Eigensystems for 3×3 Symmetric Matrices (Revisited)

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

Created: October 9, 2006
Last Modified: May 19, 2011

Contents

1 About the Previous Version of This Document

The previous version of this document has been a quite popular download and a source of the question: Do you have an implementation to download? I finally decided to implement the method, knowing that the mathematics is simple yet realizing the classic problem with floating-point arithmetic would rear its ugly head. Indeed, it did and led to my rewriting this document.

The previous version had one inaccuracy, which had to be resolved for an implementation. The 3×3 symmetric matrix A has the characteristic equation $p(\lambda) = \det(A - \lambda I) = 0$, where $p(\lambda)$ is a cubic polynomial. The q variable that is calculated during the polynomial root construction is first used to classify the multiplicity of the roots. I mentioned that the case $q > 0$ meant there is exactly one real-valued root to $p(\lambda) = 0$, which implies A is diagonal. This is incorrect; the correct interpretation is that the cubic polynomial has one real-valued root and two non-real-valued roots (complex-valued roots with nonzero imaginary parts). For the problem at hand, all the roots must be real-valued. As it turns out, the only choices for the eigensolver are $q < 0$ (three distinct real-valued roots) or $q = 0$ (all real-valued roots, one having at least a multiplicity of 2).

The numerical problems first occur in computing the roots, as expected, but can be dealt with appropriately. However, eigenvector construction also must be handled carefully. It is necessary to compute correctly the rank of the matrix $M = A - \lambda I$, a number that is 0, 1, or 2 (and not 3 since M is necessarily singular).

2 The Motivation

A standard result from linear algebra is that an $n \times n$ symmetric matrix A with real-valued entries must have n real-valued and unit-length eigenvectors $\mathbf{V}_1$ through $\mathbf{V}_n$ that are mutually orthogonal. Each vector satisfies the *eigenvector equation*, $A\mathbf{V}_i = \lambda_i \mathbf{V}_i$, where λ_i is the eigenvalue associated with the eigenvector. The eigenvalues are not necessarily distinct. Let $R = [\mathbf{V}_1 \cdots \mathbf{V}_n]$, where the columns of the matrix are the eigenvectors; this matrix is orthogonal. Also, let $D = \text{Diag}(\lambda_1, \ldots, \lambda_n)$, a diagonal matrix whose diagonal entries are the eigenvalues. The n eigenvector equations can be written as a single matrix equation, $AR = RD$. Since R is orthogonal, $R^{-1} = R^{\text{T}}$, so equivalently $A = RDR^{\text{T}}$.

Various iterative numerical methods may be applied to A to extract the eigenvectors (stored in R) and eigenvalues (stored in D). One method uses Jacobi transformations to approximate R by a composition of rotation matrices, $Q = Q_1 \cdots Q_k$, with k large enough so that $Q^{\text{T}}AQ$ is effectively diagonal. Another method uses either Givens reductions or Householder reductions to obtain a matrix, Q, in a fixed number of steps so that $B = Q^{\text{T}}AQ$ is tridiagonal. The matrix B is then factored to RDR^{T} using an iterative scheme such as the QR or QL algorithms. These methods are designed to be accurate and robust.

When $n = 3$, the polynomial $p(\lambda) = \det(A - \lambda I)$ has degree 3. Closed-form equations exist for the roots of the polynomial, so in theory it is possible to construct the eigenvalues and eigenvectors using a noniterative approach. An application that frequently computes the eigenvalues and eigenvectors of A will benefit from this approach if the computational time is less than that of the iterative approach. At the same time, the noniterative approach must be numerically robust.

3 Roots of Cubic Polynomials

We first look at the construction of roots to any cubic polynomial with real-valued coefficients. Those cubic polynomials obtained as characteristic polynomials $\det(A - \lambda I)$ have some unique properties that allow the root finding to be specialized.

3.1 The General Equation

Consider the cubic polynomial equation $\lambda^3 - c_2\lambda^2 + c_1\lambda - c_0 = 0$, where the coefficients c_0, c_1, and c_2 are real-valued numbers. The squared term may be eliminated by the change of variables, $\lambda = \xi + c_2/3$, leading to the equation $\xi^3 + a\xi + b = 0$, where

$$a = \frac{3c_1 - c_2^2}{3}, \quad b = \frac{-2c_2^3 + 9c_1c_2 - 27c_0}{27}, \quad q = \frac{b^2}{4} + \frac{a^3}{27} \tag{1}$$

The quantity q is not part of the coefficients of the ξ-polynomial, but it is used in the equations for the roots. There are a few cases to consider for the root construction. The ξ-roots are computed; the λ-roots are obtained from the change of variables $\lambda = \xi + c_2/3$.

Case 1. Let $a = 0$ and $b = 0$. The polynomial equation is $\xi^3 = 0$, so zero is a root of multiplicity 3. The ξ-roots are

$$\xi_0 = \xi_1 = \xi_2 = 0$$

Case 2. Let $a = 0$ and $b \neq 0$. The polynomial equation is $\xi^3 = -b$. Define $\rho = (-b)^{1/3}$, the real-valued cube root of $-b$. If $b > 0$, the ξ-roots are

$$\xi_0 = \rho, \quad \xi_1 = \rho(\cos(\pi/3) + i\sin(\pi/3)), \quad \xi_2 = \rho(\cos(\pi/3) - i\sin(\pi/3))$$

If $b < 0$, the ξ-roots are

$$\xi_0 = \rho, \quad \xi_1 = \rho(\cos(2\pi/3) + i\sin(2\pi/3)), \quad \xi_2 = \rho(\cos(2\pi/3) - i\sin(2\pi/3))$$

Case 3. Let $a \neq 0$ and $b = 0$. The polynomial equation is $\xi^3 + a\xi = 0$. The ξ-roots are

$$\xi_0 = 0, \quad \xi_1 = \sqrt{-a}, \quad \xi_2 = -\sqrt{-a}$$

Case 4. Let $a \neq 0$ and $b \neq 0$. The trick to solving the equation is to make use of a trigonometric identity,

$$4\cos^3\theta - 3\cos\theta - \cos(3\theta) = 0$$

Make the change of variables $\xi = \rho\cos\theta$ to obtain

$$\rho^3\cos^3\theta + a\rho\cos\theta + b = 0$$

The left-hand side has a form similar to that of the trigonometric identity as long as $(\rho^3, a\rho, b)$ is a vector parallel to $(4, -3, -\cos(3\theta))$. To be parallel, the cross product must be the zero vector,

$$(0,0,0) = (4,-3,-\cos(3\theta)) \times (\rho^3, a\rho, b) = (a\rho\cos(3\theta) - 3b, -\rho^3\cos(3\theta) - 4b, \rho(4a + 3\rho^2))$$

When ρ is not zero, the last and first components may be solved for

$$\rho = 2\sqrt{\frac{-a}{3}}, \;\; \cos(3\theta) = \frac{3b}{a\rho}$$

If $a = 0$, then $\rho = 0$. If $a < 0$, then $\rho = 2\sqrt{|a|/3}$, which is a real-valued number. However, if $a > 0$, then $\rho = 2\sqrt{|a|/3}\, i$, which is pure imaginary (non-real-valued).

The equation $\cos(3\theta) = 3b/(a\rho)$ poses a more subtle problem. If ρ is pure imaginary, we must resort to complex analysis to interpret what it means for the cosine of an angle to equal a complex number. Even if ρ is real-valued, we still need to resort to complex analysis. This is evident when $|2b/(a\rho)| > 1$. We know that for real numbers, $|\cos(3\theta)| \le 1$. When using complex numbers, it is possible for the complex-valued cosine to have magnitude larger than 1. the complex-valued sine function is defined by $\sin(z) = (e^{iz} - e^{-iz})/(2i)$ and the complex-valued cosine function is defined by $\cos(z) = (e^{iz} + e^{-iz})/2$. Thus, the equation $\cos(3\theta) = 3b/(a\rho)$ may be solved using complex numbers using these definitions. That said, notice that if θ is a solution to the equation, then so are $\theta + 2\pi/3$ and $\theta + 4\pi/3$. In summary, the ξ-roots are

$$\xi_0 = \rho\cos(\theta), \;\; \xi_1 = \rho\cos(\theta + 2\pi/3), \;\; \xi_2 = \rho\cos(\theta + 4\pi/3)$$

In an implementation, three cosine evaluations are avoided by using a single sine and a single cosine evaluation. That is, $\cos(\theta+2\pi/3) = -(1/2)\cos(\theta) - (\sqrt{3}/2)\sin(\theta)$ and $\cos(\theta+4\pi/3) = -(1/2)\cos(\theta) + (\sqrt{3}/2)\sin(\theta)$.

All Cases. The four cases may be combined, using some algebraic manipulation, to the following. Define

$$u = (-b/2 + \sqrt{q}\,)^{1/3}, \;\; v = (-b/2 - \sqrt{q}\,)^{1/3} \tag{2}$$

The principal square roots and cube roots are implied by the expressions. The ξ-roots are

$$\xi_0 = u + v, \;\; \xi_1 = -\frac{u+v}{2} + \frac{u-v}{2}\sqrt{-3}, \;\; \xi_2 = -\frac{u+v}{2} - \frac{u-v}{2}\sqrt{-3} \tag{3}$$

Even though the term $\sqrt{-3}$ appears in the expressions, the roots can still be real-valued. This is the case when $u - v$ is a pure imaginary number. Also notice the absence of explicit sine and cosine functions. In fact, these functions will appear when computing the cube roots associated with u and v.

A closer analysis of Equations (1), (2), and (3) will lead to the classification

$$\begin{aligned} &q > 0, \quad \text{one real root, two conjugate complex roots} \\ &q = 0, \quad \text{three real roots of which at least two are equal} \\ &q < 0, \quad \text{three distinct real roots} \end{aligned} \tag{4}$$

3.2 The Characteristic Equation

To match the notation of the previous section, we will consider the negated characteristic equation,

$$0 = -\det(A - \lambda I) = -\det\begin{bmatrix} a_{00} - \lambda & a_{01} & a_{02} \\ a_{01} & a_{11} - \lambda & a_{12} \\ a_{02} & a_{12} & a_{22} - \lambda \end{bmatrix} = \lambda^3 - c_2\lambda^2 + c_1\lambda - c_0$$

where

$$\begin{aligned} c_0 &= a_{00}a_{11}a_{22} + 2a_{01}a_{02}a_{12} - a_{00}a_{12}^2 - a_{11}a_{02}^2 - a_{22}a_{01}^2 \\ c_1 &= a_{00}a_{11} - a_{01}^2 + a_{00}a_{22} - a_{02}^2 + a_{11}a_{22} - a_{12}^2 \\ c_2 &= a_{00} + a_{11} + a_{22} \end{aligned}$$

The roots of the cubic polynomial may be computed as shown in the previous section for the general equation. From that section, the relationship $\rho = \sqrt{-a/3}$ leads to

$$\cos(3\theta) = \frac{3b}{a\rho} = \frac{-b/2}{(-a/3)\sqrt{-a/3}}$$

Thinking of the right-hand side of the $\cos(3\theta)$ equation as a ratio of the opposite side of a triangle to its hypotenuse, the adjacent side has a value

$$\sqrt{((-a/3)\sqrt{-a/3})^2 - (-b/2)^2} = \sqrt{-q}$$

We may instead compute the tangent of 3θ as the ratio of opposite to adjacent. To compute θ in the correct quadrant, though, an implementation will use instead `atan2` and compute

$$\theta = \text{atan2}(\sqrt{-q}, -b/2)/3$$

An ANSI implementation of `atan2` will return zero when its two arguments are both zero, so there is no need to trap this special case. The roots are therefore

$$\lambda_0 = c_2/3 + 2\rho\cos(\theta), \quad \lambda_1 = c_2/3 - \rho\left(\cos(\theta) + \sqrt{3}\,\sin(\theta)\right), \quad \lambda_2 = c_2/3 - \rho\left(\cos(\theta) - \sqrt{3}\,\sin(\theta)\right) \tag{5}$$

To revisit the classification of roots, consider the following. If $a = 0$, then $\rho = 0$ and the roots are

$$\lambda_0 = \lambda_1 = \lambda_2 = c_2/3$$

It is necessarily the case that $b = 0$ and $q = 0$. If $a \neq 0$, it is necessarily the case that $a < 0$. Suppose that also $q = 0$. If $b \leq 0$, it must be that $\theta = 0$, in which case the roots are

$$\lambda_0 = c_2/3 + 2\rho, \quad \lambda_1 = c_2/3 - \rho, \quad \lambda_2 = c_2/3 - \rho$$

If instead $b > 0$, it must be that $\theta = \pi/3$, in which case the roots are

$$\lambda_0 = c_2/3 + \rho, \quad \lambda_1 = c_2/3 - 2\rho, \quad \lambda_2 = c_2/3 + \rho$$

In either case, there are two distinct roots, one of them repeated. If $q \neq 0$, it must be that $q < 0$. The three real-valued roots of Equation (5) are distinct.

3.3 Understanding the Numerical Issues

Now let us take a closer look at the a, b, and q values for polynomials with only real-valued roots. This will help us understand how to deal with the numerical problems that arise when attempting to compute the roots using floating-point arithmetic.

Let the real-valued roots be λ_0, λ_1, and λ_2, not necessarily distinct. For the sake of argument, let the roots be ordered by $\lambda_0 \le \lambda_1 \le \lambda_2$. The polyonomial is

$$\begin{aligned} p(\lambda) &= (\lambda-\lambda_0)(\lambda-\lambda_1)(\lambda-\lambda_2) \\ &= \lambda^3 - (\lambda_0+\lambda_1+\lambda_2)\lambda^2 + (\lambda_0\lambda_1+\lambda_0\lambda_2+\lambda_1\lambda_2)\lambda - (\lambda_0\lambda_1\lambda_2) \\ &= \lambda^3 - c_2\lambda^2 + c_1\lambda - c_0 \end{aligned}$$

Equating the c_i with the root expressions and substituting into Equation (1) leads to

$$\begin{aligned} a &= -((\lambda_1-\lambda_0)^2 + (\lambda_2-\lambda_0)^2 + (\lambda_2-\lambda_1)^2)/6 \\ b &= (2\lambda_0-\lambda_1-\lambda_2)(\lambda_0-2\lambda_1+\lambda_2)(-\lambda_0-\lambda_1+2\lambda_2)/27 \\ q &= -(\lambda_1-\lambda_0)^2(\lambda_2-\lambda_0)^2(\lambda_2-\lambda_1)^2/108 \end{aligned} \tag{6}$$

These expressions make it clear that $a \le 0$ and $q \le 0$. For a to be zero, all the roots must be equal. For q to be zero, at least two roots must be equal. Any of three conditions makes b zero. The first condition is $\lambda_0 = (\lambda_1+\lambda_2)/2$, which states that λ_0 is the average of λ_1 and λ_2. Since λ_0 is the smallest root, the only way it can be the average is if $\lambda_0 = \lambda_1 = \lambda_2$. The second condition is $\lambda_2 = (\lambda_0+\lambda_1)/2$, which similarly implies $\lambda_0 = \lambda_1 = \lambda_2$. The third condition is $\lambda_1 = (\lambda_0+\lambda_2)/2$, which can happen even when there are three distinct roots ($q < 0$). Notice that $b = 0$ and $q < 0$ imply $\theta = \pi/2$, in which case the roots are $\lambda_0 = c_2/3 - \rho\sqrt{3}$, $\lambda_1 = c_2/3$, and $\lambda_2 = c_2/3 + \rho\sqrt{3}$.

The maximum difference of the roots is $\Delta = \lambda_2 - \lambda_0$. Define $\lambda_1 - \lambda_0 = \mu\Delta$ and $\lambda_2 - \lambda_1 = (1-\mu)\Delta$; it is necessary that $\mu \in [0,1]$. An algebraic construction will show that

$$\begin{aligned} a &= -(\mu^2-\mu+1)\Delta^2/3 && \in [-\Delta^2/3, -\Delta^2/4] \\ b &= (-2\mu^3+3\mu^2+3\mu-2)\Delta^3/27 && \in [-2\Delta^3/27, 2\Delta^3/27] \\ q &= -\mu^2(1-\mu)^2\Delta^6/108 && \in [-\Delta^6/108, -\Delta^6/1728] \end{aligned} \tag{7}$$

The calculations that are the least robust are for a root that is repeated two or three times, in theory, but the numerical round-off errors in computing c_0, c_1, c_2, a, b, and q make it appear as if the roots are all distinct (and nearly the same floating-point values).

In the case of three repeated roots, theoretically $\Delta = 0$ but numerically Δ is a very small floating-point number. Theoretically, $a = 0$ and $q = 0$ but numerically $a = O(\Delta^2)$ and $q = O(\Delta^6)$. You expect both a and q to be nearly zero with q a much smaller quantity than a.

In the case of two distinct roots, say $\lambda_0 = \lambda_1 < \lambda_2$, theoretically $\mu = 0$ but numerically μ is a small floating-point number. Δ is relatively large in comparison, so you expect that a is sufficiently far away from zero (it looks like $-\Delta^2/3$). However, you expect that q is nearly zero (it is $O(\mu^2)$).

There are a few sources of numerical round-off errors that cause the problems. First, the entries of A already can reflect round-off errors in whatever process was using A. For example, if you choose a diagonal matrix $D = dI$ for some scalar d, you may think of its diagonal entries as exactly the eigenvalues you seek (multiplicity 3). Now numerically generate a rotation matrix R and compute $A = RDR^{\mathrm{T}}$. Theoretically, $A = D$ but numerically the off-diagonal terms are numbers very close to zero. This can lead to the construction of three distinct roots, all of them nearly the same floating-point number. Second, the computation of c_0, c_1, and c_2 can have some subtractive cancellation that affects further computations. Third, the computation of

a, b, and q also are subject to subtractive cancellation. The worst offenders are the computations `sqrt(-a/3)` and `sqrt(-q)`. If the arguments are theoretically zero but numerically nonzero and on the order of 10^{-6}, a number that is effectively zero when using single-precision floating-point arithmetic, the square roots are on the order of 10^{-3}. Thus, the error is magnified by these operations.

If you precondition A by dividing by its maximum magnitude entry when that maximum is larger than 1, and if you use high-precision calculations, the total of all these round-off errors tend not to affect the accuracy of the root calculations. The problems, however, show up when you attempt to construct the eigenvectors by solving the equations $(A - \lambda I)\mathbf{V} = \mathbf{0}$. It is important to correctly compute $\text{rank}(A - \lambda I)$. Small errors in the eigenvalue calculations can lead to misclassification of the rank.

3.4 Computing the Roots

As mentioned, A is preconditioned by dividing by its maximum magnitude entry when that maximum is larger than 1. The roots are computed in double precision. The pseudocode is shown below. The global variables are `inv3`, which is computed once as $1/3$, and `root3`, which is computed once as $\sqrt{3}$.

```
void ComputeRoots (Matrix3 A, double root[3])
{
    double a00 = (double)A[0][0];
    double a01 = (double)A[0][1];
    double a02 = (double)A[0][2];
    double a11 = (double)A[1][1];
    double a12 = (double)A[1][2];
    double a22 = (double)A[2][2];
    double c0 = a00*a11*a22 + 2.0*a01*a02*a12 - a00*a12*a12 - a11*a02*a02 - a22*a01*a01;
    double c1 = a00*a11 - a01*a01 + a00*a22 - a02*a02 + a11*a22 - a12*a12;
    double c2 = a00 + a11 + a22;
    double c2Div3 = c2*inv3;
    double aDiv3 = (c1 - c2*c2Div3)*inv3;
    if (aDiv3 > 0.0) { aDiv3 = 0.0; }
    double mbDiv2 = 0.5*(c0 + c2Div3*(2.0*c2Div3*c2Div3 - c1));
    double q = mbDiv2*mbDiv2 + aDiv3*aDiv3*aDiv3;
    if (q > 0.0) { q = 0.0; }
    double magnitude = sqrt(-aDiv3);
    double angle = atan2(sqrt(-q),mbDiv2)*inv3;
    double cs = cos(angle);
    double sn = sin(angle);
    root[0] = c2Div3 + 2.0*magnitude*cs;
    root[1] = c2Div3 - magnitude*(cs + root3*sn);
    root[2] = c2Div3 - magnitude*(cs - root3*sn);

    // Sort the roots here to obtain root[0] <= root[1] <= root[2].
}
```

4 Computing the Eigenvectors

The main problem caused by floating-point round-off errors is the correct classification of the rank of $A-\lambda I$. The theoretical results are discussed first, followed by the numerical implementation that attempts to be robust. In this discussion, the eigenvalues are ordered by $\lambda_0 \le \lambda_1 \le \lambda_2$. The related singular matrices are $M_i = A - \lambda_i I$ for all i. The eigenvectors selected by the algorithm are unit-length, mutually perpendicular, and named $\mathbf{V}_0$, $\mathbf{V}_1$, and $\mathbf{V}_2$.

4.1 Theoretical Construction

The eigenvector construction depends on the multiplicity of the eigenvalues. Each possibility is discussed here.

4.1.1 Three Distinct Eigenvalues

Suppose that the eigenvalues are distinct: $\lambda_0 < \lambda_1 < \lambda_2$. Each eigenvalue has a corresponding eigenspace of dimension 1. The construction for the eigenspace of λ_0 is shown here. An eigenvector $\mathbf{V}_0$ is a nonzero vector that solves $M_0\mathbf{V}_0 = \mathbf{0}$. The fact that the eigenspace must be 1-dimensional means that $\text{rank}(M_0) = 2$. Two rows of M_0 must be linearly independent, the other row dependent on them. For the sake of argument, suppose the first two rows of M_0 are linearly independent. As 3×1 vectors, call them $\mathbf{r}_0$ and $\mathbf{r}_1$. As a block matrix of row vectors,

$$M_0 = \left[\begin{array}{c} \mathbf{r}_0^{\mathrm{T}} \\ \hline \mathbf{r}_1^{\mathrm{T}} \\ \hline \alpha\mathbf{r}_0^{\mathrm{T}} + \beta\mathbf{r}_1^{\mathrm{T}} \end{array}\right]$$

for some scalars α and β. When you multiply the matrix times an eigenvector, you get

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = M_0\mathbf{V}_0 = \left[\begin{array}{c} \mathbf{r}_0^{\mathrm{T}} \\ \hline \mathbf{r}_1^{\mathrm{T}} \\ \hline \alpha\mathbf{r}_0^{\mathrm{T}} + \beta\mathbf{r}_1^{\mathrm{T}} \end{array}\right]\mathbf{V}_0 = \begin{bmatrix} \mathbf{r}_0^{\mathrm{T}}\mathbf{V}_0 \\ \mathbf{r}_1^{\mathrm{T}}\mathbf{V}_0 \\ \alpha\mathbf{r}_0^{\mathrm{T}}\mathbf{V}_0 + \beta\mathbf{r}_1^{\mathrm{T}}\mathbf{V}_0 \end{bmatrix}$$

Thus, $\mathbf{r}_0^{\mathrm{T}}\mathbf{V}_0 = 0$ and $\mathbf{r}_1^{\mathrm{T}}\mathbf{V}_0 = 0$, which says geometrically that $\mathbf{V}_0$ is perpendicular to both $\mathbf{r}_0$ and $\mathbf{r}_1$. The cross product of the rows has this property, so

$$\mathbf{V}_0 = \frac{\mathbf{r}_0 \times \mathbf{r}_1}{|\mathbf{r}_0 \times \mathbf{r}_1|}$$

Similar constructions applied to M_1 and M_2 produce $\mathbf{V}_1$ and $\mathbf{V}_2$.

4.1.2 Two Distinct Eigenvalues

Let $\lambda_0 = \lambda_1 < \lambda_2$. Since λ_2 has multiplicity 1, the construction in the previous section produces an eigenvector $\mathbf{V}_2$ from the matrix M_2.

The eigenvalue λ_0 has multiplicity 2, so $\text{rank}(M_0) = 1$. One row of M_2 is linearly independent and the other rows are multiples of it. For the sake of argument, suppose the first row of M_0 is linearly independent. As a 3×1 vector, call it $\mathbf{r}_0$. As a block matrix of row vectors,

$$M_0 = \left[\begin{array}{c} \mathbf{r}_0^{\mathrm{T}} \\ \hline \alpha\mathbf{r}_0^{\mathrm{T}} \\ \hline \beta\mathbf{r}_0^{\mathrm{T}} \end{array}\right]$$

for some scalars α and β. When you multiply the matrix times an eigenvector, you get

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = M_0\mathbf{V}_i = \left[\begin{array}{c} \mathbf{r}_0^{\mathrm{T}} \\ \hline \alpha\mathbf{r}_0^{\mathrm{T}} \\ \hline \beta\mathbf{r}_0^{\mathrm{T}} \end{array}\right]\mathbf{V}_i = \begin{bmatrix} \mathbf{r}_0^{\mathrm{T}}\mathbf{V}_i \\ \alpha\mathbf{r}_0^{\mathrm{T}}\mathbf{V}_i \\ \beta\mathbf{r}_0^{\mathrm{T}}\mathbf{V}_i \end{bmatrix}$$

for $i = 0$ or $i = 1$. Thus, $\mathbf{r}_0^{\mathrm{T}}\mathbf{V}_i = 0$, which says geometrically that the $\mathbf{V}_i$ are in the plane perpendicular to $\mathbf{r}_0$. Any two unit-length and mutually perpendicular vectors in this plane may be chosen for the eigenvectors associated with λ_0.

4.1.3 One Distinct Eigenvalue

If λ_0 has multiplicity 3, then A was already a scalar times the identity, $A = \lambda_0 I$. Any orthonormal basis of space may be chosen for the eigenvectors. The simplest is

$$\mathbf{V}_0 = (1, 0, 0), \;\; \mathbf{V}_1 = (0, 1, 0), \;\; \mathbf{V}_2 = (0, 0, 1)$$

It must be that $\text{rank}(M_0) = \text{rank}(M_1) = \text{rank}(M_2) = 0$ (the matrices are all the zero matrix).

4.1.4 Theoretical Algorithm

The pseudocode for the eigensolver is the following. The `ComputeRoots` function was described previously. The othre functions will be described later in this section.

```
void Eigensolver (Matrix3 A, Real evalue[3], Vector3 evector[3])
{
    double root[3];
    ComputeRoots(A,root);
    evalue[0] = (Real)root[0];
    evalue[1] = (Real)root[1];
    evalue[2] = (Real)root[2];

    Matrix3 M0 = A - evalue[0]*I;  // I is the identity matrix
    int rank0 = ComputeRank(M0);
    if (rank0 == 0)
    {
        // evalue[0] = evalue[1] = evalue[2]
        evector[0] = Vector3(1,0,0);
        evector[1] = Vector3(0,1,0);
        evector[2] = Vector3(0,0,1);
        return;
    }

    if (rank0 == 1)
    {
        // evalue[0] = evalue[1] < evalue[2]
        GetComplement2(M0.GetRow(0),evector[0],evector[1]);
        evector[2] = evector[0].Cross(evector[1]);
        return;
    }

    // rank0 == 2
    GetComplement1(M0.GetRow(0),M0.GetRow(1),evector[0]);

    Matrix3 M1 = A - evalue[1]*I;
    int rank1 = ComputeRank(M1);  // zero rank detected earlier, rank1 must be positive
    if (rank1 == 1)
    {
        // evalue[0] < evalue[1] = evalue[2]
        GetComplement2(evector[0],evector[1],evector[2]);
        return;
    }

    // rank1 == 2
    GetComplement1(M1.GetRow(0),M1.GetRow(1),evector[1]);

    // rank2 == 2 (eigenvalues must be distinct at this point, rank2 must be 2)
    evector[2] = evector[0].Cross(evector[1]);
}
```

The function `GetComplement1` simply computes the normalized cross product of its first two arguments, assigning the result to the last argument.

```
void GetComplement1 (Vector3 U, Vector3 V, Vector3& W)
{
    W = U.Cross(V);
    W.Normalize();
}
```

The function `GetComplement2` constructs two unit-length and perpendicular vectors in the plane perpendicular to the first argument. Those vectors are assigned to the last two arguments of the function. There are infinitely many possibilities, but the simplest to compute that is also numerically robust is listed here.

```
void GetComplement1 (Vector3 U, Vector3& V, Vector3& W)
{
    U.Normalize();
    if (|U[0]| >= |U[1]|)
    {
        Real invLength = 1/sqrt(U[0]*U[0] + U[2]*U[2]);
        V[0] = -U[2]*invLength;
        V[1] = 0;
        V[2] = U[0]*invLength;
        W[0] = U[1]*V[2];
        W[1] = U[2]*V[0] - U[0]*V[2];
        W[2] = -U[1]*V[0];
    }
    else
    {
        Real invLength = 1/sqrt(U[1]*U[1] + U[2]*U[2]);
        V[0] = 0;
        V[1] = U[2]*invLength;
        V[2] = -U[1]*invLength;
        W[0] = U[1]*V[2] - U[2]*V[1];
        W[1] = -U[0]*V[2];
        W[2] = U[0]*V[1];
    }
}
```

The function `ComputeRank` is the workhorse. A side effect of the function is that the input matrix has been modified so that the linearly independent rows occur first in the matrix for access by the caller. Assuming exact arithmetic, a robust implementation is the following.

```
int ComputeRank (Matrix3& M)
{
    // Compute the maximum magnitude matrix entry.
    Real abs, save, max = -1;
    int row, col, maxRow = -1, maxCol = -1;
    for (row = 0; row < 3; row++)
    {
        for (col = row; col < 3; col++)
        {
```

```
        abs = |M[row][col]|;
        if (abs > max)
        {
            max = abs;
            maxRow = row;
            maxCol = col;
        }
    }
}
if (max == 0)
{
    // The rank is 0.  The eigenvalue has multiplicity 3.
    return 0;
}

// The rank is at least 1.  Swap the row containing the maximum-magnitude
// entry with row 0.
if (maxRow != 0)
{
    for (col = 0; col < 3; col++)
    {
        save = M[0][col];
        M[0][col] = M[maxRow][col];
        M[maxRow][col] = save;
    }
}

// Row-reduce the matrix...

// Scale row 0 to generate a 1-valued pivot.
Real invMax = 1/M[0][maxCol];
M[0][0] *= invMax;
M[0][1] *= invMax;
M[0][2] *= invMax;

// Eliminate the maxCol column entries in rows 1 and 2.
if (maxCol == 0)
{
    M[1][1] -= M[1][0]*M[0][1];
    M[1][2] -= M[1][0]*M[0][2];
    M[2][1] -= M[2][0]*M[0][1];
    M[2][2] -= M[2][0]*M[0][2];
    M[1][0] = 0;
    M[2][0] = 0;
}
else if (maxCol == 1)
{
    M[1][0] -= M[1][1]*M[0][0];
    M[1][2] -= M[1][1]*M[0][2];
    M[2][0] -= M[2][1]*M[0][0];
    M[2][2] -= M[2][1]*M[0][2];
    M[1][1] = 0;
```

```
    M[2][1] = 0;
}
else
{
    M[1][0] -= M[1][2]*M[0][0];
    M[1][1] -= M[1][2]*M[0][1];
    M[2][0] -= M[2][2]*M[0][0];
    M[2][1] -= M[2][2]*M[0][1];
    M[1][2] = 0;
    M[2][2] = 0;
}

// Compute the maximum-magnitude entry of the last two rows of the
// row-reduced matrix.
max = -1;
maxRow = -1;
maxCol = -1;
for (row = 1; row < 3; row++)
{
    for (col = 0; col < 3; col++)
    {
        abs = |M[row][col]|;
        if (abs > max)
        {
            max = abs;
            maxRow = row;
            maxCol = col;
        }
    }
}
if (max == 0)
{
    // The rank is 1.  The eigenvalue has multiplicity 2.
    return 1;
}

// If row 2 has the maximum-magnitude entry, swap it with row 1.
if (maxRow == 2)
{
    for (col = 0; col < 3; col++)
    {
        save = M[1][col];
        M[1][col] = M[2][col];
        M[2][col] = save;
    }
}

// Scale row 1 to generate a 1-valued pivot.
invMax = 1/M[1][maxCol];
M[1][0] *= invMax;
M[1][1] *= invMax;
M[1][2] *= invMax;
```

```
    // The rank is 2.  The eigenvalue has multiplicity 1.
    return 2;
}
```

The first part of the pseudocode checks to see if M is the zero matrix. If it is, the rank is zero. If it is not, the row with the maximum-magnitude entry is swapped with row 0 and used to row-reduce the remaining rows. For the sake of argument, suppose that the maximum-magnitude entry, call this value μ, occurs in row 2 and column 1.

$$M = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & \mu & m_{22} \end{bmatrix}$$

The last row is swapped with the first row and then is divided by μ to obtain

$$M' = \begin{bmatrix} m'_{20} & 1 & m'_{22} \\ m_{10} & m_{11} & m_{12} \\ m_{00} & m_{01} & m_{02} \end{bmatrix}$$

where $m'_{20} = m_{20}/\mu$ and $m'_{22} = m_{22}/\mu$. The row reduction leads to

$$M'' = \begin{bmatrix} m'_{20} & 1 & m'_{22} \\ m_{10} - m_{11}m'_{20} & 0 & m_{12} - m_{11}m'_{22} \\ m_{00} - m_{01}m'_{20} & 0 & m_{02} - m_{01}m'_{22} \end{bmatrix} = \begin{bmatrix} m''_{00} & 1 & m''_{02} \\ m''_{10} & 0 & m''_{12} \\ m''_{20} & 0 & m''_{22} \end{bmatrix}$$

where the last equality defines the double-primed quantities.

The same process is used to determine whether one of the last two rows of M'' is another linearly independent vector. If all the entries in the last two rows are zero, then the rank of M is 1. The row $(m'_{20}, 1, m'_{22}$ is accessed by the caller of `ComputeRank` and is then passed to `GetComplement2`.

Otherwise, the rank of M'' is 2. Another row-reduction is applied. For the sake of argument, suppose that the maximum-magnitude entry, call it ν, of the last two rows of M'' is in row 2 and column 1. Then

$$M'' = \begin{bmatrix} m''_{00} & 1 & m''_{02} \\ \nu & 0 & m''_{12} \\ m''_{20} & 0 & m''_{22} \end{bmatrix}$$

No row swapping is necessary, so divide row 1 by ν.

$$M''' = \begin{bmatrix} m''_{00} & 1 & m''_{02} \\ 1 & 0 & m'''_{12} \\ m''_{20} & 0 & m''_{22} \end{bmatrix}$$

where $m'''_{12} = m''_{12}/nu$. The rows $(m''_{00}, 1, m''_{02})$ and $(1, 0, m'''_{12})$ are accessed by the caller of `ComputeRank` and are then passed to `GetComplement1`. It is not necessary to row-reduce the last row, but if you did, it will be the zero vector $(0, 0, 0)$.

4.2 Numerical Construction

The problem in an implementation is determining the multiplicity of a root. The theoretical classification is in Equation (4). For the eigenvalue problem we know that $q \leq 0$. If $q < 0$, there are three distinct eigenvalues. You expect that $\text{rank}(M_0) = \text{rank}(M_1) = \text{rank}(M_2) = 2$. Numerically, it is possible that q is just slightly negative (due to round-off errors) yet the numerically computed ranks are not all 2.

In the event that theoretically $q = 0$, the condition that distinguishes between one or two distinct eigenvalues depends on the parameter a. If $a = 0$, there is only one distinct eigenvalue; otherwise, $a < 0$ and there are two distinct eigenvalues. If $a = 0$ theoretically, you expect that $\text{rank}(M_0) = \text{rank}(M_1) = \text{rank}(M_2) = 0$. it is possible numerically that a is just slightly negative (due to round-off errors) yet the numerically computed ranks for the M_i are not all zero.

The pseudocode for `ComputeRank` has exact comparisons of the maximum-magnitude entry to zero, because the pseudocode assumes exact arithmetic. Of course, equality comparisons are generally not robust when using floating-point arithmetic. I implemented the algorithm anyway, but compared the magnitudes of the matrix entries to a small tolerance. The pseudocode

```
if (max == 0)
{
    // The rank is 0.  The eigenvalue has multiplicity 3.
    return 0;
}
```

was implemented as

```
Real epsilon = (Real)1e-05;
if (max < epsilon)
{
    // The rank is (numerically) 0.  The eigenvalue has multiplicity 3.
    return 0;
}
```

The specified epsilon led to fewer misclassifications than did the canned value `Mathf::ZERO_TOLERANCE`. Similarly, the pseudocode

```
if (max == 0)
{
    // The rank is 1.  The eigenvalue has multiplicity 2.
    return 1;
}
```

was implemented as

```
if (max < epsilon)
{
    // The rank is (numerically) 1.  The eigenvalue has multiplicity 2.
```

```
    return 1;
}
```

Using this approach but with double-precision arithmetic for computing the roots, `Eigensolver` worked quite well. The actual code was sprinkled with assertions to try to trap misclassifications and see what the results were. The modified pseudocode is

```
void Eigensolver (Matrix3 A, float evalue[3], Vector3 evector[3])
{
    double root[3];
    ComputeRoots(A,root);
    evalue[0] = (float)root[0];
    evalue[1] = (float)root[1];
    evalue[2] = (float)root[2];

    Matrix3 M0, M1, M2;
    int rank0, rank1, rank2;
    Vector3 row0, row1;

    M0 = A - evalue[0]*I;  // I is the identity matrix
    rank0 = ComputeRank(M0);
    if (rank0 == 0)
    {
        // evalue[0] = evalue[1] = evalue[2]
        M1 = A - evalue[1]*I;
        M2 = A - evalue[2]*I;
        rank1 = ComputeRank(M1);
        rank2 = ComputeRank(M2);
        assert(rank1 == 0 && rank1 == 0);

        evector[0] = Vector3(1,0,0);
        evector[1] = Vector3(0,1,0);
        evector[2] = Vector3(0,0,1);
        return;
    }

    if (rank0 == 1)
    {
        // evalue[0] = evalue[1] < evalue[2]
        M1 = A - evalue[1]*I;
        M2 = A - evalue[2]*I;
        rank1 = ComputeRank(M1);
        rank2 = ComputeRank(M2);
        assert(rank1 == 1 && rank1 == 2);

        row0 = M0.GetRow(0);
        row0.Normalize();
        assert(row0 != Vector3::ZERO);

        GetComplement2(row0,evector[0],evector[1]);
        evector[2] = evector[0].Cross(evector[1]);
        assert(evector[2] != Vector3::ZERO);
```

```
        return;
    }

    // rank0 == 2
    row0 = M0.GetRow(0);
    row1 = M0.GetRow(1);
    row0.Normalize();
    row1.Normalize();
    assert(row0 != Vector3::ZERO && row1 != Vector3::ZERO);
    GetComplement1(row0,row1,evector[0]);
    assert(evector[0] != Vector3::ZERO);

    M1 = A - evalue[1]*I;
    rank1 = ComputeRank(M1);
    assert(rank1 > 0);
    if (rank1 == 1)
    {
        // evalue[0] < evalue[1] = evalue[2]
        M2 = A - evalue[2]*I;
        rank2 = ComputeRank(M2);
        assert(rank2 == 1);
        GetComplement2(evector[0],evector[1],evector[2]);
        return;
    }

    // rank1 == 2
    row0 = M1.GetRow(0);
    row1 = M1.GetRow(1);
    row0.Normalize();
    row1.Normalize();
    assert(row0 != Vector3::ZERO && row1 != Vector3::ZERO);
    GetComplement1(row0,row1,evector[1]);
    assert(evector[1] != Vector3::ZERO);

    // rank2 == 2 (eigenvalues must be distinct at this point, rank2 must be 2)
    M2 = A - evalue[2]*I;
    rank2 = ComputeRank(M2);
    assert(rank2 == 2);
    evector[2] = evector[0].Cross(evector[1]);
    assert(evector[2] != Vector3::ZERO);
}
```

The experiment involves generation of 2^{28} symmetric matrices. A diagonal matrix $D = (d_0, d_1, d_2)$, $d_i \in [-1, 1]$, and a rotation matrix R are randomly generated, and then the matrix is computed by $A = RDR^{\mathrm{T}}$. The diagonal matrices are chosen so that $1/4$ of them have $d_0 = d_1 = d_2$, $1/4$ of them have $d_0 = d_1 < d_2$, $1/4$ of them have $d_0 < d_1 = d_2$, and $1/4$ of them have $d_0 < d_1 < d_2$. Of all these matrices, only one assertion was triggered. However, valid eigenvectors were computed in that case.

When the double-precision root finder was replaced with a single-precision one, the number of triggered assertions was significant. Regardless of the experiment, it appears that double-precision root finding is necessary for accurate eigenvector construction. The problem is simply the magnification of error when calling `Sqrt(-a/3)` and `Sqrt(-q)`.

4.3 An Alternate Construction

Having to trap all the misclassifications in `Eigensolver` is tractable. However, here is an alternate construction that performs as well, yet has no assertions.

The maximum-magnitude entries for all three of the $M_i = A - \lambda_i I$ are computed. If all three maxima are smaller than `Mathf::ZERO_TOLERANCE`, the three eigenvalues are assumed to be the same and the eigenvectors are $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$. If at least one maximum-magnitude entry is larger than the tolerance, the matrix producing the maximum of these is processed further to construct the eigenvectors of A. The function `PositiveRank` returns `true` when the matrix has positive rank and also returns the maximum-magnitude entry and the row of the matrix in which this entry occurs.

```
void Eigensolver (Matrix3 A, Real evalue[3], Vector3 evector[3])
{
    Real amax = ComputeMaximumMagnitudeEntry(A);
    if (amax > 1)
    {
        A /= amax;
    }

    double root[3];
    ComputeRoots(A,root);
    evalue[0] = (Real)root[0];
    evalue[1] = (Real)root[1];
    evalue[2] = (Real)root[2];

    Real max[3];
    Vector3 maxrow[3];
    for (i = 0; i < 3; i++)
    {
        Matrix3 M = A - evalue[i]*I;
        if (!PositiveRank(M,max[i],maxrow[i]))
        {
            if (amax > 1)
            {
                for (j = 0; j < 3; j++)
                {
                    evalue[j] *= amax;
                }
            }
            evector[0] = (1,0,0);
            evector[1] = (0,1,0);
            evector[2] = (0,0,1);
            return;
        }
    }

    k0 = index of maxrow[] that contains maximum of max[];
    k1 = (k0+1) modulo 3;
    k2 = (k1+1) modulo 3;
    maxrow[k0].Normalize();
    ComputeVectors(A,maxrow[k0],k1,k2,k0);
```

```
    if (amax > 1)
    {
        for (j = 0; j < 3; j++)
        {
            evalue[j] *= amax;
        }
    }
}
```

Briefly, if the maximum-magnitude entry of A is larger than 1, the matrix is scaled so that its entries are bounded by 1 in magnitude. If the scaling occurs, say by μ to produce $A' = A/\mu$, the eigensolver computes the eigenvalues $\lambda' = \lambda/\mu$ of A', where λ are the corresponding eigenvalues of A. Throughout the code, once the eigenvectors are computed, the eigenvalues need to be rescaled.

The eigenvalues are computed as the roots of a cubic polynomial. Double-precision arithmetic is used to avoid the magnification of error that occurs in the function calls `Sqrt(-a/3)` and `Sqrt(-q)` in the root finder. The ranks of $M_i = A - \lambda_i I$ are computed numerically. If any of them report a rank of 0, the eigenvalues are assumed to be all the same. The eigensolver returns the standard basis vectors as the eigenvectors.

If all the ranks are reported as positive, the matrix M corresponding to the maximum-magnitude entry out of all the matrix entries is processed. The idea is to obtain a row vector whose length is as large as possible in order to reduce the chances of round-off errors affecting the subsequent calculations.

For the sake of argument, suppose $M_2 = A - \lambda_2 I$ produces the largest entry. Let $\mathbf{R}$ be the row that contains that entry. It is necessary that an eigenvector $\mathbf{V}_2$ corresponding to λ_2 be perpendicular to $\mathbf{R}$; this simply follows from $M_2\mathbf{V}_2 = \mathbf{0}$, which says that the eigenvector is perpendicular to each row of M_2. Now choose vectors $\mathbf{U}_0$ and $\mathbf{U}_1$ so that $\{\mathbf{U}_0, \mathbf{U}_1, \mathbf{R}\}$ is an orthonormal set (unit-length and mutually perpendicular vectors). Because $\mathbf{V}_2$ is perpendicular to $\mathbf{R}$,

$$\mathbf{V}_2 = c_0\mathbf{U}_0 + c_1\mathbf{U}_1 \tag{8}$$

for coefficients $c_0 = \mathbf{U}_0 \cdot \mathbf{V}_2$ and $c_1 = \mathbf{U}_1 \cdot \mathbf{V}_2$. We require the the eigenvectors be unit length, so it is necessary that $c_0^2 + c_1^2 = 1$. Multiply Equation (8) by A to obtain

$$\lambda_2\mathbf{V}_2 = A\mathbf{V}_2 = c_0A\mathbf{U}_0 + c_1A\mathbf{U}_1 \tag{9}$$

Dot this equation with $\mathbf{U}_0$ and $\mathbf{U}_1$ to obtain the system of equations

$$\begin{bmatrix} \mathbf{U}_0^{\mathrm{T}}A\mathbf{U}_0 & \mathbf{U}_0^{\mathrm{T}}A\mathbf{U}_1 \\ \mathbf{U}_0^{\mathrm{T}}A\mathbf{U}_1 & \mathbf{U}_1^{\mathrm{T}}A\mathbf{U}_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \lambda_2 \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \tag{10}$$

In fact, this is another eigensystem but only 2×2. It may be solved similarly to how we are handling the 3×3 systems (an algorithm recursive in dimension, so to speak).

Subtract the right-hand-side vector to the left so that the system is of the form

$$\begin{bmatrix} p_{00} & p_{01} \\ p_{01} & p_{11} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{11}$$

where $p_{00} = \mathbf{U}_0^{\mathrm{T}} A \mathbf{U}_0 - \lambda_2$, $p_{01} = \mathbf{U}_0^{\mathrm{T}} A \mathbf{U}_1$, and $p_{11} = \mathbf{U}_1^{\mathrm{T}} A \mathbf{U}_1 - \lambda_2$. If the maximum-magnitude entry of this system is not zero (or sufficiently different from zero when using floating-point arithmetic), then our best bet is to choose the row containing that entry to construct the eigenvector. Suppose that the entry occurs in the first row; then

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \frac{1}{\sqrt{p_{00}^2 + p_{01}^2}} \begin{bmatrix} p_{01} \\ -p_{00} \end{bmatrix}$$

If the entry occurs in the second row, then

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \frac{1}{\sqrt{p_{01}^2 + p_{11}^2}} \begin{bmatrix} -p_{11} \\ p_{01} \end{bmatrix}$$

In either case, we have constructed the eigenvector $\mathbf{V}_2 = c_0 \mathbf{U}_0 + c_1 \mathbf{U}_1$.

If the maximum-magnitude entry of the system is zero (or nearly zero when using floating-point arithmetic), then it is sufficient to choose $\mathbf{V}_2$ to be either of $\mathbf{U}_0$ or $\mathbf{U}_1$. In the implementation, I choose $\mathbf{V}_2 = \mathbf{U}_1$ when the maximum-magnitude entry is in the first row; otherwise, I choose $\mathbf{V}_2 = \mathbf{U}_0$.

Up to now, we have constructed eigenvector $\mathbf{V}_2$. The two other eigenvectors we need to construct must be in the plane perpendicular to $\mathbf{V}_2$. We already know that $\mathbf{R}$ is in that plane. A second vector in the plane is $\mathbf{S} = \mathbf{R} \times \mathbf{V}_2$ and is necessarily unit-length and perpendicular to $\mathbf{R}$. Just as in Equation (8), we may represent

$$\mathbf{V}_0 = c_0 \mathbf{R} + c_1 \mathbf{S}$$

for coefficients $c_0 = \mathbf{R} \cdot \mathbf{V}_0$ and $c_1 = \mathbf{S} \cdot \mathbf{V}_0$. We require the the eigenvectors be unit length, so it is necessary that $c_0^2 + c_1^2 = 1$. Similar to the construction of $\mathbf{V}_2$, namely, Equations (9) through (11), multiply by A and dot the equation with $\mathbf{R}$ and $\mathbf{S}$ to obtain a system

$$\begin{bmatrix} \mathbf{R}^{\mathrm{T}} A \mathbf{R} & \mathbf{R}^{\mathrm{T}} A \mathbf{S} \\ \mathbf{R}^{\mathrm{T}} A \mathbf{S} & \mathbf{S}^{\mathrm{T}} A \mathbf{S} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \lambda_0 \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

Rewrite this as

$$\begin{bmatrix} p_{00} & p_{01} \\ p_{01} & p_{11} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

where $p_{00} = \mathbf{R}^{\mathrm{T}} A \mathbf{R} - \lambda_0$, $p_{01} = \mathbf{R}^{\mathrm{T}} A \mathbf{S}$, and $p_{11} = \mathbf{S}^{\mathrm{T}} A \mathbf{S} - \lambda_0$. This may be solved as we did Equation (11) for c_0 and c_1.

We now have constructed $\mathbf{V}_2$ and $\mathbf{V}_0$. The remaining eigenvector is $\mathbf{V}_1 = \mathbf{V}_2 \times \mathbf{V}_0$. The ordering of the vectors is such they form a right-handed orthonormal set. When written as the columns of a matrix in the order specified, the matrix must be a rotation matrix.

The function `ComputeVectors` in the pseudocode implements the discussion here.

The actual code for the noniterative eigensolver and a sample application are downloadable. The files for Wild Magic 3 are

```
GeometricTools/WildMagic3/Foundation/Numerics/Wm3NoniterativeEigen3x3.{h,cpp}
GeometricTools/WildMagic3/SampleMiscellaneous/NoniterativeEigensolver.{h,cpp}
```

The files for Wild Magic 4 are

```
GeometricTools/WildMagic4/LibFoundation/NumericalAnalysis/Wm4NoniterativeEigen3x3.{h,cpp}
GeometricTools/WildMagic4/SampleFoundation/NoniterativeEigensolver.{h,cpp}
```

5 Performance Measurements

The sample application measures the accuracy of the solutions and the computational time required. The input matrices are randomly generated. The main file has three conditional defines of which only one should be enabled at a time.

The `MEASURE_NONITERATIVE` mode solves 2^{28} eigensystems using the noniterative method. To infer the accuracy, the computed eigenvalues and eigenvectors are used to evaluate the error metric

$$\mu = \max\{|(A - \lambda_0 I)\mathbf{V}_0|, |(A - \lambda_1 I)\mathbf{V}_1|, |(A - \lambda_2 I)\mathbf{V}_2|\}$$

The application keeps track of the maximum of the μ values. In my testing, this was $\max(\mu) = 6.03475e-6$. The iterative solver (using the implicit QL method) reported $\max(\mu) = 1.135875e-6$, so the maximum error of the noniterative approach is comparable to that of the iterative approach.

The `TIMING_NONITERATIVE` mode solves 2^{24} eigensystems using the noniterative approach. A timer is used to compute the execution time. On my Pentium D 3.2 GHz dual-core machine (running the application only on one core), the total time was 37750 milliseconds.

The `TIMING_ITERATIVE` mode solves the same 2^{24} eigensystems as the noniterative approach solves. The total time was 162280 milliseconds. The noniterative approach is comparable in accuracy and has a speed-up of about 4.5, which is a significant savings in computation time.