Mesh Differential Geometry

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

Created: May 8, 2005
Last Modified: August 12, 2023

Contents

1 Introduction

This document describes how to estimate various differential geometric quantites at the vertices of a triangle mesh. The mesh must be an oriented 2-dimensional manifold; that is, each edge is shared by one or two triangles and the mesh is oriented so that a consistent set of triangle normal vectors can be generated.

1.1 Basic Mesh Concepts

For a closed mesh with the topology of a sphere, space is partitioned into an *inside* region that has finite volume and an *outside* region that is unbounded. Triangle normals can all be chosen to point to the outside region or to the inside region; the choice depends on the needs of your application. Typically, the region to which the normals point is referred to as the *positive* side of the mesh and the other side is referred to as the *negative* side of the mesh.

For an open mesh with the topology of a rectangle, space is not partitioned by the mesh; however, a triangle locally partitions space into two regions. You may label the region locally on one side of the triangles the *positive* side of the mesh and the region on the other side of the triangles the *negative* side of the mesh. The convention is that the side of the mesh to which the normals point is called the positive side. Naturally, you want consistency in the sense that if a triangle's normal vector points to the positive side, then normal vectors at the adjacent triangles also point to the positive side.

If you have a set of normal vectors that are not guaranteed to point consistently to the positive side of the oriented mesh, the set can be modified so that the normals do point to the positive side. Considering the mesh as a graph whose nodes are the triangles and whose arcs are the edges of the triangles, a depth-first traversal of the graph may be used to detect an inconsistent normal vector and change sign on it so that it is consistent with the normal vectors of the neighboring triangles.

The concept is sometimes discussed in terms of *winding order* of the triangles. An observer on the positive side of the mesh who looks at the mesh sees triangles whose vertices are ordered either clockwise or counterclockwise. If all triangles are viewed as having clockwise winding order or all are viewed as having counterclockwise winding order, the triangle normal vectors are consistent. Assuming the convention that the triangle normals point to the positive side of the mesh, the winding order is counterclockwise. The observer sees a triangle with vertices $\boldsymbol{V}_0$, $\boldsymbol{V}_1$, and $\boldsymbol{V}_2$ listed in counterclockwise order. The edges sharing the first vertex have directions $\boldsymbol{E}_1 = \boldsymbol{V}_1 - \boldsymbol{V}_0$ and $\boldsymbol{E}_2 = \boldsymbol{V}_2 - \boldsymbol{V}_0$. A normal vector pointing to the positive side of the mesh is $\boldsymbol{N} = \boldsymbol{E}_1 \times \boldsymbol{E}_2$. Typically one wants unit-length normal vectors, $\boldsymbol{N} = (\boldsymbol{E}_1 \times \boldsymbol{E}_2)/|\boldsymbol{E}_1 \times \boldsymbol{E}_2|$.

The word *vertex* has different definitions depending on context. Sometimes the term refers solely to the position of a point. This is the case in the previous paragraph where I mentioned triangle vertices. In other contexts, the term refers to a collection of attributes of a point of which one attribute is the position. Generally, we can keep track of attributes that are related to a mesh of vertices, say, *vertex positions*, *vertex normals*, *vertex tangents*, *vertex texture coordinates*, *vertex colors*, and so on. I use the more general definition in this document.

1.2 Differential Geometric Quantities for Parameteric Surfaces

In this document, a triangle mesh is assumed to be an approximation to a parametric surface $\boldsymbol{P}(s,t)$ that has continuous second-order partial derivatives. The mesh vertices have positions that are assumed to be n

samples from this surface: $\boldsymbol{P}_i = \boldsymbol{P}(s_i, t_i)$ for $0 \le i < n$. If the parameters (s_i, t_i) are known, I will refer to the mesh as a *parametric mesh*. Each vertex is listed as a tuple of attributes, say, $\boldsymbol{V}_i = (\boldsymbol{P}_i, s_i, t_i)$.

In applications where the mesh vertices have been assigned texture coordinates, either automatically or by an artist, we do not usually know explicitly the parameteric domain of $\boldsymbol{P}(s,t)$. In a sense the texture coordinates themselves may be thought of as a parametrization. I will refer to the mesh as a *textured mesh*, although in this document it is irrelevant what colored texture is used—we care only about the texture coordinate assignment. If the texture coordinates at position $\boldsymbol{P}_i$ are (u_i, v_i), the vertex is listed as a tuple $\boldsymbol{V}_i = (\boldsymbol{P}_i, u_i, v_i)$. In the estimates discussed later in the document, I refer to parameters s and t, but the same algorithms apply if you use texture coordinates u and v.

An application might have meshes that are parametric and textured. A vertex is listed as a tuple $\boldsymbol{V}_i = (\boldsymbol{P}_i, s_i, t_i, u_i, v_i)$. For example, an artist might create a NURBS surface in a modeling package and assign texture coordinates so that a texture applied to the surface has minimal distortion. The parameters s and t themselves might not serve as suitable texture coordinates because they lead to significant distortion of an applied texture.

If the mesh vertices have no known parameters or texture coordinates, then I will refer to the mesh as a *nonparametric mesh*. The vertex consists solely of a single attribute listed as the 1-tuple $\boldsymbol{V}_i = (\boldsymbol{P}_i)$.

In this document I provide algorithms for estimating various differential geometric quantities that are typically associated with a parametric surface $\boldsymbol{P}(s,t)$ that has continuous second-order partial derivatives. The parametric formulation defines the position $\boldsymbol{P}(s,t)$. The first-order partial derivatives are vectors in the tangent space at the position; these are $\boldsymbol{P}_s = \partial\boldsymbol{P}/\partial s$ and $\boldsymbol{P}_t = \partial\boldsymbol{P}/\partial t$. If the first-order partial derivatives are linearly independent vectors, then a unit-length normal to the surface is the vector $\boldsymbol{N}(s,t) = \boldsymbol{P}_s \times \boldsymbol{P}_t / |\boldsymbol{P}_s \times \boldsymbol{P}_t|$. If we have estimates for $\boldsymbol{P}_s$ and $\boldsymbol{P}_t$, then we automatically have an estimate for $\boldsymbol{N}$.

The principal curvatures for a parametric surface $\boldsymbol{P}(s,t)$ with normal vector field $\boldsymbol{N}(s,t) = \boldsymbol{P}_s \times \boldsymbol{P}_t / |\boldsymbol{P}_s \times \boldsymbol{P}_t|$ are the generalized eigenvalues κ for the linear system $B\boldsymbol{Y} = \kappa G\boldsymbol{Y}$, where the *metric tensor* is

$$G = J^{\mathsf{T}} J = \begin{bmatrix} \boldsymbol{P}_s^{\mathsf{T}} \\ \boldsymbol{P}_t^{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \boldsymbol{P}_s & \boldsymbol{P}_t \end{bmatrix} = \begin{bmatrix} \boldsymbol{P}_s \cdot \boldsymbol{P}_s & \boldsymbol{P}_s \cdot \boldsymbol{P}_t \\ \boldsymbol{P}_t \cdot \boldsymbol{P}_s & \boldsymbol{P}_t \cdot \boldsymbol{P}_t \end{bmatrix} \tag{1}$$

and J is the 3×2 matrix whose columns are the first-order derivatives of position, and where the *curvature tensor* is

$$B = \begin{bmatrix} -\boldsymbol{N}_s \cdot \boldsymbol{P}_s & -\boldsymbol{N}_s \cdot \boldsymbol{P}_t \\ -\boldsymbol{N}_t \cdot \boldsymbol{P}_s & -\boldsymbol{N}_t \cdot \boldsymbol{P}_t \end{bmatrix} = \begin{bmatrix} \boldsymbol{N} \cdot \boldsymbol{P}_{ss} & \boldsymbol{N} \cdot \boldsymbol{P}_{st} \\ \boldsymbol{N} \cdot \boldsymbol{P}_{ts} & \boldsymbol{N} \cdot \boldsymbol{P}_{tt} \end{bmatrix} \tag{2}$$

The quantities $\boldsymbol{N}_s(s,t)$ and $\boldsymbol{N}_t(s,t)$ are the first-order partial derivatives of the normal vector $\boldsymbol{N}$. The last equality of equation (2) is a consequence of orthogonality of the normal $\boldsymbol{N}$ with tangent vectors $\boldsymbol{P}_s$ and $\boldsymbol{P}_t$; that is, we know $\boldsymbol{N} \cdot \boldsymbol{P}_s = 0$ and $\boldsymbol{N} \cdot \boldsymbol{P}_t = 0$. Computing first-order derivatives of these equations leads to the aforementioned equality of matrices. Assuming continuity of second-order derivatives, it is guaranteed that the mixed partial derivatives are the same, $\boldsymbol{P}_{ts} = \boldsymbol{P}_{st}$.

The generalized eigenvectors $\boldsymbol{Y}$ are 2×1 vectors whose elements are the coordinates for the principal directions for the surface. The principal directions themselves are the 3×1 columns of $J\boldsymbol{Y}$. As linear combinations of the derivatives, the principal directions are tangent vectors to the surface.

To estimate the principal curvatures and principal directions we need estimates for $\boldsymbol{P}_s$ and $\boldsymbol{P}_t$, which as noted automatically lead to an estimate for $\boldsymbol{N}$. Additionally, we need estimates either for the first-order

derivatives of the normal vector; $\boldsymbol{N}_s$ and $\boldsymbol{N}_t$; or for the second-order derivatives of the position vector; $\boldsymbol{P}_{ss}$, $\boldsymbol{P}_{st}$, and $\boldsymbol{P}_{tt}$.

1.3 Mesh Data Structures

The mesh is assumed to be stored as a vertex array and a triangle index array. The indices are into the vertex array. Each triple of indices corresponds to a triangle whose vertices are counterclockwise ordered when viewed from the positive side of the mesh; see listing 1. The vertex attributes include position as well as differential geometric quantities for which we provide estimates.

Listing 1. Data structures to represent a triangle mesh.

```
struct Vertex
{
    // This attribute is required.
    Point3 position;

    // One or more of these attributes are included as needed by your application.
    Vector3 normal;
    Vector3 dpds, dpdt;
    float s, t;
};

struct Triangle
{
    // Indices into the vertex array.
    int v[3];
};

struct Mesh
{
    int numVertices;
    Vertex vertex[numVertices];
    int numTriangles;
    Triangle triangle[numTriangles];
};
```

---

2 Derivative Estimates for Rectangular Lattice Topology

If the mesh topology is known to be a rectangular lattice with sampled positions $\{\boldsymbol{P}_{i,j}\}$ for $0 \le i < n$ and $0 \le j < m$, the texture coordinates are implied by the topology and may be chosen to be $(s_i, t_j) = (i/n, j/m)$; they do not need to be stored explicitly in the Vertex data structure. The sampled positions are considered to be $\boldsymbol{P}_{i,j} = \boldsymbol{P}(s_i, t_j)$. Derivative estimates at a vertex position require information from neighboring vertex positions, which means we need special-case handling of estimates at the boundary vertices where $i = 0$, $i = n-1$, $j = 0$, and $j = m-1$.

In the case of no wrap-around at the boundaries, the derivative estimates can be one-sided at the boundary vertices. From calculus, the first-order derivatives are

$$\boldsymbol{P}_s(s_i, t_j) = \lim_{\Delta_s \to 0} \frac{\boldsymbol{P}(s_i + \Delta_s, t_j) - \boldsymbol{P}(s_i, t_j)}{\Delta_s} = \lim_{\Delta_s \to 0} \frac{\boldsymbol{P}(s_i + \Delta_s, t_j) - \boldsymbol{P}(s_i - \Delta_s, t_j)}{2\Delta_s} \tag{3}$$

and

$$\boldsymbol{P}_t(s_i, t_j) = \lim_{\Delta_t \to 0} \frac{\boldsymbol{P}(s_i, t_j + \Delta_t) - \boldsymbol{P}(s_i, t_j)}{\Delta_t} = \lim_{\Delta_t \to 0} \frac{\boldsymbol{P}(s_i, t_j + \Delta_t) - \boldsymbol{P}(s_i, t_j - \Delta_t)}{2\Delta_t} \tag{4}$$

At an interior vertex where $0 < i < n-1$ and $0 < j < m-1$, estimate the s-derivative using a centered finite difference,

$$\boldsymbol{P}_s(s_i, t_j) \doteq \frac{\boldsymbol{P}(s_{i+1}, t_j) - \boldsymbol{P}(s_{i-1}, t_j)}{2\Delta_s} = \frac{\boldsymbol{P}(s_{i+1}, t_j) - \boldsymbol{P}(s_{i-1}, t_j)}{s_{i+1} - s_{i-1}} \tag{5}$$

where $\Delta_s = 1/n$, $s_{i+1} = s_i + \Delta_s$, and $s_{i-1} = s_i - \Delta_s$. Similarly, the t-derivative is estimated at interior vertices by

$$\boldsymbol{P}_t(s_i, t_j) \doteq \frac{\boldsymbol{P}(s_i, t_{j+1}) - \boldsymbol{P}(s_i, t_{j-1})}{2\Delta_t} = \frac{\boldsymbol{P}(s_i, t_{j+1}) - \boldsymbol{P}(s_i, t_{j-1})}{t_{j+1} - t_{j-1}} \tag{6}$$

where $\Delta_t = 1/m$, $t_{j+1} = t_j + \Delta_t$, and $t_{j-1} = t_j - \Delta_t$.

At a boundary vertex, we can estimate derivatives using one-sided derivatives. For example,

$$\boldsymbol{P}_s(s_0, t_j) \doteq \frac{\boldsymbol{P}(s_1, t_j) - \boldsymbol{P}(s_0, t_j)}{\Delta_s} \doteq \frac{\boldsymbol{P}(s_1, t_j) - \boldsymbol{P}(s_0, t_j)}{s_1 - s_0} \tag{7}$$

Similar one-sided estimates can be used at the other boundaries. The centered finite difference for estimating the s-derivative has an error of $O(\Delta_s^2)$, whereas the one-sided difference has an error of $O(\Delta_s)$. You are free to use other derivative estimates that match the order of the errors and/or use larger neighborhoods; see Derivative Approximation by Finite Differences.

Estimates for second-order positional derivatives are obtained similarly. In the remainder of this section we are going to use centered finite differences with appropriate assumptions about handling boundary positions. Using only the immediate neighbors, the centered finite difference estimates for the second-order derivatives are

$$\boldsymbol{P}_{ss}(s_i, t_j) \doteq \frac{\boldsymbol{P}(s_{i+1}, t_j) - 2\boldsymbol{P}(s_i, t_j) + \boldsymbol{P}(s_{i-1}, t_j)}{\Delta_s^2} \tag{8}$$

and

$$\boldsymbol{P}_{tt}(s_i, t_j) \doteq \frac{\boldsymbol{P}(s_i, t_{j+1}) - 2\boldsymbol{P}(s_i, t_j) + \boldsymbol{P}(s_i, t_{j-1})}{\Delta_t^2} \tag{9}$$

and

$$\boldsymbol{P}_{st}(s_i, t_j) \doteq \frac{\boldsymbol{P}(s_{i+1}, t_{j+1}) + \boldsymbol{P}(s_{i-1}, t_{j-1}) - \boldsymbol{P}(s_{i+1}, t_{j-1}) - \boldsymbol{P}(s_{i-1}, t_{j+1})}{4\Delta_s\Delta_t} \tag{10}$$

These may be used in estimating the principal curvatures and directions at the mesh vertices.

2.1 Rectangle Topology

Assuming no wrap-around of the mesh at the boundary vertices, we can assume for the mesh a *clamp mode* that is similar to the clamp mode used when sampling texture coordinates. Any tuple (i, j) for which at least one of the indices is outside the range of that index is clamped to a boundary index when looking up positions. For example, $(-1, j)$ is clamped to $(0, j)$, (n, j) is clamped to $(n-1, j)$, $(i, -1)$ is clamped to $(i, 0)$, (i, m) is clamped to $(i, m-1)$, and $(-1, -1)$ is clamped to $(0, 0)$ with similar clamping at the other three corners of the lattice. In terms of texture coordinates, for example, $s_{-1} = s_0$ and $s_n = s_{n-1}$ in which case $\boldsymbol{P}(s_{-1}, t_j) = \boldsymbol{P}(s_0, t_j)$ and $\boldsymbol{P}(s_n, t_j) = \boldsymbol{P}(s_{n-1}, t_j)$. Applying clamp mode, we can estimate first-order derivatives using the centered finite differences of equations (5), (6), (8), (9), and (10), whether at interior or boundary mesh vertices.

2.2 Cylindrical Topology

If the rectangular mesh is wrapped around at the s-boundaries or at the t-boundaries but not both, we have a mesh with cylindrical topology. If the mesh has a texture applied, we must deal with the discontinuity in the implied texture coordinates at the wrapped boundary. The standard approach is to duplicate the positions—but not the texture coordinates—at the wrapped boundary.

For example, consider a mesh that is wrapped at the s-boundaries. The original mesh has positions $\boldsymbol{P}_{i,j}$ for $0 \le i < n$ and $0 < j < m$. Increase the lattice size so that $0 < i \le n$ and duplicate the positions $\boldsymbol{P}_{n,j} = \boldsymbol{P}_{0,j}$. At the $i = 0$ boundary the implied texture coordinates are $(0, j/m)$. At the $i = n$ boundary the implied texture coordinates are $(1, j/m)$. A texture image $C(i, j)$ of size $n \times m$ to be applied to the mesh is usually created to avoid visual artifacts at the shared boundary, a *seam* so to speak, by requiring $C(n-1, j) = C(0, j)$. A texture sampling unit is configured so that its s-coordinate is in *wrap mode* or *repeat mode*; when $s = 1$ is encountered during sampling, it is wrapped to $s = 0$.

To estimate first-order derivatives $\boldsymbol{P}_s(s,t)$ and $\boldsymbol{P}_t(s,t)$ at boundary points, even the duplicated ones, positional information from neighbors must be used. This requires either appending yet another set of duplicated positions or using modular arithmetic when computing differences. The second option is preferred; for example, when the mesh is s-wrapped and an estimate for $\boldsymbol{P}_s(0, j/m)$ is required using a centered finite difference, abstractly neighbors with texture coordinates $(-1/n, j/m)$ and $(1/n, j/m)$ are used. The position associated with $(-1/n, j/m)$ is $\boldsymbol{P}((n-1)/n, j/m)$. The centered finite difference estimates are written as

$$\boldsymbol{P}_s(s_i, t_j) \doteq \frac{\boldsymbol{P}(s_{i\oplus 1,j}, t_{i,j}) - \boldsymbol{P}(s_{i\ominus j}, t_{i,j})}{\Delta_s} \tag{11}$$

where $i \oplus 1 = (i+1) \mod n$, $i \ominus 1 = (i-1+n) \mod n$, and $\Delta_s = 1/n$. Think of this estimate as the one mentioned in equation (5) but with wrap mode in the s-coordinate rather than clamp mode used for the rectangle topology mesh.

For wrap mode in the t-coordinate, the index computations are $j \oplus 1 = (j+1) \mod m$ and $j \ominus 1 = (j-1+m) \mod m$. The centered finite difference estimates of equations (5), (6), (8), (9), and (10) may be used with the appropriate index computations based on the active wrap mode. For a cylinder wrapped in the s-coordinate, the index arithmetic for s_i is computed with wrap mode (i is computed modulo n) and the index arithmetic for t_j is computed with clamp mode. For a cylinder wrapped in the t-coordinate, the index arithmetic for s_i is computed with clamp mode and the index arithmetic for t_j is computed with wrap mode (j is computed modulo m).

2.3 Toroidal Topology

If both the s-boundaries and t-boundaries are wrapped, the mesh has with toroidal topology. The ideas for cylindrical topology apply to duplicating positions at both pairs of boundaries but choosing appropriate texture coordinates for the duplicated positions. The estimates are equations (5), (6), (8), (9), and (10) where both indices are computed using modular arithmetic.

2.4 Spherical Topology

A mesh with spherical topology can be constructed from a rectangle-lattice mesh by adding a column of duplicate positions, say, wrapping around in the s-direction to form a cylinder and then adding two extra

rows of positions, each row consisting of the same position. One row corresponds to the north pole and one row corresponds to the south pole.

If the original rectangle-lattice mesh has n columns and m rows, the new mesh has $n+1$ columns and $m+2$ rows. Choose texture coordinates (s_i, t_j) with $s_i = i/n$ for $0 \le i \le n$ and t_j with $t_0 = j/(m+1)$ for $0 \le j \le m+1$. The s-wrap is obtained by $\boldsymbol{P}(0,t_j) = \boldsymbol{P}(s_n,t_j) = \boldsymbol{P}(s_0,t_j) = \boldsymbol{P}(1,t_j)$. The extra rows have $\boldsymbol{P}(s_i,0) = \boldsymbol{A}$ (the south pole) and $\boldsymbol{P}(s_i,1) = \boldsymbol{B}$ (the north pole). For derivative estimates using centered differences, the new mesh is set to have wrap mode in the s-direction and clamp mode in the t-direction.

When rendering a mesh with spherical topology, half the triangles sharing the north pole and half the triangles sharing the south pole are degenerate because of the duplicated positions. If you so choose, you can eliminate the triangles from the mesh after estimating the differential geometric quantities of interest.

3 Differential Geometric Estimates for Parametric Meshes

In this section I assume an arbitrary topology for the mesh. If instead the mesh has a rectangle-lattice topology, you can estimate tangents, normals, principal curvatures, and principal directions using the derivations of Section 2.

3.1 Estimates for Tangent Vectors

Consider an edge of the mesh whose endpoints correspond to the vertices $\boldsymbol{V}_i$ and $\boldsymbol{V}_j$ where $i \ne j$. The vertex positions are surface samples $\boldsymbol{P}_i = \boldsymbol{P}(s_i,t_i)$ and $\boldsymbol{P}_j = \boldsymbol{P}(s_j,t_j)$. The unit-length normal vector at $\boldsymbol{P}_i$ is $\boldsymbol{N}_i = \boldsymbol{N}(s_i,t_i)$.

The *Jacobian matrix* for the parametric surface is the 3×2 matrix of first-order partial derivatives, $J = [\boldsymbol{P}_s\ \boldsymbol{P}_t]$, where the columns are the indicated positional derivatives. The concept of a *directional derivative* that one typically sees for scalar-valued functions extends to multivariable functions. In the case at hand, let $\boldsymbol{w}$ be a unit-length 2×1 direction vector whose components are w_0 and w_1. The rate of change of the position at $\boldsymbol{P}_i$ and in the parametric-coordinate direction $\boldsymbol{w}$ is

$$J_i\boldsymbol{w} = \lim_{h\to 0} \frac{\boldsymbol{P}(s_i + hw_0, t_i + hw_1) - \boldsymbol{P}(s_i,t_i)}{h} \tag{12}$$

The parametric-coordinate direction to be towards $\boldsymbol{P}_j$ is $\boldsymbol{w}_{ji} = (s_j - s_i, t_j - t_i)/\sqrt{(s_j-s_i)^2 + (t_j-t_i)^2}$. The directional derivative at $\boldsymbol{P}_i$ in this direction can be estimated using the right-hand side of equation (12) by removing the limit and setting h to $\sqrt{(s_j-s_i)^2 + (t_j-t_i)^2}$. However, without the limit, the difference of positions is not generally perpendicular to the normal vector $\boldsymbol{N}_i$. A projection of the difference onto the tangent plane is required which is accomplished using multiplication by the matrix $I - \boldsymbol{N}_i\boldsymbol{N}_i^{\mathsf{T}}$,

$$J_i\boldsymbol{w}_{ji} \doteq \frac{(I - \boldsymbol{N}_i\boldsymbol{N}_i^{\mathsf{T}})(\boldsymbol{P}(s_j,t_j) - \boldsymbol{P}(s_i,t_i))}{\sqrt{(s_j-s_i)^2 + (t_j-t_i)^2}} = \frac{(I - \boldsymbol{N}_i\boldsymbol{N}_i^{\mathsf{T}})(\boldsymbol{P}_j - \boldsymbol{P}_i)}{\sqrt{(s_j-s_i)^2 + (t_j-t_i)^2}} = \boldsymbol{d}_{ji} \tag{13}$$

The last equality defines the vector $\boldsymbol{d}_{ji}$. There are $m \ge 2$ edges that share vertex $\boldsymbol{P}_i$ because the vertex is shared by at least one triangle. Equation (13) provides an estimate for a directional derivative. Discarding the approximation error, there are m equations of the form $J_i\boldsymbol{w}_{ji} = \boldsymbol{d}_{ji}$, where j corresponds to the indices of the positions $\boldsymbol{P}_j$ adjacent to $\boldsymbol{P}_i$.

The linear system is overconstrained when $m > 2$, but it can be solved in the sense of least squares. Define W_i to be the $m \times 2$ matrix whose rows are the $\boldsymbol{w}_{ji}$ and define D_i to be the $m \times 3$ matrix whose rows are the $\boldsymbol{d}_{ji}$. The linear system is $JW^{\mathsf{T}} = D^{\mathsf{T}}$. The normal-form system is $J_i W_i^{\mathsf{T}} W_i = D_i^{\mathsf{T}} W_i$. Having at least two distinct triangle edges sharing $\boldsymbol{P}_i$ guarantees that W_i has rank 2, which in turn ensures that $W_i^{\mathsf{T}} W_i$ is invertible. The solution to the system is

$$J_i = D_i^{\mathsf{T}} W_i \left(W_i^{\mathsf{T}} W_i\right)^{-1} \tag{14}$$

The columns of J_i are estimates to the first-order partial derivatives of position at $\boldsymbol{P}_i$. Observe that the estimate of equation (13) is equivalent to $J_i(h\boldsymbol{w}_{ji}) = h\boldsymbol{d}_{ji}$, where as a 2-tuple $h\boldsymbol{w}_{ji} = (s_j - s_i, t_j - t_i)$ and where $h\boldsymbol{d}_{ji} = \boldsymbol{P}_j - \boldsymbol{P}_i$. Thus, an implementation of the algorithm for estimates does not require dividing by h.

Keep in mind that the algorithm produces estimates. In many applications you want unit-length tangent vectors. It is known that a continuous vector field defined on a sphere must have at least one zero vector; thus, if your mesh has the topology of a sphere, even with a dense set of vertices, it is not possible to have a continuously varying set of unit-length tangent vectors [3]. In the discrete world, you likely expect all your estimates to be nonzero, but a visualization of the unit-length tangent vector field can be convincing that there are continuity problems.

3.2 Estimates for Normal Vectors

An estimate of a unit-length normal vector is the normalized cross product of the estimates of the first-order partial derivatives. It is possible that the derivative estimates are parallel, in which case the cross product is zero. Let $\boldsymbol{T}_0$ be the estimate for $\boldsymbol{P}_s$ and let $\boldsymbol{T}_1$ be the estimate for $\boldsymbol{P}_t$. Choose the vertex normal to be

$$\boldsymbol{N} = \begin{cases} \frac{\boldsymbol{T}_0 \times \boldsymbol{T}_1}{|\boldsymbol{T}_0 \times \boldsymbol{T}_1|}, & \boldsymbol{T}_0 \text{ and } \boldsymbol{T}_1 \text{ are not parallel} \\ \boldsymbol{0}, & \boldsymbol{T}_0 \text{ and } \boldsymbol{T}_1 \text{ are parallel} \end{cases} \tag{15}$$

3.3 Estimates for Principal Curvatures

Centered finite differences for a mesh of rectangle-lattice topology allowed us to estimate $\boldsymbol{P}_{ss}$, $\boldsymbol{P}_{st}$, and $\boldsymbol{P}_{tt}$ at a mesh vertex using only the immediate neighbors of the vertex. In turn, the derivative estimates allow us to estimate principal curvatures and principal directions. For a mesh of arbitrary topology, there is no concept of centered finite difference, so there is no simple method to estimate $\boldsymbol{P}_{ss}$, $\boldsymbol{P}_{st}$, and $\boldsymbol{P}_{tt}$. Instead, the approach is to estimate $\boldsymbol{N}_s$ and $\boldsymbol{N}_t$ and use the formulas for principal curvatures and principal directions involving the normal derivatives; see equation (2).

Define the matrix $M = [\boldsymbol{N}_s\ \boldsymbol{N}_t]$. The normal vectors are unit length, so $\boldsymbol{N} \cdot \boldsymbol{N} = 1$. Computing first-order derivatives leads to $\boldsymbol{N} \cdot \boldsymbol{N}_s = 0$ and $\boldsymbol{N} \cdot \boldsymbol{N}_t = 0$. Therefore, the columns of M are perpendicular to $\boldsymbol{N}$ which means they are in the tangent plane to the surface. Similar to the approximation of equation (13) and using the same notation,

$$M_i \boldsymbol{w}_{ji} \doteq \frac{(I - \boldsymbol{N}_i \boldsymbol{N}_i^{\mathsf{T}})(\boldsymbol{N}(s_j, t_j) - \boldsymbol{N}(s_i, t_i))}{\sqrt{(s_j - s_i)^2 + (t_j - t_i)^2}} = \frac{(I - \boldsymbol{N}_i \boldsymbol{N}_i^{\mathsf{T}})(\boldsymbol{N}_j - \boldsymbol{N}_i)}{\sqrt{(s_j - s_i)^2 + (t_j - t_i)^2}} = \boldsymbol{f}_{ji} \tag{16}$$

where the last equality defines the vector $\boldsymbol{f}_{ji}$. For m triangles sharing the vertex, the matrix M_i is the solution to a linear system $M_i W_i^{\mathsf{T}} = F_i^{\mathsf{T}}$, where W_i is the matrix as defined previously and F_i is a matrix

whose rows are the $\boldsymbol{f}_{ji}$. The solution is $M_i = F_i^{\mathsf{T}} W_i (W_i^{\mathsf{T}} W_i)^{-1}$. As in the previous discussion, multiply through by the denominator $\sqrt{(s_j - s_i)^2 + (t_j - t_i)^2}$ to avoid that calculation in an implementation.

Given the estimates for $\boldsymbol{P}_s$, $\boldsymbol{P}_t$, $\boldsymbol{N}_s$, and $\boldsymbol{N}_t$ at position $\boldsymbol{P}_i$, the principal curvatures can be computed by solving the determinant equation $\det(B_i - \kappa G_i) = 0$, where G_i and B_i are the matrices of equations (1) and (2). The principal directions are $J_i \boldsymbol{Y}$ are generated by solving $(B_i - \kappa G_i)\boldsymbol{Y} = \boldsymbol{0}$ for each principal curvature κ. At an umbilic point where the principal curvatures are the same, the dimension of the corresponding generalized eigenspace is 2.

4 Differential Geometric Estimates for Nonparametric Meshes

4.1 Estimates for Normal Vectors

A vertex normal can be estimated by the area-weighted sum of triangle face normals for those triangles sharing the vertex. Let a vertex have position $\boldsymbol{P}_0$ and have m triangles sharing it. Triangle T_i has vertex positions $\boldsymbol{P}_0$, $\boldsymbol{P}_i$, and $\boldsymbol{P}_{i+1}$ for $1 \le i \le m$; the triangles form a triangle fan centered at $\boldsymbol{P}_0$. The area-weighted normal to triangle T_i is $\boldsymbol{W}_i = (\boldsymbol{P}_i - \boldsymbol{P}_0) \times (\boldsymbol{P}_{i+1} - \boldsymbol{P}_0)$ and has the property $|\boldsymbol{W}_i| = 2\,\text{Area}(T_i)$. The estimate of the vertex normal is

$$\boldsymbol{N}_0 = \frac{\sum_{i=1}^{m} \boldsymbol{W}_i}{|\sum_{i=1}^{m} \boldsymbol{W}_i|} \tag{17}$$

Listing 2 contains pseudocode for efficiently estimating the surface normals and using those estimates as the vertex normals.

Listing 2. Computing vertex normals for a nonparametric mesh.

```
void ComputeVertexNormals(in−out Mesh mesh)
{
    for (int v = 0; v < mesh.numVertices; ++v) { mesh.vertex[v].normal = { 0, 0, 0 }; }

    for (int t = 0; t < mesh.numTriangles; ++t)
    {
        // Get the vertex positions for triangle i.
        Point3 P0 = mesh.vertex[mesh.triangle[t].v[0]].position;
        Point3 P1 = mesh.vertex[mesh.triangle[t].v[1]].position;
        Point3 P2 = mesh.vertex[mesh.triangle[t].v[2]].position;

        // Compute the directions for the edges sharing P0.
        Vector3 E1 = P1 − P0;
        Vector3 E2 = P2 − P0;

        // Compute the triangle normal. Its length is twice the area of the triangle.
        Vector3 triangleNormal = Cross(E1, E2);

        mesh.vertex[i0].normal += triangleNormal;
        mesh.vertex[i1].normal += triangleNormal;
        mesh.vertex[i2].normal += triangleNormal;
    }

    for (int v = 0; v < mesh.numVertices; ++v)
    {
        // Normalize the vector to unit-length if it is not the zero vector; otherwise it remains the zero vector.
        Normalize(mesh.vertex[v].normal);
    }
}
```

An alternative is to sum the unit-length normals of the triangles sharing a vertex and then normalizing the sum to produce the vertex normal. This algorithm is biased in the sense that small-area triangles have just as much influence on the normal vector estimate that large-area triangles do. The average of area-weighted normal vectors avoids the bias.

4.2 Estimates for Tangent Vectors

After computing a vertex normal $\boldsymbol{N}$ at the vertex $\boldsymbol{V}$ with position $\boldsymbol{P}$, which modst of the time will not be the zero vector, the tangent plane at the vertex is simply $\boldsymbol{N} \cdot (\boldsymbol{X} - \boldsymbol{P}) = 0$, where $\boldsymbol{X}$ is any point on the tangent plane.

4.2.1 Nonlocal Estimates

If all that is needed is a pair of unit-length tangent vectors at the vertex, choose vectors $\boldsymbol{U}$ and $\boldsymbol{V}$ in the tangent plane for which $\{\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{N}\}$ is a right-handed orthonormal set; that is, the vectors are unit length, mutually perpendicular, and $\boldsymbol{N} = \boldsymbol{U} \times \boldsymbol{V}$. A robust algorithm is shown in listing 3.

Listing 3. Robust computation of unit-length tangent vectors from a unit-length normal vector; the normal cannot be the zero vector on input. The vector components are accessed as array elements. On output, the normal and tangent vectors form a right-handed orthonormal basis.

```
void ComputeTangents(in Vector3 N, out Vector3 U, out Vector3 V)
{
    // On input, N is not the zero vector and must be unit length.
    if (abs(N[0]) > abs(N[1]))
    {
        U = { −N[2], 0, +N[0] };
    }
    else
    {
        U = { 0, +N[2], −N[1] };
    }
    Normalize(U);

    // V is unit length because N and U are unit length and perpendicular.
    V = Cross(N, U);
}
```

By definition, the first-order derivatives make sense only when you have a parameterization, which the nonparametric mesh does not have. However, a local parameterization of the tangent plane is used in order to produce estimates that can be used for higher-order information. Alternatively, it is possible to generating texture coordinates for the mesh and use them as the underlying parameterization of the surface. A good discussion of such algorithms is [1]. An algorithm specific to meshes with spherical topology is [2].

4.2.2 Estimates using a Local Parameterization

The vertex has position $\boldsymbol{P}_0$ which plays the role of origin for the tangent plane. Let $\boldsymbol{N}_0$ be the vertex normal computed from the area-weighted normals of the triangles sharing the vertex. Let $\boldsymbol{U}_0$ and $\boldsymbol{V}_0$ be the tangent vectors computed from $\boldsymbol{N}_0$ using the nonlocal construction discussed previously. Any point in the

plane is of the form $\boldsymbol{X} = \boldsymbol{P}_0 + s\boldsymbol{U}_0 + t\boldsymbol{V}_0$ for local parameters s and t. The triangle edges that share $\boldsymbol{P}_0$ give positional variation in the edge directions. An edge direction can be projected onto the tangent plane, and the local parameters may be used to estimate partial derivatives with respect to those parameters using the methods of Section 3.

Specifically, let T_i for $1 \le i \le m$ be the triangles sharing $\boldsymbol{P}_0$. The triangle vertices are $\boldsymbol{P}_0$, $\boldsymbol{P}_i$, and $\boldsymbol{P}_{i+1}$, which form a triangle fan centered at $\boldsymbol{P}_0$. Consider an edge $\boldsymbol{P}_0$ to $\boldsymbol{P}_j$ for some j. The position $\boldsymbol{P}_0$ is assigned parameters $(s_0, t_0) = (0, 0)$. The position $\boldsymbol{P}_j$ is assigned parameters (s_j, t_j) by projection onto the tangent plane,

$$(I - \boldsymbol{N}_0\boldsymbol{N}_0^{\mathsf{T}})(\boldsymbol{P}_j - \boldsymbol{P}_0) = s_j\boldsymbol{U}_0 + t_j\boldsymbol{V}_0 \tag{18}$$

in which case

$$s_j = \boldsymbol{U}_0 \cdot (I - \boldsymbol{N}_0\boldsymbol{N}_0^{\mathsf{T}})(\boldsymbol{P}_j - \boldsymbol{P}_0), \quad t_j = \boldsymbol{V}_0 \cdot (I - \boldsymbol{N}_0\boldsymbol{N}_0^{\mathsf{T}})(\boldsymbol{P}_j - \boldsymbol{P}_0) \tag{19}$$

The methods of Section 3 can now be applied to estimate positional derivatives and normal derivatives.

It is possible in the construction that the projection of $\boldsymbol{P}_j$ is $\boldsymbol{P}_0$ for some j. We can discard such points in the estimate. If $\boldsymbol{P}_0$ is contained by only one triangle, both edge projections lead to $(s_j, t_j) \neq (0, 0)$. If $\boldsymbol{P}_0$ is contained by two or more triangles, the constraints placed on the manifold mesh guarantees that even with a discarded edge, there are still at least two edges that are not parallel to $\boldsymbol{N}_0$. Therefore, the matrix W in the construction of Section 3 will have full rank so that $W^{\mathsf{T}}W$ is invertible.

4.3 Estimates for Principal Curvatures

Using the local parameterization discussed in the previous section, we can estimate $\boldsymbol{P}_s$ and $\boldsymbol{P}_t$. From these we can estimate the normals $\boldsymbol{N}$ and then the derivatives $\boldsymbol{N}_s$ and $\boldsymbol{N}_t$. The methods of Section 3.3 then apply to estimate principal curvatures and principal directions.

Listing 4 contains a Geometric Tools implementation of the algorithm described here.

Listing 4. A Geometric Tools implementation for estimating principal curvatures and principal directions at the vertices of a triangle mesh. The vertices has n elements and the array indices has $3n$ elements. Each triple of indices defines a triangle of the mesh.

```
void EstimatePrincipalCurvatureInformation(
    std::vector<Vector3<double>> const& vertices,
    std::vector<int32_t> const& indices,
    std::vector<Vector3<double>>& normals,
    std::vector<double>& minCurvatures,
    std::vector<double>& maxCurvatures,
    std::vector<Vector3<double>>& minDirections,
    std::vector<Vector3<double>>& maxDirections)
{
    size_t const numVertices = vertices.size();
    size_t const numTriangles = indices.size() / 3;

    minCurvatures.resize(numVertices);
    maxCurvatures.resize(numVertices);
    minDirections.resize(numVertices);
    maxDirections.resize(numVertices);

    // Also, keep track of the adjacent vertices to each mesh vertex for
    // use in computing local surface parameterizations.
    normals.resize(numVertices);
    std::fill(normals.begin(), normals.end(), Vector3<double>::Zero());
    std::vector<std::set<size_t>> adjacents(numVertices);
```

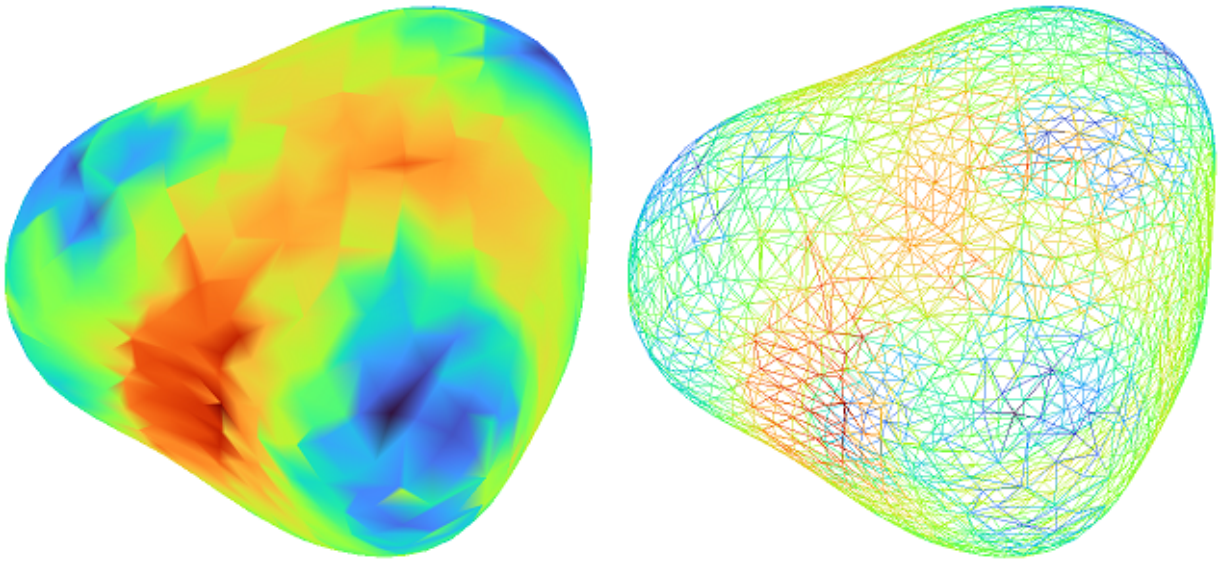
```
for (size_t t = 0; t < numTriangles; ++t)
{
    // Vertex positions for triangle t.
    size_t v0 = indices[3 * t];
    size_t v1 = indices[3 * t + 1];
    size_t v2 = indices[3 * t + 2];

    Vector3<double> const& P0 = vertices[v0];
    Vector3<double> const& P1 = vertices[v1];
    Vector3<double> const& P2 = vertices[v2];

    // Edge directions for triangle t.
    Vector3<double> E10 = P1 - P0;
    Vector3<double> E20 = P2 - P0;

    // Normal vector for triangle t, twice the area of the triangle.
    Vector3<double> triangleNormal = Cross(E10, E20);

    // Update the sums of normals at the vertices.
    normals[v0] += triangleNormal;
    normals[v1] += triangleNormal;
    normals[v2] += triangleNormal;

    // Store the indices to the adjacent vertices.
    adjacents[v0].insert(v1);
    adjacents[v0].insert(v2);
    adjacents[v1].insert(v0);
    adjacents[v1].insert(v2);
    adjacents[v2].insert(v0);
    adjacents[v2].insert(v1);
}

// Normalize the sums to obtain unit-length normal vectors.
for (size_t v = 0; v < numVertices; ++v)
{
    Normalize(normals[v]);
}

// Compute an orthonormal basis {U,V,N} at each vertex.
std::vector<Vector3<double>> U(numVertices), V(numVertices);
for (size_t v = 0; v < numVertices; ++v)
{
    std::array<Vector3<double>, 3> basis{};
    basis[0] = normals[v];
    ComputeOrthogonalComplement(1, basis.data());
    U[v] = basis[1];
    V[v] = basis[2];
}

// Compute principal curvatures.
for (size_t v = 0; v < numVertices; ++v)
{
    int32_t numRows = static_cast<int32_t>(adjacents[v].size());
    GMatrix<double> W(numRows, 2);
    GMatrix<double> D(numRows, 3);
    GMatrix<double> F(numRows, 3);
    int32_t row = 0;
    for (auto a : adjacents[v])
    {
        Vector3<double> deltaP = vertices[a] - vertices[v];
        W(row, 0) = Dot(U[v], deltaP);
        W(row, 1) = Dot(V[v], deltaP);
        D(row, 0) = deltaP[0];
        D(row, 1) = deltaP[1];
        D(row, 2) = deltaP[2];

        Vector3<double> deltaN = normals[a] - normals[v];
        F(row, 0) = deltaN[0];
        F(row, 1) = deltaN[1];
        F(row, 2) = deltaN[2];

        ++row;
```

```
}

// Estimate dP/ds and dP/dt.
GVector<double> N(3);
N[0] = normals[v][0];
N[1] = normals[v][1];
N[2] = normals[v][2];
GMatrix<double> ProjectionN = GMatrix<double>::Identity(3, 3) - OuterProduct(N, N);
GMatrix<double> WTW = Transpose(W) * W;
GMatrix<double> invWTW = Inverse(WTW);
GMatrix<double> DTW = Transpose(D) * W;
GMatrix<double> J = ProjectionN * DTW * invWTW;
GVector<double> DPDS = J.GetCol(0);
GVector<double> DPDT = J.GetCol(1);

// Estimate dN/ds and dN/dt.
GMatrix<double> FTU = Transpose(F) * W;
GMatrix<double> M = ProjectionN * FTU * invWTW;
GVector<double> DNDS = M.GetCol(0);
GVector<double> DNDT = M.GetCol(1);

// Compute the curvature and metric tensors.
Matrix2x2<double> B{}, G{};
double b00 = -Dot(DNDS, DPDS);
double b01 = -0.5 * (Dot(DNDS, DPDT) + Dot(DNDT, DPDS));
double b11 = -Dot(DNDT, DPDT);
double g00 = Dot(DPDS, DPDS);
double g01 = Dot(DPDS, DPDT);
double g11 = Dot(DPDT, DPDT);

// Compute the principal curvatures as generalized eigenvalues
// of B-k*G.
double c0 = b00 * b11 - b01 * b01;
double c1 = 2.0 * b01 * g01 - b00 * g11 - b11 * g00;
double c2 = g00 * g11 - g01 * g01;
std::array<PolynomialRoot<double>, 2> roots{};
size_t numRoots = RootsQuadratic<double>::Solve(false, c0, c1, c2, roots.data());
double& kMin = minCurvatures[v];
double& kMax = maxCurvatures[v];
if (numRoots == 2)
{
    kMin = roots[0].x;
    kMax = roots[1].x;
}
else if (numRoots == 1)
{
    kMin = roots[0].x;
    kMax = minCurvatures[v];
}
else
{
    // This is an unexpected condition theoretically, but it might
    // be possible because of floating-point rounding errors. Return
    // all zero-valued information as a signal to the caller.
    kMin = 0.0;
    kMax = 0.0;
}

// Compute the corresponding principal directions. At an umbilic,
// the principal direction pair is not unique. Return zero-valued
// vectors in this case.
Vector3<double>& dMin = minDirections[v];
Vector3<double>& dMax = maxDirections[v];
if (kMin != kMax)
{
    // Compute the rows of B - kMin * G. Select the row with largest
    // length for numerical robustness. The generalized eigenvector
    // is perpendicular to the selected row.
    Vector2<double> row0 = { b00 - kMin * g00, b01 - kMin * g01 };
    Vector2<double> row1 = { row0[1], b11 - kMin * g11 };
    double sqrLength0 = Dot(row0, row0);
    double sqrLength1 = Dot(row1, row1);
```

```
            if (sqrLength0 >= sqrLength1)
            {
                Normalize(row0);
                dMin = row0[0] * U[v] + row0[1] * V[v];
            }
            else
            {
                Normalize(row1);
                dMin = row1[0] * U[v] + row1[1] * V[v];
            }

            // Repeat the process with B - kMax * G.
            row0 = { b00 - kMax * g00, b01 - kMax * g01 };
            row1 = { row0[1], b11 - kMax * g11 };
            sqrLength0 = Dot(row0, row0);
            sqrLength1 = Dot(row1, row1);
            if (sqrLength0 >= sqrLength1)
            {
                Normalize(row0);
                dMax = row0[0] * U[v] + row0[1] * V[v];
            }
            else
            {
                Normalize(row1);
                dMax = row1[0] * U[v] + row1[1] * V[v];
            }
        }
        else
        {
            dMin = { 0.0, 0.0, 0.0 };
            dMax = { 0.0, 0.0, 0.0 };
        }
    }
}
```

The triangle mesh used in the verifying the ideas is based on https://houdinigubbins.wordpress.com/2017/04/25/discrete-curvature/. However, the mesh I used is low resolution. The figures in the weblink are based on more sophisticated algorithms that require more programming effort. Screen captures from my experiment show the maximum principal curvature using pseudocoloring; see figure 1.

Figure 1. Estimated maximum principal curvatures for a low-resolution triangle mesh. The left image is a solid-fill rendering. The right image is a wireframe rendering.



References

[1] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon Mesh Processing*. AK Peters Ltd., Natick, MA, 2010.

[2] S. Haker, S. Angenant, A. Tannenbaum, R. Kikinis, G. Sapiro, and M. Halle. Conformal surface parameterization for texture mapping. In *IEEE Transactions on Visualization and Computer Graphics*, volume 6, pages 181–189, April-June 2000.

[3] Wikipedia. Hairy ball theorem.
https://en.wikipedia.org/wiki/Hairy_ball_theorem.
accessed August 12, 2023.