Supplemental Information
for
Spin Transformations of Discrete Surfaces

Keenan Crane, Ulrich Pinkall, and Peter Schröder

ACM Transactions on Graphics (SIGGRAPH 2011)

1 Performance

The main computational cost when computing conformal deformations is solving the eigenvalue problem

$$(D - \rho)\lambda = \gamma\lambda$$

for the smallest eigenvalue γ and corresponding eigenvector λ. In practice, however, **a good solution can often be found by solving a single linear system**.

More specifically, let $\mathsf{X} = \mathsf{A}^\star\mathsf{A}$ be the matrix used in the standard eigenvalue problem. This matrix is symmetric and positive-semidefinite with only about 7 nonzeros in each row/column, which means it can be efficiently inverted using standard methods like Cholesky factorization or the conjugate gradient method. A simple algorithm for computing the smallest eigenvector is then:

Algorithm 1 The Inverse Power Method

```
Require: Initial guess $\lambda_0$.
1: for $i = 1, \ldots, k$ do
2:     Solve $\mathsf{X}\lambda_i = \lambda_{i-1}$
3:     $\lambda_i \leftarrow \lambda_i / ||\lambda_i||$
4: end for
```

Starting from a random initial guess this procedure tends to produce good results after only a few iterations (see Figure 1). A more intelligent initial guess sets $\lambda = 1$ at each vertex, which corresponds to the identity transformation. In this case we often get *very* close to the solution after only a single iteration, requiring only a single linear solve (see Figure 2).

The figures below depict two tests: `bumpy`, where we add random bumps to a sphere, and `moon`, where we paint a face on a disk. On a 2.4 GHz Core 2 Duo laptop, a single iteration (using `mldivide` in MATLAB) takes 0.3 seconds on a mesh with 8k faces (`bumpy`) and 1.26 seconds on a mesh with 33k faces (`moon`).

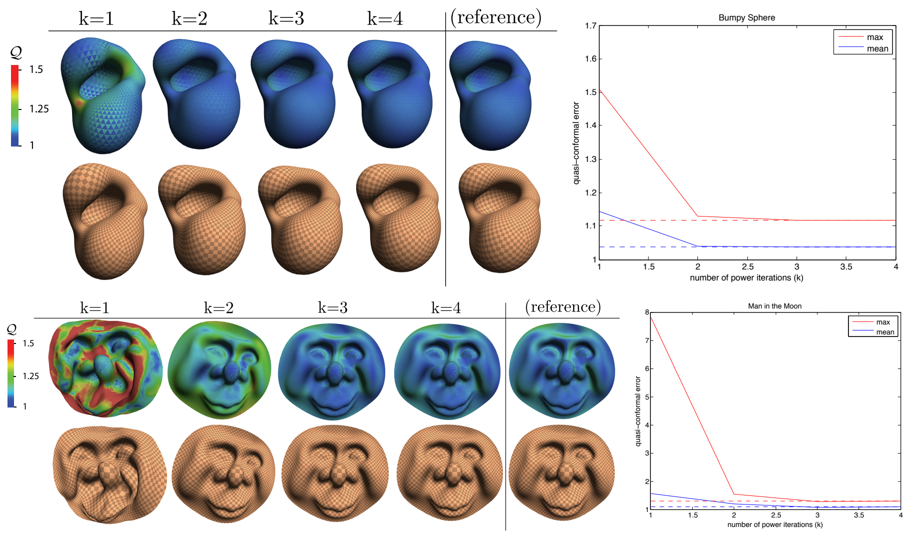

Figure 1: *Left:* solutions for `bumpy` and `moon` found after k iterations of the inverse power method. *Right:* quasi-conformal distortion as a function of iteration count. A random initial guess was used in both tests.

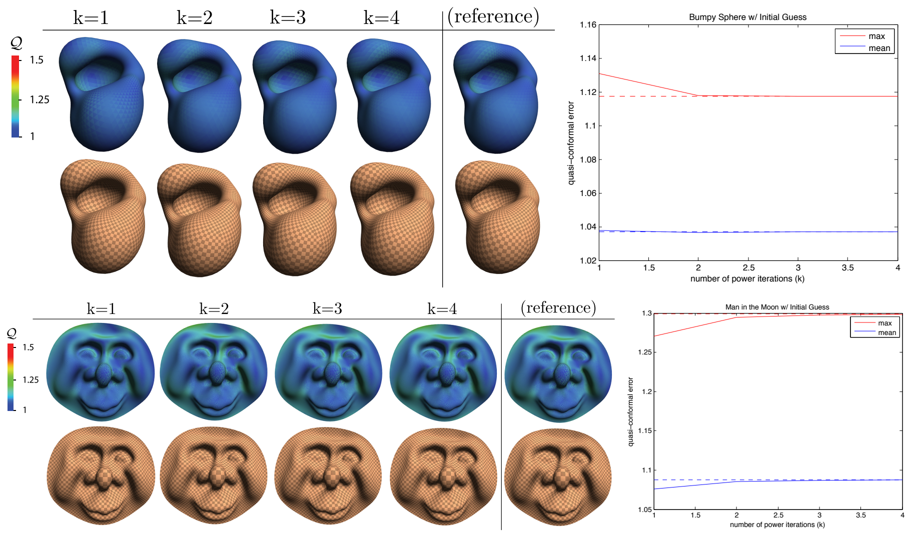
k=1
k=2
k=3
k=4
(reference)
Q
1.5
1.25
1
Bumpy Sphere w/ Initial Guess
max
mean
quasi−conformal error
number of power iterations (k)
Man in the Moon w/ Initial Guess

Figure 2: The same test depicted in Figure 1 but using a constant initial guess. Note that after the first iteration we obtain a solution that is virtually indistinguishible from the reference solution.

2 Further Performance Enhancements

Suppose a good initial guess λ_0 is known (e.g., $\lambda_0 = 1$). Then the Rayleigh quotient

$$\frac{\lambda_0^H \mathsf{X} \lambda_0}{\lambda_0^H \lambda_0}$$

should provide a good estimate γ_0 of the smallest eigenvalue. One can therefore improve the rate of convergence of the inverse power method by using the shifted matrix $\mathsf{X}' = \mathsf{X} - \gamma_0 \mathsf{I}$ in place of the usual matrix X (where I is the identity).

3 Building the Eigenvalue System

In practice it may be convenient to build the matrix X directly (rather than building it up from constituent matrices). The following algorithm provides a simple *facewise* construction of X:

Algorithm 2 Facewise Construction of Eigenvalue System

```
1: for $k = 1, \ldots, |F|$ do
2:     $a \leftarrow -\frac{1}{4\mathcal{A}}$
3:     $b \leftarrow \rho/6$
4:     $c \leftarrow \rho^2 \mathcal{A}/9$
5:     for all $(i, j) \in \{1, 2, 3\} \times \{1, 2, 3\}$ do
6:         $\mathsf{X}_{ij} += a e_i e_j + b(e_j - e_i) + c$
7:     end for
8: end for
```

Here k is the index of the current face, $\mathcal{A}$ is its area, and ρ is the desired change in mean curvature half-density. The inner loop visits *all* ordered pairs of edges e_1, e_2, e_3 of the current face. In C++, the algorithm might look something like Listing 1.

Listing 1: Facewise Construction of Eigenvalue System in C++

```
void buildEigenvalueProblem( const vector<Face>& faces,
                             const vector<Vertex>& vertices,
                             QuaternionSparseMatrix& E )
{
   // allocate a sparse |V|x|V| matrix
   int nV = vertices.size();
   E.resize( nV, nV );

   // visit each face
   for( size_t k = 0; k < faces.size(); k++ )
   {
      double   A = face[k].area();
      double rho = face[k].rho;

      // compute coefficients
      double a = -1. / (4.*A);
      double b = rho / 6.;
      double c = A*rho*rho / 9.;

      // get vertex indices
      int I[3] =
      {
         faces[k].vertex[0],
         faces[k].vertex[1],
         faces[k].vertex[2]
      };

      // compute edges across from each vertex
      Quaternion e[3];
      for( int i = 0; i < 3; i++ )
      {
         e[i] = vertices[ I[ (i+2) % 3 ]] -
                vertices[ I[ (i+1) % 3 ]] ;
      }

      // increment matrix entry for each ordered pair of vertices
      for( int i = 0; i < 3; i++ )
      for( int j = 0; j < 3; j++ )
      {
         E(I[i],I[j]) += a*e[i]*e[j] + b*(e[j]-e[i]) + c;
      }
   }
}
```