A Robust Eigensolver for 2×2 Symmetric Matrices

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

Created: July 23, 2016

Contents

1 Introduction **2**

2 The Algorithm **2**

2.1 Computing $\cos(2\theta)$ and $\sin(2\theta)$ 2

2.2 Computing $\cos(\theta)$ and $\sin(\theta)$ 3

2.3 Computing the Diagonal Entries of $R^{\mathsf{T}}AR$ 4

1 Introduction

Let A be a 2×2 symmetric matrix of real numbers. From linear algebra, we know that A has all real-valued eigenvalues and a full basis of eigenvectors. Let $D = \text{Diag}(\lambda_0, \lambda_1)$ be the diagonal matrix whose diagonal entries are the eigenvalues. The eigenvalues are not necessarily distinct. Let $R = [\mathbf{U}_0\ \mathbf{U}_1]$ 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 quadratic polynomial $\det(A - \lambda I) = 0$, where I is the 2×2 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 quadratic polynomial, so in theory one could compute the roots and, for each root, solve the equation $(A - \lambda I)\mathbf{U} = \mathbf{0}$ for unit-length vectors $\mathbf{U}$. Although correct theoretically, this is not always the best approach when using floating-point arithmetic.

2 The Algorithm

The matrices are listed next, where R is chosen to be a rotation matrix with entries including $c = \cos\theta$ and $s = \sin\theta$,

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{01} & a_{11} \end{bmatrix}, \quad R = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \tag{1}$$

and the product that must be diagonal is

$$R^{\mathsf{T}}AR = \begin{bmatrix} c^2a_{00} + 2sca_{01} + s^2a_{11} & sc(a_{00} - a_{11}) + (s^2 - c^2)a_{01} \\ sc(a_{00} - a_{11}) + (s^2 - c^2)a_{01} & c^2a_{11} - 2sca_{01} + s^2a_{00} \end{bmatrix} \tag{2}$$

To be diagonal, we need

$$\begin{aligned} 0 &= sc(a_{00} - a_{11}) + (s^2 - c^2)a_{01} \\ &= ((a_{00} - a_{11})/2)\sin(2\theta) - a_{01}\cos(2\theta) \\ &= ((a_{00} - a_{11})/2, -a_{01}) \cdot (\sin(2\theta), \cos(2\theta)) \end{aligned} \tag{3}$$

This condition is satisfied trivially when $a_{00} = a_{11}$ and $a_{01} = 0$, in which case A is already diagonal—a multiple of the identity matrix. If A is not a multiple of the identity matrix, then the solutions are

$$(\cos(2\theta), \sin(2\theta)) = \pm\frac{((a_{00} - a_{11})/2, a_{01})}{\sqrt{((a_{00} - a_{11})/2)^2 + a_{01}^2}} \tag{4}$$

2.1 Computing $\cos(2\theta)$ and $\sin(2\theta)$

Theoretically, it does not matter which solution we choose in equation (4), so we use the positive sign on the right-hand side of the equation. A simple implementation is shown in Listing 1.

Listing 1. Computation of $\cos(2\theta)$ and $\sin(2\theta)$.

```
// c2 = cos(2*theta), s2 = sin(2*theta)
Real t0 = (a00 - a11)/2, t1 = a01;
Real length = sqrt(t0 * t0 + t1 * t1);
if (length > 0)
{
    // A is not a multiple of the identity.
    c2 = t0 / length;
    s2 = t1 / length;
}
else
{
    // A is a multiple of the identity.
    c2 = 1;
    s2 = 0;
}
```

If t0 and t1 are finite floating-point numbers of large enough magnitude to cause overflow in the computation of (t0 * t0 + t1 * t1), the length variable will be floating-point infinity and the floating-point computations for c2 and s2 will be zero, which is not correct. In practice, if you know your matrix entries will not have such large entries, you can use the code as it is currently shown. However, if you are uncertain and want a more robust solution, the overflow can be avoided. If $|t_0| \geq |t_1|$, then $\sqrt{t_0^2 + t_1^2} = |t_0|\sqrt{1 + (t_1/t_0)^2}$. The floating-point computation of $1 + (t_1/t_0)^2$ will not overflow, producing a finite floating-point number whose square root is also a finite floating-point number. A similar computation applies when $|t_1| \geq |t_0|$. The modified implementation is shown in Listing 2.

Listing 2. Robust computation of $\cos(2\theta)$ and $\sin(2\theta)$ that avoids potential floating-point overflow when computing the argument of the sqrt function.

```
// c2 = cos(2*theta), s2 = sin(2*theta)
Real t0 = (a00 - a11)/2, t1 = a01;
Real maxAbsT = max(abs(t0), abs(t1));
if (maxAbsT > 0)
{
    // A is not a multiple of the identity.  One of c2 or s2 will be 1.
    t0 /= maxAbsT;
    t1 /= maxAbsT;
    Real length = sqrt(t0 * t0 + t1 * t1);
    c2 = t0 / length;
    s2 = t1 / length;  // not zero because a01 is not zero
}
else
{
    // A is a multiple of the identity.
    c2 = 1;
    s2 = 0;
}
```

2.2 Computing $\cos(\theta)$ and $\sin(\theta)$

We now must compute $c = \cos(\theta)$ and $s = \sin(\theta)$ from $\cos(2\theta)$ and $\sin(2\theta)$. If $\sin(2\theta)$ is zero, then we may choose $s = 0$ and $c = 1$, in which case R is the identity matrix and A is already diagonal. When $\sin(2\theta)$ is

not zero, use the identities

$$\sin(\theta) = \sqrt{(1-\cos(2\theta))/2}, \quad \cos(\theta) = \sin(2\theta)/(2\sin(\theta)) \tag{5}$$

Although theoretically correct, if $\cos(2\theta)$ is nearly 1, then $\sin(\theta)$ is nearly 0 and the computation of $\cos(\theta)$ involves a division by a number nearly zero. For numerical robustness, we can avoid this case by choosing the opposite-sign solution of equation (4). Specifically, when c2 is computed and found to be positive, we negate both c2 and s2 (choosing the opposite-sign solution) so that $\sin(\theta)$ is bounded away from zero; that is, if $\cos(2\theta) \le 0$, then $\sin(\theta) = \sqrt{(1-\cos(2\theta))/2} \ge \sqrt{1/2}$ and the computation of $\cos(\theta)$ has a denominator bounded away from zero.

The robust computation of the four trigonometric functions is shown in Listing 3.

Listing 3. Robust computation of $\cos(2\theta)$, $\sin(2\theta)$, $\cos(\theta)$, and $\sin(\theta)$.

```
// c2 = cos(2*theta), s2 = sin(2*theta)
Real t0 = (a00 - a11)/2, t1 = a01;
Real maxAbsT = max(abs(t0), abs(t1));
if (maxAbsT > 0)
{
    // A is not a multiple of the identity.  One of c2 or s2 will be 1.
    // This block is a robust computation of the argument of sqrt().
    t0 /= maxAbsT;
    t1 /= maxAbsT;
    Real length = sqrt(t0 * t0 + t1 * t1);
    c2 = t0 / length;
    s2 = t1 / length;  // not zero because a01 is not zero
    if (c2 > 0)
    {
        // Choose c2 < 0 for robust computation of sin(theta) and cos(theta).
        c2 = -c2;
        s2 = -s2;
    }
}
else
{
    // A is a multiple of the identity.  Choose c2 < 0 for robust
    // computation of sin(theta) and cos(theta).
    c2 = -1;
    s2 = 0;
}

// s = sin(theta), c = cos(theta)
Real s = sqrt(half * (one - c2));  // >= 1/sqrt(2)
Real c = half * s2 / s;
```

2.3 Computing the Diagonal Entries of $R^{\mathsf{T}}AR$

The diagonal entries of $R^{\mathsf{T}}AR = \mathrm{Diag}(\lambda_0, \lambda_1)$ are

$$\lambda_0 = c^2 a_{00} + 2sca_{01} + s^2 a_{11}, \quad \lambda_1 = c^2 a_{11} - 2sca_{01} + s^2 a_{00} \tag{6}$$

Listing 4 shows the computation of the diagonal entries after we have computed c2, s2, c, and s.

Listing 4. Computation of the diagonal entries of $D = R^{\mathsf{T}}AR$. The last block has code to sort the eigenvalues if so desired. If sortType is 0, sorting is irrelevant. If sortType is 1, the eigenvalues are sorted so that the minimum is first and the maximum is second. If sortType is -1, the eigenvalues are sorted so that the maximum is first and the minimum is second. In the sorted cases, the eigenvectors are also reordered as necessary.

```
Real diagonal[2];
Real csqr = c * c, ssqr = s * s, mid = s2 * a01;
diagonal[0] = csqr * a00 + mid + ssqr * a11;
diagonal[1] = csqr * a11 - mid + ssqr * a00;

Real eval[2];
Vector2<Real> evec[2];
if (sortType == 0 || sortType * diagonal[0] <= sortType * diagonal[1])
{
    eval[0] = diagonal[0];  eval[1] = diagonal[1];
    evec[0][0] = c;  evec[0][1] = s;  evec[1][0] = -s;  evec[1][1] = c;
}
else
{
    eval[0] = diagonal[1];  eval[1] = diagonal[0];
    evec[0][0] = s;  evec[0][1] = -c;  evec[1][0] = c;  evec[1][1] = s;
}
```

The Geometric Tools implementation is in the file

GeometricTools/GTEngine/Include/Mathematics/GteSymmetricEigensolver2x2.h

Listing 5 shows the implementation.

Listing 5. The Geometric Tools implementation for the eigensolver for a 2×2 symmetric matrix.

```
// David Eberly, Geometric Tools, Redmond WA 98052
// Copyright (c) 1998-2016
// Distributed under the Boost Software License, Version 1.0.
// http://www.boost.org/LICENSE_1_0.txt
// http://www.geometrictools.com/License/Boost/LICENSE_1_0.txt
// File Version: 3.0.0 (2016/06/19)

#pragma once

#include <GTEngineDEF.h>
#include <algorithm>
#include <array>
#include <cmath>

namespace gte
{

template <typename Real>
class SymmetricEigensolver2x2
{
public:
    // The input matrix must be symmetric, so only the unique elements must
    // be specified: a00, a01, and a11.
    //
    // 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]} is guaranteed to be a
    // right-handed orthonormal set.

    void operator()(Real a00, Real a01, Real a11, int sortType,
        std::array<Real, 2>& eval, std::array<std::array<Real, 2>, 2>& evec)
        const;
};


template <typename Real>
void SymmetricEigensolver2x2<Real>::operator()(Real a00, Real a01, Real a11,
    int sortType, std::array<Real, 2>& eval,
    std::array<std::array<Real, 2>, 2>& evec) const
{
    // Normalize (c2,s2) robustly, avoiding floating-point overflow in the
    // sqrt call.
    Real const zero = (Real)0, one = (Real)1, half = (Real)0.5;
    Real c2 = half * (a00 - a11), s2 = a01;
    Real maxAbsComp = std::max(std::abs(c2), std::abs(s2));
    if (maxAbsComp > zero)
    {
        c2 /= maxAbsComp;  // in [-1,1]
        s2 /= maxAbsComp;  // in [-1,1]
        Real length = sqrt(c2 * c2 + s2 * s2);
        c2 /= length;
        s2 /= length;
        if (c2 > zero)
        {
            c2 = -c2;
            s2 = -s2;
        }
    }
    else
    {
        c2 = -one;
        s2 = zero;
    }

    Real s = sqrt(half * (one - c2));  // >= 1/sqrt(2)
    Real c = half * s2 / s;

    Real diagonal[2];
    Real csqr = c * c, ssqr = s * s, mid = s2 * a01;
    diagonal[0] = csqr * a00 + mid + ssqr * a11;
    diagonal[1] = csqr * a11 - mid + ssqr * a00;

    if (sortType == 0 || sortType * diagonal[0] <= sortType * diagonal[1])
    {
        eval[0] = diagonal[0];
        eval[1] = diagonal[1];
        evec[0][0] = c;
        evec[0][1] = s;
        evec[1][0] = -s;
        evec[1][1] = c;
    }
    else
    {
        eval[0] = diagonal[1];
        eval[1] = diagonal[0];
        evec[0][0] = s;
        evec[0][1] = -c;
        evec[1][0] = c;
        evec[1][1] = s;
    }
}

}
```