A Robust Eigensolver for $3 \times 3$ Symmetric Matrices

David Eberly  
Geometric Tools, LLC  
http://www.geometrictools.com/  
Copyright © 1998-2015. All Rights Reserved.

Created: December 6, 2014

Contents

1 Introduction 2

2 An Iterative Algorithm 2

3 A Variation of the Algorithm 3

4 Implementation of the Algorithm 5
1 Introduction

Let $A$ be a $3 \times 3$ symmetric matrix of real numbers. From linear algebra, we know that $A$ has all real-valued eigenvalues and a full basis of eigenvectors. Let $D = \text{Diagonal}(\lambda_0, \lambda_1, \lambda_2)$ be the diagonal matrix whose diagonal entries are the eigenvalues. The eigenvalues are not necessarily distinct. Let $R = [U_0 \ U_1 \ U_2]$ be an orthogonal matrix whose columns are linearly independent eigenvectors, ordered consistently with the diagonal entries of $D$. That is, $AU_i = \lambda_i U_i$. The eigendecomposition is $A = RDR^T$.

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 $3 \times 3$ identity matrix. The left-hand side of the equation is the determinant of the matrix $A - \lambda I$. Closed-form equations exist for the roots of a cubic polynomial, so in theory one could compute the roots and for each one solve the equation $(A - \lambda I)U = 0$ for nonzero vectors $U$. Although correct theoretically, computing roots of the cubic polynomial using the closed-form equations is known to be a non-robust algorithm (generally).

2 An Iterative Algorithm

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

$$G_{n-1}^T \cdots G_1^T H^T A H G_0 \cdots G_{n-1} = D' = D + E$$

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

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

```
GeometricTools/GTEngine/Include/GteSymmetricEigenSolver.h, inl
```

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

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

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

3 A Variation of the Algorithm

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

\[
H^T A H = \begin{bmatrix} c & s & 0 \\ s & -c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{01} & a_{11} & a_{12} \\ a_{02} & a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} c & s & 0 \\ s & -c & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

\[
= \begin{bmatrix} c^2 a_{00} + 2sca_{01} + s^2 a_{11} & sc(a_{00} - a_{11}) + (s^2 - c^2)a_{01} & ca_{02} + sa_{12} \\ sc(a_{00} - a_{11}) + (s^2 - c^2)a_{01} & c^2 a_{11} - 2sca_{01} + s^2 a_{00} & sa_{02} - ca_{12} \\ ca_{02} + sa_{12} & sa_{02} - ca_{12} & a_{22} \end{bmatrix}
\]

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

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

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

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

\[
P_1 = G_1^T B G_1 = \begin{bmatrix} c_1^2 b_{00} + 2s_1 c_1 b_{01} + s_1^2 b_{11} & s_1 b_{12} & s_1 c_1 (b_{11} - b_{00}) + (c_1^2 - s_1^2) b_{01} \\ s_1 b_{12} & b_{22} & c_1 b_{12} \\ s_1 c_1 (b_{11} - b_{00}) + (c_1^2 - s_1^2) b_{01} & c_1 b_{12} & c_1^2 b_{11} - 2s_1 c_1 b_{01} + s_1^2 b_{00} \end{bmatrix}
\]
We need $P_1$ to be tridiagonal, which requires
\[
0 = s_1 c_1 (b_{11} - b_{00}) + (c_1^2 - s_1^2) b_{01} = \sin(2\theta_1)(b_{11} - b_{00})/2 + \cos(2\theta_1)b_{01}
\]
leading us to two possible choices
\[
(\cos(2\theta_1), \sin(2\theta_1)) = \pm \frac{(b_{00} - b_{11}, 2b_{01})}{\sqrt{(b_{00} - b_{11})^2 + 4b_{01}^2}}
\]
We must extract $c_1 = \cos \theta_1$ and $s_1 = \sin \theta$ from whichever solution we choose. In fact, we will choose $\cos(2\theta_1) \leq 0$ so that $|c_1| \leq |s_1| = s_1$. Let $\sigma = \text{Sign}(b_{00} - b_{11})$; then
\[
(\cos(2\theta_1), \sin(2\theta_1)) = \left( -|b_{00} - b_{11}|, -2\sigma b_{01} \right) / \sqrt{(b_{00} - b_{11})^2 + 4b_{01}^2}, \quad \sin \theta_1 = \sqrt{1 - \cos^2(2\theta_1)}/2, \quad \cos \theta_1 = \frac{\sin(2\theta_1)}{2\sin \theta_1}
\]
Notice that
\[
|\cos \theta_1| = \sqrt{(1 + \cos(2\theta_1))/2} \leq \sqrt{(1 - \cos(2\theta_1))/2} = \sin \theta_1
\]
because we chose $\cos(2\theta_1) \leq 0$.

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

Define $P_0 = B$ and let $P_{i+1} = G^T_{i+1} P_i G_{i+1}$ be the iterative process. The conclusion is that the $(1, 2)$-entry of the output matrix is smaller than the $(1, 2)$-entry of the input matrix, so indeed repeated iterations will drive the $(1, 2)$-entry to zero. If $c_i = \cos \theta_i$ and $s_i = \sin \theta_i$, we have
\[
|p_{12}^{(i+1)}| = |c_i + 1 p_{12}^{(i)}| \leq 2^{-i/2} |p_{12}^{(i)}|
\]
which implies
\[
|p_{12}^{(i)}| \leq 2^{-i/2} |b_{12}|, \quad i \geq 1
\]
In the limit as $i \to \infty$, the $(1, 2)$-entry is forced to zero. In fact the bounds here allow you to select the final $i$ so that $2^{-i/2} |b_{12}|$ is a number whose nearest floating-point representation is zero. The number of iterations of the algorithm is limited by this $i$.

If instead we find that $|b_{01}| \leq |b_{12}|$, we can choose the reflections to be of the form
\[
G_i = \begin{bmatrix}
0 & 1 & 0 \\
c_1 & 0 & s_1 \\
-s_1 & 0 & c_1
\end{bmatrix}
\]
Consider the product
\[
P_1 = G^T_1 B G_1 = \begin{bmatrix}
c_1^2 b_{11} - 2 s_1 c_1 b_{12} + s_1^2 b_{22} & c_1 b_{01} & s_1 c_1 (b_{11} - b_{22}) + (c_1^2 - s_1^2) b_{12} \\
c_1 b_{01} & b_{00} & s_1 b_{01} \\
s_1 c_1 (b_{11} - b_{22}) + (c_1^2 - s_1^2) b_{12} & s_1 b_{01} & s_1^2 b_{11} + 2 s_1 c_1 b_{12} + c_1^2 b_{22}
\end{bmatrix}
\]
Define $\sigma = \text{Sign}(b_{22} - b_{11})$ and choose

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

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

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

## 4 Implementation of the Algorithm

The source code that implements the iterative algorithm for symmetric $3 \times 3$ matrices is in class `SymmetricEigenSolver3x3` found in the files

```
GeometricTools/GTEngine/Include/GteSymmetricEigenSolver3x3.h, inl
```

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

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

template <typename Real>
class SymmetricEigenSolver3x3 {
    public:
        // The input matrix must be symmetric, so only the unique elements must
        // be specified: a00, a01, a02, a11, a12, and a22.
        //
        // If 'aggressive' is 'true', the iterations occur until a superdiagonal
        // entry is exactly zero. If 'aggressive' is 'false', the iterations
        // occur until a superdiagonal entry is effectively zero compared to the
        // sum of magnitudes of its diagonal neighbors. Generally, the
        // nonaggressive convergence is acceptable.
        //
        // The order of the eigenvalues is specified by sortType: -1 (decreasing),
        // 0 (no sorting), or +1 (increasing). When sorted, the eigenvectors are
        // ordered accordingly, and (evec[0], evec[1], evec[2]) is guaranteed to
        // be a right-handed orthonormal set. The return value is the number of
        // iterations used by the algorithm.
        int operator() (Real a00, Real a01, Real a02, Real a11, Real a12, Real a22,
                        bool aggressive, int sortType, std::array<Real, 3>& eval,
                        std::array<std::array<Real, 3>, 3>& evec) const;

    private:
        // Update Q = Q*G in-place using G = {{c,0,-s},{s,0,c},{0,0,1}}.
        void Update0(Real Q[3][3], Real c, Real s) const;
        // Update Q = Q*G in-place using G = {{0,1,0},{c,0,s},{-s,0,c}}.
        void Update1(Real Q[3][3], Real c, Real s) const;
        // Update Q = Q*H in-place using H = {{c,s,0},{s,-c,0},{0,0,1}}.
        void Update2(Real Q[3][3], Real c, Real s) const;
        // Update Q = Q*H in-place using H = {{1,0,0},{0,c,s},{0,s,-c}}.
        void Update3(Real Q[3][3], Real c, Real s) const;

```

5
// Normalize (u,v) robustly, avoiding floating-point overflow in the sqrt
// call. The normalized pair is (cs,sn) with cs <= 0. If (u,v) = (0,0),
// the function returns (cs,sn) = (-1.0). When used to generate a
// Householder reflection, it does not matter whether (cs,sn) or (-cs,-sn)
// is used. When generating a Givens reflection, cs = cos(2*theta) and
// sn = sin(2*theta). Having a negative cosine for the double-angle
// term ensures that the single-angle terms c = cos(theta) and
// s = sin(theta) satisfy |c| <= |s|

void GetCosSin<Real u, Real v, Real& cs, Real& sn> const;

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

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

and the implementation is

template<typename Real>
int SymmetricEigenSolve3x3<Real>::operator() (Real a00, Real a01,
Real a02, Real a11, Real a12, Real a22, bool aggressive, int sortType,
std::array<Real, 3>& eval, std::array<std::array<Real, 3>, 3>& evec) const
{
    // Compute the Householder reflection H and B = H\cdot A\cdot H, where b02 = 0.
    Real const zero = (Real)0, one = (Real)1, half = (Real)0.5;
    bool isRotation = false;
    Real c, s;
    GetCosSin(a12, -a02, c, s);
    Real Q[3][3] = { { c, s, zero }, { s, -c, zero }, { zero, zero, one } };
    Real term0 = c * a00 + s * a01;
    Real term1 = c * a01 + s * a11;
    Real b00 = c * term0 + s * term1;
    Real b01 = s * term0 - c * term1;
    term0 = s * a00 - c * a01;
    term1 = s * a01 - c * a11;
    Real b11 = s * term0 - c * term1;
    Real b12 = s * a02 - c * a12;
    Real b22 = a22;

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

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

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

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

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

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

    // Update D = Q^T*B*Q,
    saveB00 = b00;
    saveB01 = b01;
    saveB11 = b11;
    term0 = c * saveB00 + s * saveB01;
    term1 = c * saveB01 + s * saveB11;
    b00 = c * term0 + s * term1;
    b11 = b22;
    term0 = c * saveB01 - s * saveB00;
    term1 = c * saveB11 - s * saveB01;
    b22 = c * term1 - s * term0;
    b01 = s * b12;
    b12 = c * b12;

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

        // Update Q by the Householder reflection.
        Update2(Q, c, s);
        isRotation = !isRotation;
    }
}
else
{
    Real saveB11, saveB12, saveB22;
    for (iteration = 0; iteration < maxIteration; ++iteration)
    {
        // Compute the Givens reflection.
        GetCosSin(half * (b22 - b11), b12, c2, s2);
        s = sqrt(half * (one - c2)); c = half * s2 / s;

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

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

        if (Converged(aggressive, b11, b22, b12))
        {
            // Compute the Householder reflection.
            GetCosSin(half * (b11 - b22), b12, c2, s2);
            s = sqrt(half * (one - c2));
        }
    }
}
\[ c = \text{half} \times s_2 / s; \quad \text{// } >= 1/\sqrt{2} \]

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

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

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

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

// Ensure the columns of Q form a right-handed set.
if (!isRotation)
{
  for (int j = 0; j < 3; ++j)
  {
    evec[2][j] = -evec[2][j];
  }
}
return iteration;
template <typename Real>
void SymmetricEigensolver3x3<Real>::Update0(Real Q[3][3], Real c, Real s) const
{
    for (int r = 0; r < 3; ++r)
    {
        Real tmp0 = c * Q[r][0] + s * Q[r][1];
        Real tmp1 = Q[r][2];
        Real tmp2 = c * Q[r][1] - s * Q[r][0];
        Q[r][0] = tmp0;
        Q[r][1] = tmp1;
        Q[r][2] = tmp2;
    }
}

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

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

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

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

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

template <typename Real>
bool SymmetricEigenSolver3x3<Real>::Converged(bool aggressive, Real bDiag0, Real bDiag1, Real bSuper) const
{
    if (aggressive)
    {
        return bSuper == (Real)0;
    }
    else
    {
        Real sum = std::abs(bDiag0) + std::abs(bDiag1);
        return sum + std::abs(bSuper) == sum;
    }
}

template <typename Real>
bool SymmetricEigenSolver3x3<Real>::Sort(std::array<Real, 3> const& d, int& i0, int& i1, int& i2) const
{
    bool odd;
    if (d[0] < d[1])
    {
        if (d[2] < d[0])
        {
            i0 = 2; i1 = 0; i2 = 1; odd = true;
        }
        else if (d[2] < d[1])
        {
            i0 = 2; i1 = 1; i2 = 0; odd = false;
        }
        else
        {
            i0 = 0; i1 = 1; i2 = 2; odd = true;
        }
    }
    else
    {
        if (d[2] < d[1])
        {
            i0 = 2; i1 = 1; i2 = 0; odd = false;
        }
        else if (d[2] < d[0])
        {
            i0 = 1; i1 = 2; i2 = 0; odd = true;
        }
        else
        {
            i0 = 1; i1 = 0; i2 = 2; odd = false;
        }
    }
    return odd;
}
```