22 = s * term2 - c * term3;
```

```
            if (Converged(aggressive, b00, b11, b01))
            {
                // Compute the Householder reflection H_1 = {{c, s, 0}, {s, -c, 0}, {0, 0, 1}} where each inner
                // triple is a row of H_1.
                GetCosSin(half * (b00 - b11), b01, c2, s2);
                s = std::sqrt(half * (one - c2));
                c = half * s2 / s;

                // Update Q <- QH_1.
                for (int r = 0; r < 3; ++r)
                {
                    term0 = c * Q[r][0] + s * Q[r][1];
                    term1 = s * Q[r][0] - c * Q[r][1];
                    Q[r][0] = term0;
                    Q[r][1] = term1;
                }
                isRotation = !isRotation;

                // Compute the diagonal estimate D = Q^T BQ.
                term0 = c * b00 + s * b01;
                term1 = c * b01 + s * b11;
                term2 = s * b00 - c * b01;
                term3 = s * b01 - c * b11;
                b00 = c * term0 + s * term1;
                b11 = s * term2 - c * term3;
                break;
            }
        }
    }
    else
    {
        The maximum number of iterations i_max can be computed from Equation (12).
        std::frexp(b01, &power);)
        imax = 2 * (power + alpha + 1);
        for (i = 0; i < imax; ++i)
        {
            // Compute the Givens reflection G = {{0, 1, 0}, {c, 0, -s}, {s, 0, c}} where each inner triple is a row of G.
            GetCosSin(half * (b11 - b22), b12, c2, s2);
            s = std::sqrt(half * (one - c2));
            c = half * s2 / s;

            // Update Q <- QG.
            for (int r = 0; r < 3; ++r)
            {
                term0 = c * Q[r][1] + s * Q[r][2];
                term1 = Q[r][0];
                term2 = c * Q[r][2] - s * Q[r][1];
                Q[r][0] = term0;
                Q[r][1] = term1;
                Q[r][2] = term2;
            }
            isRotation = !isRotation;

            // Update B <- Q^T BQ, ensuring that b_02 is zero and |b_01| has strictly decreased.
            term0 = c * b11 + s * b12;
            term1 = c * b12 + s * b22;
            term2 = s * b11 - c * b12;
            term3 = s * b12 - c * b22;
            //b02 = s * c * (b22 - b11) + (c * c - s * s) * b12; // theoretically zero
            b22 = s * term2 - c * term3;
            b12 = -s * b01;
            b11 = b00;
            b01 = c * b01;
            b00 = c * term0 + s * term1;

            if (Converged(aggressive, b11, b22, b12))
            {
                // Compute the Householder reflection H_1 = {{1, 0, 0}, {0, c, s}, {0, s, -c}} where each inner
```

```
                // triple is a row of H1.
                GetCosSin(half * (b11 - b22), b12, c2, s2);
                s = std::sqrt(half * (one - c2));
                c = half * s2 / s;

                // Update Q ← QH1.
                for (int r = 0; r < 3; ++r)
                {
                    term0 = c * Q[r][1] + s * Q[r][2];
                    term1 = s * Q[r][1] - c * Q[r][2];
                    Q[r][1] = term0;
                    Q[r][2] = term1;
                }
                isRotation = !isRotation;

                // Compute the diagonal estimate D = Q^T B Q.
                term0 = c * b11 + s * b12;
                term1 = c * b12 + s * b22;
                term2 = s * b11 - c * b12;
                term3 = s * b12 - c * b22;
                b11 = c * term0 + s * term1;
                b22 = s * term2 - c * term3;
                break;
            }
        }
    }

    eigenvalues = { b00, b11, b22 };
    for (int row = 0; row < 3; ++row)
    {
        for (int col = 0; col < 3; ++col)
        {
            eigenvectors[row][col] = Q[col][row];
        }
    }

    return i;
}
```

---

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. 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,

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{01} & a_{11} & a_{12} \\ a_{02} & a_{12} & a_{22} \end{bmatrix} \tag{31}$$

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

$$f(\alpha) = \det(\alpha I - A) = \alpha^3 - c_2\alpha^2 + c_1\alpha - c_0 \tag{32}$$

where I is the 3×3 identity matrix and where

$$\begin{aligned} c_2 &= a_{00} + a_{11} + a_{22} = \mathrm{tr}(A) \\ c_1 &= (a_{00}a_{11} - a_{01}^2) + (a_{00}a_{22} - a_{02}^2) + (a_{11}a_{22} - a_{12}^2) = \left(\mathrm{tr}^2(A) - \mathrm{tr}(A^2)\right)/2 \\ c_0 &= a_{00}a_{11}a_{22} + 2a_{01}a_{02}a_{12} - a_{00}a_{12}^2 - a_{11}a_{02}^2 - a_{22}a_{01}^2 = \det(A) \end{aligned} \tag{33}$$

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 $\boldsymbol{v}$ is an eigenvector of A, then by definition $A\boldsymbol{v} = \alpha\boldsymbol{v}$ where α is an eigenvalue of A; moreover,

$$\alpha\boldsymbol{v} = A\boldsymbol{v} = pB\boldsymbol{v} + q\boldsymbol{v} \tag{34}$$

which implies $B\boldsymbol{v} = ((\alpha - q)/p)\boldsymbol{v}$. Thus, $\boldsymbol{v}$ 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.

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

$$g(\beta) = \det(\beta I - B) = \beta^3 - 3\beta - \det(B) \tag{35}$$

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

$$g(\theta) = 2\left(4\cos^3(\theta) - 3\cos(\theta)\right) - \det(B) \tag{36}$$

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

$$g(\theta) = 2\cos(3\theta) - \det(B) \tag{37}$$

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 $|\cos(3\theta)| \leq 1$.[1] This additionally implies that $|\det(B)| \leq 2$. The real-valued roots of $g(\theta)$ are

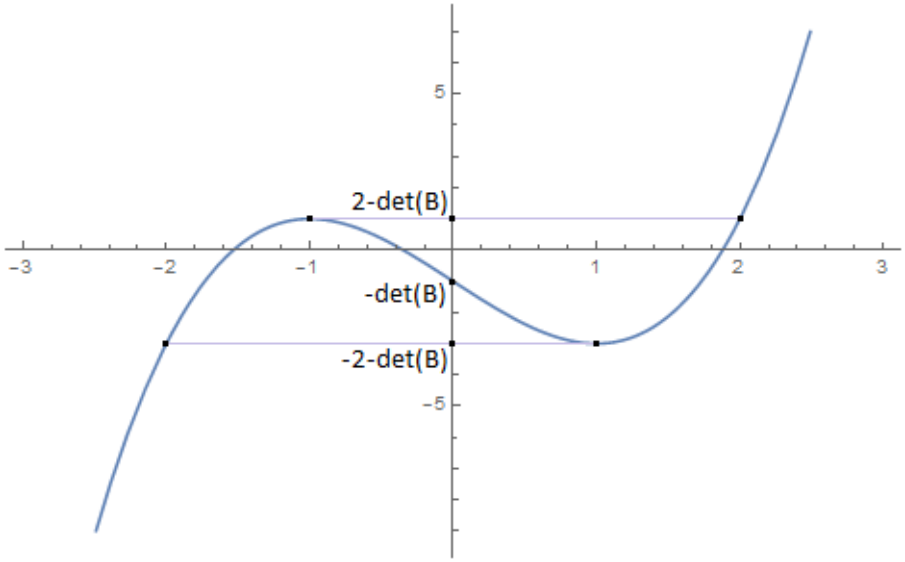
$$\beta = 2\cos\left(\theta + \frac{2\pi k}{3}\right), \quad k = 0, 1, 2 \tag{38}$$

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

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.

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 $\text{tr}(B) = 0$, 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 $(A - \alpha_0 I)\boldsymbol{v} = \boldsymbol{0}$. 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,

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{0} = (A - \alpha_0 I)\boldsymbol{v} = \begin{bmatrix} \boldsymbol{r}_0^{\mathsf{T}} \\ \boldsymbol{r}_1^{\mathsf{T}} \\ \boldsymbol{r}_2^{\mathsf{T}} \end{bmatrix} \boldsymbol{v} = \begin{bmatrix} \boldsymbol{r}_0 \cdot \boldsymbol{v} \\ \boldsymbol{r}_1 \cdot \boldsymbol{v} \\ \boldsymbol{r}_2 \cdot \boldsymbol{v} \end{bmatrix} \tag{39}$$

where $\boldsymbol{r}_i$ 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 $\boldsymbol{r}_0 \cdot \boldsymbol{v} = 0$ and $\boldsymbol{r}_1 \cdot \boldsymbol{v} = 0$ imply $\boldsymbol{v}$ is perpendicular to both rows. Consequently, $\boldsymbol{v}$ is parallel to the cross product $\boldsymbol{r}_0 \times \boldsymbol{r}_1$. 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 4, the cross product approach is used.

Listing 4. 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 = r0xr2 / sqrt(d1);
    }
    else
    {
        eigenvector0 = r1xr2 / sqrt(d2);
    }
}
```

If the other two eigenvalues are well separated, the algorithm of Listing 4 may be used for each eigenvalue. However, if the two eigenvalues are nearly equal, Listing 4 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 $\boldsymbol{W}$ of α_0. Compute unit-length vectors $\boldsymbol{U}$ and $\boldsymbol{V}$ such that $\{\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}\}$ is a right-handed orthonormal set; that is, the vectors are all unit length, mutually perpendicular, and $\boldsymbol{W} = \boldsymbol{U} \times \boldsymbol{V}$. The computations can be done robustly when using floating-point arithmetic, as shown in Listing 5.

Listing 5. Robust computation of $\boldsymbol{U}$ and $\boldsymbol{V}$ for a specified $\boldsymbol{W}$.

```
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 $\boldsymbol{E}$ of $A-\alpha_1 I$ must be a circular combination of $\boldsymbol{U}$ and $\boldsymbol{V}$; that is, $\boldsymbol{E} = x_0\boldsymbol{U}+x_1\boldsymbol{V}$ 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 $J = [\boldsymbol{U}\ \boldsymbol{V}]$ 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 $\boldsymbol{X}$ whose rows are x_0 and x_1, which implies $\boldsymbol{E} = J\boldsymbol{X}$. The 3×3 linear system $(A - \alpha_1 I)\boldsymbol{E} = \boldsymbol{0}$ reduces to the 2×2 linear system $M\boldsymbol{X} = \boldsymbol{0}$. 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 $\boldsymbol{X}$ is perpendicular to the normalized row, and we can choose $\boldsymbol{X}$ 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 $\boldsymbol{X}$ suffices. Listing 6 contains pseudocode for computing a unit-length eigenvector corresponding to α_1.

Listing 6. 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 7.

Listing 7. 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 class NISymmetricEigensolver3x3, also found in the file SymmetricEigensolver3x3.h. The code was written not to use any GTE object.

References

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, Maryland, 4th edition, 2013.