# Eigensystem Solvers for Symmetric Matrices

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


Created: March 2, 1999
Last Modified: March 2, 2008

## Contents

This document discusses numerical solution of eigensystems of the form $A\mathbf{v} = \lambda\mathbf{v}$ where $A$ is a real, symmetric matrix. I also consider generalized eigensystems of the form $A\mathbf{v} = \lambda B\mathbf{v}$ where $B$ is nonegative definite. For regular eigensystems, standard packages may be used. In particular, for general dimensions I use routines from Numerical Recipes in C. I use the routine `tred2` for reduction of the matrix to tridiagonal form, followed by `tqli` for computing eigenstuff for a tridiagonal matrix. However, for dimensions $n = 2, 3, 4$ I have written special reduction routines which run about $n$ times faster than `tred2`. The code is found in `eigen.h` and `eigen.c`.

# 1 Householder Reduction to Tridiagonal Form

The standard way to solve $A\mathbf{v} = \lambda\mathbf{v}$ is to compute an orthogonal matrix $Q$ such that $T = Q^{\mathrm{T}}AQ$ is tridiagonal. The equivalent eigensystem is $T\mathbf{w} = \lambda\mathbf{w}$ where $\mathbf{w} = Q^{\mathrm{T}}\mathbf{v}$. Such systems are fairly easy to solve numerically since roots of $\det(T - \lambda I) = 0$ can be bounded *a priori* and then located using standard root-finding techniques.

The reduction $T = Q^{\mathrm{T}}AQ$ is obtained where $Q$ is a product of a sequence of $n - 2$ Householder transformations, where $n$ is the size of the matrix. Since $A$ is symmetric, the matrix $T$ is symmetric, so only the main diagonal and the adjacent subdiagonal need to be computed. A description of the algorithm is given in Numerical Recipes in C. The routine which implements the reduction is

```
void mgcEigen::
TridiagonalN (int n, float** mat, float* diag, float* subd);

// input:   n = size of matrix
//          mat = nxn real, symmetric A
// output:  mat = orthogonal Q
//          diag = diagonal entries of tridiagonal T, diag[0..n-1]
//          subd = subdiagonal entries of T, subd[0..n-2]
```

I modified the Numerical recipes code so that array indexing starts at 0 rather than at 1. I also have the first $n - 1$ entries of `subd` storing the subdiagonal values rather than the last $n - 1$ (`subd[1..n-1]`) as in NRC. In the book, routine `tred2` returns the subdiagonal entries in the last $n - 1$ positions. The code `tqli` for finding eigenstuff of tridiagonal matrices rotates the input subdiagonal entries so that the first $n - 1$ entries are valid, and the last entry is arbitrary. I placed the rotation in `tred2` because I have my own reduction code which can be called in place of `tred2`, so there would be no reason to waste time rotating in `tqli`.

The computation of eigenstuff for tridiagonal $T$ is accomplished by factoring $T = QL$ where $Q$ is orthogonal and $L$ is lower triangular. The diagonal entries of $L$ are the eigenvalues of $T$ and the columns of $Q$ are the corresponding eigenvectors. The routine below is found in Numerical Recipes in C and uses implicit shifting to accelerate the convergence and avoid loss of precision of the eigenvalues.

```
void mgcEigen::
QLAlgorithm (int n, float* diag, float* subd, float** mat);

// input:   n = size of matrix
//          diag = diagonal entries of tridiagonal T, diag[0..n-1]
//          subd = subdiagonal entries of T, subd[0..n-2]
```

```
//          mat = the output matrix from NRC_tred2 if the eigenstuff
//                of real, symmetric A is desired (run NRC_tred2 on A
//                first), otherwise just the identity matrix if
//                eigenstuff desired for T only
// output:  diag = eigenvalues of T (and of A if NRC_tred2 used first)
//          mat = eigenvectors of T (or of A if NRC_tred2 used first),
//                column k of mat is an eigenvector for the eigenvalue
//                diag[k]
```

As before, all array indexing has been modified to start at 0 rather than at 1.

For my applications I need the eigenvalues to be sorted. The following routines are selection sorts. The order of the parameters is different than in NRC.

```
void mgcEigen::
DecreasingSort (int n, float* eigval, float** eigvec);

// input:   eigval = eigenvalues in arbitrary order
//          eigvec = column k of matrix is eigenvector for eigval[k]
// output:  eigval = eigenvalues in decreasing order, maximum in
//                   eigval[0], minimum in eigval[n-1]
//          eigvec = column k of matrix is eigenvector for eigval[k]


void mgcEigen::
IncreasingSort (int n, float* eigval, float** eigvec);

// input:   eigval = eigenvalues in arbitrary order
//          eigvec = column k of matrix is eigenvector for eigval[k]
// output:  eigval = eigenvalues in increasing order, maximum in
//                   eigval[0], minimum in eigval[n-1]
//          eigvec = column k of matrix is eigenvector for eigval[k]
```

Array indexing also starts at 0 rather than at 1.

## 2 Symbolic Tridiagonalization

For small dimensions, the Householder reductions for tridiagonalization can be done symbolically. The resulting implementations are faster than the general purpose code provided in Numerical Recipes. The routines are `TridiagonalD` for $D = 2, 3, 4$. I did profiling by iterating $100,000$ times a block of code which assigned a matrix its values, tridiagonalized (using `TridiagonalN` or `TridiagonalD`), and computed eigenstuff with `QLAlgorithm`. I did the timing on an Intel 80486 at 33 MHz running under Microsoft Windows 3.1. The table below gives the timing information where the times are total seconds of execution time as measured by using the `clock()` routine in `time.h`. The reduction routines were timed on matrices of the form $M = [m_{ij}]$ where $m_{ij} = i + j + 1$.

| D | time for `TridiagonalN` | time for `TridiagonalD` |
|---|---|---|
| 2 | 9.23 | 5.99 |
| 3 | 43.52 | 15.33 |
| 4 | 99.12 | 27.98 |

Timing for numeric versus symbolic tridiagonalization

## 2.1 Reduction of $2 \times 2$ matrices

Since a $2 \times 2$ matrix is already tridiagonal, there is nothing to do theoretically. However, the implementation must do the assignment of numbers to the actual parameters needed for further processing. The method which does this is

```
void mgcEigen::
Tridiagonal2 (float** mat, float* diag, float* subd)

// input:   mat = 2x2 real, symmetric A
// output:  mat = identity matrix I
//          diag = diagonal entries of A, diag[0] = a00, diag[1] = a11
//          subd = subdiagonal entry of A, subd[0] = a01

{
    // matrix is already tridiagonal

    diag[0] = mat[0][0];
    diag[1] = mat[1][1];
    subd[0] = mat[0][1];
    subd[1] = 0;
    mat[0][0] = 1;  mat[0][1] = 0;
    mat[1][0] = 0;  mat[1][1] = 1;
}
```

## 2.2 Reduction of $3 \times 3$ matrices

There are many ways to reduce $A$ to tridiagonal form. I used a Householder transformation which does a rotation and a reflection in the $x_1x_2$-plane, where $\mathbf{x} = (x_0, x_1, x_2)$. Let the matrix entries be labeled

$$A = \begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix}.$$

If $c = 0$, the matrix is already tridiagonal, so the orthogonal transformation is just the identity matrix $Q = I$. If $c \neq 0$, a Householder transformation $Q$ and corresponding tridiagonal matrix $T = Q^{\mathrm{T}}AQ$ are

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & u & v \\ 0 & v & -u \end{bmatrix} \quad \text{and} \quad T = \begin{bmatrix} a & L & 0 \\ L & d+vq & e-uq \\ 0 & e-uq & f-vq \end{bmatrix}$$

where $L = \sqrt{b^2 + c^2}$, $u = b/L$, $v = c/L$, and $q = 2ue + v(f - d)$. The method for the reduction is

```
void mgcEigen::
Tridiagonal3 (float** mat, float* diag, float* subd)

// input:   mat = 3x3 real, symmetric A
// output:  mat = orthogonal matrix Q
//          diag = diagonal entries of T, diag[0,1,2]
//          subd = subdiagonal entry of T, subd[0,1]

{
    float a = mat[0][0], b = mat[0][1], c = mat[0][2],
                         d = mat[1][1], e = mat[1][2],
                                        f = mat[2][2];

    diag[0] = a;
    subd[2] = 0;
    if ( c != 0 ) {
        float ell = sqrt(b*b+c*c);
        b /= ell;
        c /= ell;
        float q = 2*b*e+c*(f-d);
        diag[1] = d+c*q;
        diag[2] = f-c*q;
        subd[0] = ell;
        subd[1] = e-b*q;
        mat[0][0] = 1; mat[0][1] = 0; mat[0][2] = 0;
        mat[1][0] = 0; mat[1][1] = b; mat[1][2] = c;
        mat[2][0] = 0; mat[2][1] = c; mat[2][2] = -b;
    }
    else {
        diag[1] = d;
        diag[2] = f;
        subd[0] = b;
        subd[1] = e;
        mat[0][0] = 1; mat[0][1] = 0; mat[0][2] = 0;
        mat[1][0] = 0; mat[1][1] = 1; mat[1][2] = 0;
        mat[2][0] = 0; mat[2][1] = 0; mat[2][2] = 1;
    }
}
```

## 2.3 Reduction of $4 \times 4$ matrices

The reduction of a $4 \times 4$ matrix is more complicated. The orthogonal transformation is a motion in the $x_1x_2x_3$-plane where the general 4-dimensional coordinates are $\mathbf{x} = (x_0, x_1, x_2, x_3)$. Let the matrix entries be

labeled

$$A = \begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix} = \left[\begin{array}{c|c} a & \mathbf{v}^{\mathrm{T}} \\ \hline \mathbf{v} & S \end{array}\right]$$

where the right-hand side is in block matrix form where $\mathbf{v}$ is a $3 \times 1$ column vector and $S$ is a symmetric $3 \times 3$ submatrix of $A$. I seek an orthogonal transformation in block form $Q = \mathrm{Diag}(1, P)$ where 1 is a scalar, $P$ is a $3 \times 3$ orthogonal matrix, and $T = Q^{\mathrm{T}}AQ$ is tridiagonal. The tridiagonal matrix $T$ is then given by

$$T = Q^{\mathrm{T}}AQ = \left[\begin{array}{c|c} a & \mathbf{v}^{\mathrm{T}}P \\ \hline P^{\mathrm{T}}\mathbf{v} & P^{\mathrm{T}}SP \end{array}\right].$$

Let the columns of $P$ be labeled $\mathbf{p}_k$, $k = 1, 2, 3$. For $T$ to be tridiagonal, $\mathbf{v}^{\mathrm{T}}P$ must be parallel to $(1, 0, 0)$. This implies that $\mathbf{v}$ is orthogonal to both $\mathbf{p}_2$ and $\mathbf{p}_3$. If $\mathbf{v} = (0, 0, 0)$, choose $\mathbf{p}_1 = (1, 0, 0)$; otherwise, set $\mathbf{p}_1 = \mathbf{v}/|\mathbf{v}|$.

Also for $T$ to be tridiagonal, the upper right-hand corner entry of $P^{\mathrm{T}}SP$ must be 0; that is, $\mathbf{p}_1^{\mathrm{T}}S\mathbf{p}_3 = 0$. This implies that the vector $S\mathbf{p}_1$ is in the plane of $\mathbf{p}_1$ and $\mathbf{p}_2$. If $S\mathbf{p}_1$ is parallel to $\mathbf{p}_1$ (*i.e.* $\mathbf{p}_1$ is an eigenvector for $S$), then choose any $\mathbf{p}_2$ and $\mathbf{p}_3$ such that $P$ is orthogonal. If $\mathbf{p}_1 = (\alpha, \beta, \gamma)$, choose

$$P = \begin{bmatrix} \alpha & -\beta & -\gamma \\ \beta & 1 + \frac{(\alpha-1)\beta^2}{\sqrt{\beta^2+\gamma^2}} & \frac{(\alpha-1)\beta\gamma}{\sqrt{\beta^2+\gamma^2}} \\ \gamma & \frac{(\alpha-1)\beta\gamma}{\sqrt{\beta^2+\gamma^2}} & 1 + \frac{(\alpha-1)\gamma^2}{\sqrt{\beta^2+\gamma^2}} \end{bmatrix}$$

if $\sqrt{\beta^2 + \gamma^2} \neq 0$. If this square root is zero, simply choose $\mathbf{p}_2 = (0, 1, 0)$ and $\mathbf{p}_3 = (0, 0, 1)$. If $S\mathbf{p}_1$ is not parallel to $\mathbf{p}_1$, choose

$$\mathbf{p}_3 = \frac{\mathbf{p}_1 \times S\mathbf{p}_1}{|\mathbf{p}_1 \times S\mathbf{p}_1|} \quad \text{and} \quad \mathbf{p}_2 = \mathbf{p}_3 \times \mathbf{p}_1.$$

After constructing $P$, the code also includes construction of the entries for $P^{\mathrm{T}}SP$ which is itself a tridiagonal matrix, so only 5 values need be computed. The method for the reduction is

```
void mgcEigen::
Tridiagonal4 (float** mat, float* diag, float* subd)

// input:   mat = 4x4 real, symmetric A
// output:  mat = orthogonal matrix Q
//          diag = diagonal entries of T, diag[0,1,2,3]
//          subd = subdiagonal entry of T, subd[0,1,2]

{
    // save matrix M
    float a = mat[0][0], b = mat[0][1], c = mat[0][2], d = mat[0][3],
                         e = mat[1][1], f = mat[1][2], g = mat[1][3],
                                        h = mat[2][2], i = mat[2][3],
                                                       j = mat[3][3];
```

```
diag[0] = a;
subd[3] = 0;

mat[0][0] = 1; mat[0][1] = 0; mat[0][2] = 0; mat[0][3] = 0;
mat[1][0] = 0;
mat[2][0] = 0;
mat[3][0] = 0;

if ( c != 0 || d != 0 ) {
    float q11, q12, q13;
    float q21, q22, q23;
    float q31, q32, q33;

    // build column Q1
    float len = sqrt(b*b+c*c+d*d);
    q11 = b/len;
    q21 = c/len;
    q31 = d/len;

    subd[0] = len;

    // compute S*Q1
    float v0 = e*q11+f*q21+g*q31;
    float v1 = f*q11+h*q21+i*q31;
    float v2 = g*q11+i*q21+j*q31;

    diag[1] = q11*v0+q21*v1+q31*v2;

    // build column Q3 = Q1x(S*Q1)
    q13 = q21*v2-q31*v1;
    q23 = q31*v0-q11*v2;
    q33 = q11*v1-q21*v0;
    len = sqrt(q13*q13+q23*q23+q33*q33);
    if ( len > 0 ) {
        q13 /= len;
        q23 /= len;
        q33 /= len;

        // build column Q2 = Q3xQ1
        q12 = q23*q31-q33*q21;
        q22 = q33*q11-q13*q31;
        q32 = q13*q21-q23*q11;

        v0 = q12*e+q22*f+q32*g;
        v1 = q12*f+q22*h+q32*i;
        v2 = q12*g+q22*i+q32*j;
        subd[1] = q11*v0+q21*v1+q31*v2;
        diag[2] = q12*v0+q22*v1+q32*v2;
```

```
        subd[2] = q13*v0+q23*v1+q33*v2;

        v0 = q13*e+q23*f+q33*g;
        v1 = q13*f+q23*h+q33*i;
        v2 = q13*g+q23*i+q33*j;
        diag[3] = q13*v0+q23*v1+q33*v2;
    }
    else {  // S*Q1 parallel to Q1, choose any valid Q2 and Q3
        subd[1] = 0;

        len = q21*q21+q31*q31;
        if ( len > 0 ) {
            float tmp = q11-1;
            q12 = -q21;
            q22 = 1+tmp*q21*q21/len;
            q32 = tmp*q21*q31/len;

            q13 = -q31;
            q23 = q32;
            q33 = 1+tmp*q31*q31/len;

            v0 = q12*e+q22*f+q32*g;
            v1 = q12*f+q22*h+q32*i;
            v2 = q12*g+q22*i+q32*j;
            diag[2] = q12*v0+q22*v1+q32*v2;
            subd[2] = q13*v0+q23*v1+q33*v2;

            v0 = q13*e+q23*f+q33*g;
            v1 = q13*f+q23*h+q33*i;
            v2 = q13*g+q23*i+q33*j;
            diag[3] = q13*v0+q23*v1+q33*v2;
        }
        else {  // Q1 = (+-1,0,0)
            q12 = 0; q22 = 1; q32 = 0;
            q13 = 0; q23 = 0; q33 = 1;

            diag[2] = h;
            diag[3] = j;
            subd[2] = i;
        }
    }

    mat[1][1] = q11; mat[1][2] = q12; mat[1][3] = q13;
    mat[2][1] = q21; mat[2][2] = q22; mat[2][3] = q23;
    mat[3][1] = q31; mat[3][2] = q32; mat[3][3] = q33;
}
else {
    diag[1] = e;
```

```
            subd[0] = b;
            mat[1][1] = 1;
            mat[2][1] = 0;
            mat[3][1] = 0;

            if ( g != 0 ) {
                float ell = sqrt(f*f+g*g);
                f /= ell;
                g /= ell;
                float Q = 2*f*i+g*(j-h);

                diag[2] = h+g*Q;
                diag[3] = j-g*Q;
                subd[1] = ell;
                subd[2] = i-f*Q;
                mat[1][2] = 0;  mat[1][3] = 0;
                mat[2][2] = f;  mat[2][3] = g;
                mat[3][2] = g;  mat[3][3] = -f;
            }
            else {
                diag[2] = h;
                diag[3] = j;
                subd[1] = f;
                subd[2] = i;
                mat[1][2] = 0;  mat[1][3] = 0;
                mat[2][2] = 1;  mat[2][3] = 0;
                mat[3][2] = 0;  mat[3][3] = 1;
            }
        }
    }
```

# 3 Implementation

The class declaration for handling eigensystems is

```
#include <iostream.h>

class mgcEigen
{
public:
    mgcEigen (int _size);
    ~mgcEigen ();

    // set the matrix for eigensolving
    float& Matrix (int row, int col) { return mat[row][col]; }
    mgcEigen& Matrix (float** inmat);
```

```
    // get the results of eigensolving
    float Eigenvalue (int d) { return diag[d]; }
    float Eigenvector (int row, int col) { return mat[row][col]; }
    const float* Eigenvalue () { return diag; }
    const float** Eigenvector () { return (const float**) mat; }

    // solve eigensystem
    void EigenStuff2 ();  // uses TriDiagonal2
    void EigenStuff3 ();  // uses TriDiagonal3
    void EigenStuff4 ();  // uses TriDiagonal4
    void EigenStuffN ();  // uses TriDiagonalN
    void EigenStuff  ();  // uses switch statement

    // solve eigensystem, use decreasing sort on eigenvalues
    void DecrSortEigenStuff2 ();
    void DecrSortEigenStuff3 ();
    void DecrSortEigenStuff4 ();
    void DecrSortEigenStuffN ();
    void DecrSortEigenStuff  ();

    // solve eigensystem, use increasing sort on eigenvalues
    void IncrSortEigenStuff2 ();
    void IncrSortEigenStuff3 ();
    void IncrSortEigenStuff4 ();
    void IncrSortEigenStuffN ();
    void IncrSortEigenStuff  ();

    // debugging output?
    float& Tdiag (int i) { return diag[i]; }
    float& Tsubdiag (int i) { return subd[i]; }
    void Reduce () { TridiagonalN(size,mat,diag,subd); }

private:
    int size;
    float** mat;
    float* diag;
    float* subd;

    // Householder reduction to tridiagonal form
    void Tridiagonal2 (float** mat, float* diag, float* subd);
    void Tridiagonal3 (float** mat, float* diag, float* subd);
    void Tridiagonal4 (float** mat, float* diag, float* subd);
    void TridiagonalN (int n, float** mat, float* diag, float* subd);

    // QL algorithm with implicit shifting, applies to tridiagonal matrices
    void QLAlgorithm (int n, float* diag, float* subd, float** mat);
```

```
    // sort eigenvalues from largest to smallest
    void DecreasingSort (int n, float* eigval, float** eigvec);

    // sort eigenvalues from smallest to largest
    void IncreasingSort (int n, float* eigval, float** eigvec);

// error handling
public:
    static int verbose;
    static unsigned error;
    static void Report (ostream& ostr);
private:
    static const unsigned invalid_size;
    static const unsigned allocation_failed;
    static const unsigned ql_exceeded;
    static const char* message[];
    static int Number (unsigned single_error);
    static void Report (unsigned single_error);
};
```

Methods are provided for solving low dimensions 2, 3, or 4 using the optimized routines. The other methods use the general Numerical Recipes in C code. The eigenvalues can be solved for in no particular order, in decreasing order, or in increasing order. An example is

```
int main ()
{
    mgcEigen eig(3);

    eig.Matrix(0,0) = 2;  eig.Matrix(0,1) = 1;  eig.Matrix(0,2) = 1;
    eig.Matrix(1,0) = 1;  eig.Matrix(1,1) = 2;  eig.Matrix(1,2) = 1;
    eig.Matrix(2,0) = 1;  eig.Matrix(2,1) = 1;  eig.Matrix(2,2) = 2;

    eig.IncrSortEigenStuff3();

    cout.setf(ios::fixed);

    cout << "eigenvalues = " << endl;
    for (int row = 0; row < 3; row++)
        cout << eig.Eigenvalue(row) << ' ';
    cout << endl;

    cout << "eigenvectors = " << endl;
    for (row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++)
            cout << eig.Eigenvector(row,col) << ' ';
        cout << endl;
    }
```

```
    // eigenvalues =
    //    1.000000 1.000000 4.000000
    // eigenvectors =
    //    0.411953  0.704955 0.577350
    //    0.404533 -0.709239 0.577350
    //   -0.816485  0.004284 0.577350

    return 0;
}
```

The characteristic polynomial is $\lambda^3 - 6\lambda^2 - 9\lambda + 4$ which has roots 1, 1, and 4. The eigenspace corresponding to $\lambda = 1$ is 2-dimensional and is spanned by $(1, -1, 0$ and $(1, 0, -1)$ (these are not an orthogonal basis for the eigenspace). The eigenspace corresponding to $\lambda = 4$ is spanned by $(1, 1, 1)$. Note that the eigensolver can return any set of orthonormal vectors in eigenspaces whose dimension is larger than 1.

# 4 Generalized Eigensystems

A generalized eigensystem is of the form $A\mathbf{v} = \lambda B\mathbf{v}$ where $B$ is also a matrix. If $B$ is invertible, then the system can be converted to a regular one: $B^{-1}A\mathbf{v} = \lambda\mathbf{v}$. However, there are some instances where inverting $B$ is not the best choice for a robust numerical solution, for example, when $B$ is positive definite. If $B$ is not invertible, then one must take advantage of any special structure that $B$ has, for example, if $B$ is nonnegative definite.

Suppose that $B$ is positive definite. The matrix can be uniquely factored as $B = Q^{\mathrm{T}}DQ$ for an orthogonal matrix $Q$ and a diagonal matrix $D$ with nonnegative entries. The diagonal matrix $D = \mathrm{Diag}d_1, \ldots, d_n$ has a square root $D^{1/2} = \mathrm{Diag}(d_1^{1/2}, \ldots, d_n^{1/2})$. Thus, $B = M^{\mathrm{T}}M$ where $M = D^{1/2}Q$. The generalized eigensystem is converted to

$$A\mathbf{v} = \lambda M^{\mathrm{T}}M\mathbf{v}.$$

Some matrix algebra yields

$$M^{-\mathrm{T}}AM^{-1}M\mathbf{v} = \lambda M\mathbf{v}.$$

Setting $C = M^{-\mathrm{T}}AM^{-1}$ and $\mathbf{w} = M\mathbf{v}$, the eigensystem is now

$$C\mathbf{w} = \lambda\mathbf{w}$$

which is now a regular eigensystem where $C$ is symmetric. The numerical methods described earlier can now be applied to solve for $\lambda$ and vectors $\mathbf{w}$. The eigenvectors for the original generalized system can be solved as $\mathbf{v} = M^{-1}\mathbf{w}$.

## 4.1 Principal Curvatures and Directions

As an example, consider the problem of computing the principal curvatures $\lambda$ and principal directions $\mathbf{v}$ for an 2-dimensional surface defined implicitly as $F(\mathbf{x}) = 0$ where $F : \mathbb{R}^3 \to \mathbb{R}$. They are determined by the generalized eigensystem $A\mathbf{v} = \lambda B\mathbf{v}$, where

$$A = \frac{D^2F}{\sqrt{1 + |DF|^2}} \text{ and } B = I + DFDF^{\mathrm{T}}$$

where $DF$ is the $3 \times 1$ vector of first derivatives of $F$, $D^2F$ is the $3 \times 3$ matrix of second derivatives of $F$, and $I$ is the $3 \times 3$ identity matrix.

Matrix $B$ can be decomposed as follows. Let $\mathbf{d} = DF$. Let $Q$ be an orthogonal matrix such that $Q\mathbf{d} = |\mathbf{d}|\mathbf{k}$, where $\mathbf{k} = (0, 0, 1)$; then $QBQ^{\mathrm{T}} = \mathrm{Diag}(1, 1, 1 + |\mathbf{d}|^2) = D$. Thus, $B = M^{\mathrm{T}}M$ where $M = D^{1/2}Q$. As shown previously, the eigensystem now becomes $C\mathbf{w} = \lambda\mathbf{w}$ where $C = M^{-\mathrm{T}}AM^{-1}$ and $\mathbf{w} = M\mathbf{v}$.

Matrix $Q$ may be chosen as follows. Let $\mathbf{d} = (d_1, \ldots, d_n)$ and define $\boldsymbol{\alpha} = (d_1, \ldots, d_{n-1})$. If $\boldsymbol{\alpha} = \mathbf{0}$, just choose $Q$ to be the identity matrix. Otherwise, define $\boldsymbol{\beta} = \boldsymbol{\alpha}/|\boldsymbol{\alpha}|$ and let

$$Q = \left[\begin{array}{c|c} E + (d_n - 1)\boldsymbol{\beta}\boldsymbol{\beta}^{\mathrm{T}} & \boldsymbol{\alpha} \\ \hline -\boldsymbol{\alpha}^{\mathrm{T}} & d_n \end{array}\right],$$

where $E$ is the $(n-1) \times (n-1)$ identity matrix.

## 4.2 Optimal Scale Ridges

Consider a smooth function $f : \mathbb{R}^2 \to \mathbb{R}$. Let $D^2f\mathbf{u} = \alpha\mathbf{u}$ and $D^2f\mathbf{v} = \beta\mathbf{v}$ where $\alpha \le \beta$ and where the eigenvectors are unit length. The *height ridges* of $f$ are those points $(x, y)$ for which $\mathbf{u} \cdot Df = 0$ and $\alpha = \mathbf{u}^{\mathrm{T}}D^2f\mathbf{u} = \alpha < 0$.

Consider a scale space function $M : \mathbb{R}^2 \times \mathbb{R}^+ \to \mathbb{R}$, say $M = M(x, y, \sigma)$. For each $(x, y)$ find the smallest $\sigma_0 = \sigma_0(x, y)$ such that $M$ has a local maximum through scale at $(x, y, \sigma_0(x, y))$. Thus, $M_\sigma(x, y, \sigma_0(x, y)) = 0$ and $M_{\sigma\sigma}(x, y, \sigma_0(x, y)) < 0$. For a connected component of the graph of $\sigma_0(x, y)$, define the function $f(x, y) = M(x, y, \sigma_0(x, y))$. The *optimal scale ridges* of $M$ are the height ridges of $f$.

The optimal scale ridge definition is equivalent to the following one. Note that

$$Df = (M_x + M_\sigma\sigma_{0,x}, M_y + M_\sigma\sigma_{0,y}) = (M_x, M_y).$$

The first equality is by application of the chain rule. The second equality is true since $M_\sigma(x, y, \sigma_0(x, y)) \equiv 0$. Differentiating again yields

$$D^2f = \left[\begin{array}{cc} M_{xx} + M_{x\sigma}\sigma_{0,x} & M_{xy} + M_{x\sigma}\sigma_{0,y} \\ M_{yx} + M_{y\sigma}\sigma_{0,x} & M_{yy} + M_{y\sigma}\sigma_{0,y} \end{array}\right].$$

Differentiating $M_\sigma(x, y, \sigma_0(x, y)) \equiv 0$ yields

$$\begin{aligned} \sigma_{0,x} &= -\tfrac{M_{x\sigma}}{M_{\sigma\sigma}} \\ \sigma_{0,y} &= -\tfrac{M_{y\sigma}}{M_{\sigma\sigma}} \\ M_{x\sigma}\sigma_{0,y} &= M_{y\sigma}\sigma_{0,x} \end{aligned}$$

Let $\mathbf{v} = (v_0, v_1)$ be an eigenvector for $D^2f$ with eigenvalue $\lambda$, so $D^2f\mathbf{v} = \lambda\mathbf{v}$. Define

$$v_2 = v_0\sigma_{0,x} + v_1\sigma_{0,y}.$$

This implies $v_0M_{x\sigma} + v_1M_{y\sigma} + v_2M_{\sigma\sigma} \equiv 0$. It can be shown that

$$D^2M \left[\begin{array}{c} \mathbf{v} \\ v_2 \end{array}\right] = \left[\begin{array}{c} \lambda\mathbf{v} \\ 0 \end{array}\right].$$

We can therefore rephrase the optimal scale ridge definition. Let $\mathbf{u} = (0, 0, 1)$ and $\alpha = M_{\sigma\sigma}$. Define matrix $P = \mathrm{Diag}(1, 1, 0)$. Let $\mathbf{v}$ and $\mathbf{w}$ solve the generalized eigensystem

$$D^2 M\mathbf{v} = \beta P\mathbf{v}, \quad D^2 M\mathbf{w} = \gamma P\mathbf{w}.$$

Ridge curves are obtained as solutions to $\mathbf{u} \cdot DM = 0$ and $\mathbf{v} \cdot DM = 0$ where $\alpha < 0$ and $\beta < 0$.

In the generalized eigensystem, the matrix $P$ is nonnegative definite. The factoring method that worked for positive definite matrices does not apply here. The problem is that $P = M^{\mathrm{T}}M$, but $M$ is not invertible. However, for this special problem, you can find the eigenvalues in the standard way, $\det(D^2 M - \lambda P) = 0$. Despite the fact that the matrices are $3 \times 3$, the characteristic polynomial is quadratic.

Generally, if $A$ and $B$ are real symmetric $n \times n$ matrices, and if $B$ has rank $r < n$ ($B$ is singular), the number of generalized eigenvalues of $A\mathbf{v} = \lambda B\mathbf{v}$ is $r$.