Approximating Surface Curvature on Triangle Meshes

Second Year Essay

2000642

Table of Contents

1 Introduction **1**

2 The Curvature of Smooth Surfaces in $\mathbb{R}^3$ **1**
2.1 The Tangent Plane and Normal Field 2
2.2 The Gauss Map 3
2.3 The Second Fundamental Form 4
2.4 Curvature 5

3 Approximating Curvature on Triangle Meshes **7**
3.1 Programming with Triangle Meshes 7
3.2 The Discrete Second Fundamental Form 8

4 Results **12**
4.1 Results on a Sphere 12
4.2 Results on a Torus 13

5 Summary **15**

Appendix **18**

This essay is 12 pages long with all figures removed.

1 Introduction

In the fields of geometry processing and computer graphics, triangle meshes are a common way to represent surfaces. They are essentially collections of connected triangles that approximate a smooth surface. Unlike smooth surfaces, there is no meaningful notion of differentiability for triangle meshes. As a result, many methods have been developed to analyse their differential properties. One such property is curvature, with applications of curvature analysis ranging from brain scans [2] to illustrative rendering [9] to facial recognition [20].

Many ways of calculating discrete curvature have been developed. This essay explores one of them, namely the use of the second fundamental form. We first examine the importance of the second fundamental form in analysing the curvature of smooth surfaces. A discrete analogue to the second fundamental form can then be formulated and used to calculate the curvature at the vertices of a triangle mesh. Finally, meshes of a sphere and torus with different levels of refinement will be used to analyse the accuracy of the method.

2 The Curvature of Smooth Surfaces in $\mathbb{R}^3$

At a high level, a surface in $\mathbb{R}^3$ is a local mapping of an open set in $\mathbb{R}^2$ to a smooth two-dimensional subset of $\mathbb{R}^3$. Formally, we want a definition that encapsulates a two-dimensional space that is 'smooth enough' to have differential properties. The following definition is provided by do Carmo [4, pg.52].

Definition 2.1. (Regular surface) A subset $S \subset \mathbb{R}^3$ is a regular surface if, for each $p \in S$, there exists a neighbourhood V in $\mathbb{R}^3$ and a map $\sigma : U \to V \cap S$ of an open set $U \subset \mathbb{R}^2$ onto $V \cap S \subset \mathbb{R}^3$ such that σ is a differentiable homeomorphism, and for each $q \in U$ the differential $d\sigma_q : \mathbb{R}^2 \to \mathbb{R}^3$ is injective[1].

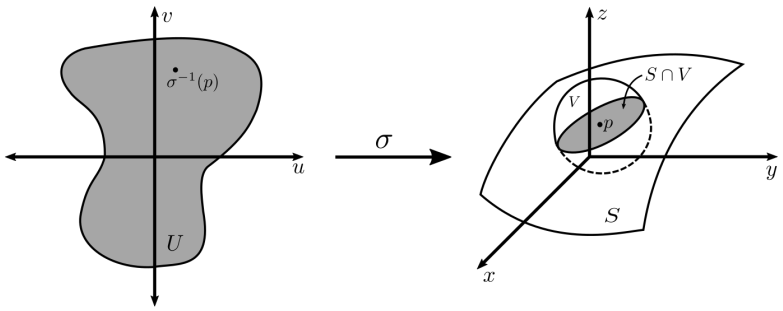


Figure 1: Mapping to from an open set U to a surface in $\mathbb{R}^3$

A surface can be represented locally as a map $\sigma : U \subset \mathbb{R}^2 \to \mathbb{R}^3$ in terms of parameters u and v as $\sigma(u, v) = (x(u, v), y(u, v), z(u, v))$, where σ is a non-unique parametrisation.

[1] Note that this definition gives us local properties of a surface, and we will need to work somewhat harder for a set of global properties. This essay is concerned with finding curvatures at given points on embedded, orientable surfaces, so most nuances can largely be ignored.

Consider a point $p = \sigma(u_0, v_0)$ on our surface. If one parameter of σ is varied, we move along a line parallel to either the x-axis or y-axis in $\mathbb{R}^2$ mapped onto our surface. These are know as coordinate curves, given by the parametrisations $c_{v_0}, c_{u_0} : [a, b] \to \mathbb{R}^3$, where $c_{v_0}(s) = \sigma(s, v_0)$ and $c_{u_0}(s) = \sigma(u_0, s)$ [19, pg.57].

The tangent vectors in $\mathbb{R}^3$ to the coordinate curves at p are

$$\sigma_u = \frac{\partial \sigma}{\partial u}(u_0, v_0) \quad \text{and} \quad \sigma_v = \frac{\partial \sigma}{\partial v}(u_0, v_0).$$

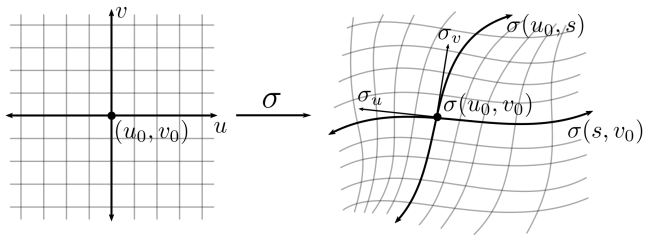


Figure 2: Coordinate curves with the set of tangent vectors for $p = \sigma(u_0, v_0)$

2.1 The Tangent Plane and Normal Field

Let $\gamma : [0, 1] \to \mathbb{R}^3$, $\gamma(t) = \sigma(u(t), v(t))$ represent a curve on our surface S. By the chain rule,

$$\frac{d\sigma}{dt} = \sigma_u \frac{du}{dt} + \sigma_v \frac{dv}{dt}. \tag{1}$$

All vectors $\frac{d\sigma}{dt}$ through p tangent to the surface satisfy equation (1) and thus lie in the plane spanned by σ_u and σ_v at p [19, pg.62]. This plane is known as the *tangent plane*, and denoted T_pS. The tangent plane gives us the best linear approximation to a surface at a point.

Definition 2.2 (Surface normal)**.** The unit surface normal at a point p is a vector at p perpendicular to the tangent plane, given by

$$N_p = \frac{\sigma_u \times \sigma_v}{||\sigma_u \times \sigma_v||}. \tag{2}$$

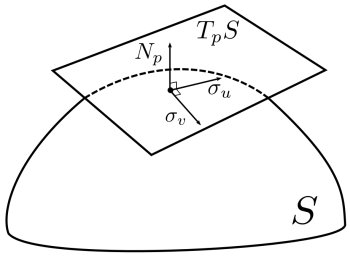


Figure 3: The tangent plane and unit normal at a point on a surface

Due to the anti-symmetric property of the cross product, N_p and $-N_p$ are both perpendicular to the tangent plane, so the normal vector at a point is not

unique. To resolve this, equation (2) will always be taken as the definition of the normal. This means that two different parametrisations of the same surface could have surface normals (consistently) pointing in opposite directions.

A normal vector field on S is a function N that assigns to each $p \in S$ a unit normal vector N_p. If a differentiable normal field can be defined over the whole surface, then our surface is orientable[2], and the normal field is known as the orientation [4, pg.105]. Orientability ensures many desirable properties, so for the purposes of this essay, we will only consider orientable surfaces.

2.2 The Gauss Map

We now introduce the Gauss map, and show that its differential is a self-adjoint linear map. In particular, this means the differential is associated with a symmetric bilinear form, which has real eigenvalues and orthogonal eigenvectors. These results will play an important role in the analysis of surfaces.

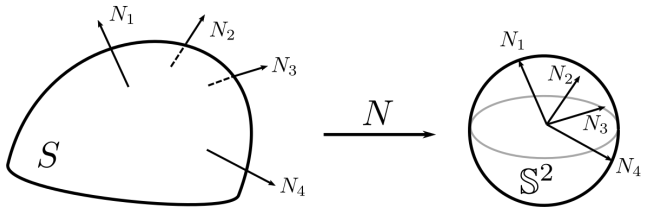


Figure 4: The Gauss map of several points on a surface

Definition 2.3 (Gauss map)**.** Let S be an orientable surface and $\mathbb{S}^2$ the unit sphere in $\mathbb{R}^3$ centred at the origin. The Gauss map $N : S \to \mathbb{S}^2$ takes a point p on S and maps it to its unit normal vector $N(p) \in \mathbb{S}^2$ [13, pg.160].

Since we are dealing with smooth, orientable surfaces, the Gauss map is differentiable, with differential dN_p at $p \in S$. This differential measures the rate of change of a normal vector relative to a curve on the surface and is characterised by a linear map. Moreover, dN_p is a map from the tangent plane of the surface at p to the tangent plane of $\mathbb{S}^2$ at $N(p)$. That is $dN_p : T_pS \to T_{N(p)}\mathbb{S}^2$. However, T_pS and $T_{N(p)}\mathbb{S}^2$ are parallel planes, so we can equally consider dN_p as a map from T_pS to itself [4, pg.136].

Let $\sigma(u, v)$ be a parametrisation of S and $\gamma(t) = \sigma(u(t), v(t))$ be a parametrised curve $\gamma : [0, 1] \to S$ such that $\gamma(0) = p$. Given $\gamma'(0) \in T_pS$,

$$dN_p(\gamma'(0)) = \frac{d}{dt}(N(\gamma(t)))\Big|_{t=0} = (N \circ \gamma)'(0) \tag{3}$$

[13, pg.160]. However, it is more useful to be able to write dN_p as the matrix of a linear map in a basis of T_pS. The tangents σ_u and σ_v to the coordinate

[2]This is equivalent to requiring that if a point belongs to two coordinate neighbourhoods, the change of coordinates has a positive Jacobian [4, pg.105], which is another common definition.

curves at p are a typical choice. We have

$$dN_p(\sigma_u) = (N \circ \sigma(u, v_0))'(u_0) = \frac{\partial N}{\partial u}\Big|_{(u_0,v_0)} = N_u,$$

and similarly, $dN_p(\sigma_v) = N_v$ [13, pg.161]. Hence, dN_p can be written as $\begin{pmatrix} N_u & N_v \end{pmatrix}$, where N_u and N_v are column vectors in the basis $\{\sigma_u, \sigma_v\}$.

Proposition 2.1. *The differential $dN_p : T_pS \to T_pS$ of the Gauss Map is a self-adjoint linear map [4, pg.140].*

Proof. It suffices to show that $dN_p(x) \cdot y = x \cdot dN_p(y)$ for any basis $\{x, y\}$ of T_pS. So, for simplicity, we will use the basis $\{\sigma_u, \sigma_v\}$ given by the parametrisation of S. Since $dN_p(\sigma_u) = N_u$ and $dN_p(\sigma_v) = N_v$, this simplifies to $N_u \cdot \sigma_v = \sigma_u \cdot N_v$. As N is perpendicular to σ_u and σ_v, we have $N \cdot \sigma_u = 0$ and $N \cdot \sigma_v = 0$. Differentiating these equations relative to v and u respectively gives

$$N_v \cdot \sigma_u + N \cdot \sigma_{uv} = 0, \tag{4}$$

$$N_u \cdot \sigma_v + N \cdot \sigma_{vu} = 0. \tag{5}$$

By subtracting equation (4) from (5), we obtain $N_u \cdot \sigma_v = N_v \cdot \sigma_u$. Hence dN_p is self-adjoint. □

2.3 The Second Fundamental Form

The second fundamental form is a concept closely related the differential of the Gauss map, and will be used extensively in Section 3.

Definition 2.4 (Second fundamental form)**.** The symmetric bilinear form defined for $x, y \in T_pS$ by $\mathrm{II}_p(x, y) = -dN_p(x) \cdot y$ is the second fundamental form of S at p. [18, pg.124]

The second fundamental form can be written as a matrix terms of two basis vectors $\{x, y\}$ of T_pS as

$$\mathrm{II}_p = \begin{pmatrix} \mathrm{II}_p(x,x) & \mathrm{II}_p(x,y) \\ \mathrm{II}_p(y,x) & \mathrm{II}_p(y,y) \end{pmatrix} =: \begin{pmatrix} L_p & M_p \\ M_p & N_p \end{pmatrix}.$$

Proposition 2.2. *For an orthonormal basis e_1, e_2 of T_pS,*

$$\mathrm{II}_p(e_1, e_2) = -dN_p.$$

Proof. For a parametrisation $\sigma(u, v)$ of a surface, dN_p can be represented by the partial derivatives N_u and N_v of the unit normal, written in terms of any basis of the tangent space. Since dN_p is a symmetric bilinear form in the orthonormal $\{e_1, e_2\}$ basis, it can be be represented by $N_u = ae_1 + be_2$ and $N_v = be_1 + ce_2$. This gives

$$\begin{aligned}
\mathrm{II}_p(e_1, e_1) &= -dN_p(e_1) \cdot e_1 = -(ae_1 + be_2) \cdot e_1 = -a, \\
\mathrm{II}_p(e_2, e_1) &= \mathrm{II}_p(e_1, e_2) = -dN_p(e_1) \cdot e_2 = -(ae_1 + be_2) \cdot e_2 = -b, \\
\mathrm{II}_p(e_2, e_2) &= -dNp(e_2) \cdot e_2 = -(be_1 + ce_2) \cdot e_2 = -c,
\end{aligned}$$

as desired. □

2.4 Curvature

We now want to develop a notion of curvature for surfaces. The curvature of a surface measures how much a surface deviates from being a plane. Analogies can be drawn to the curvature of a curve γ, which is given by $||\gamma''||$ for γ parametrised by arc-length. In fact, curves with a given tangent on the surface passing through a given point can be used to define the curvature of the surface at that point, in the direction of the tangent vector.

More formally, let $\gamma : [0, L] \to \mathbb{R}^3$ be a curve parametrised by arc length on S, with $\gamma(0) = p$ and $\gamma'(0) \in T_pS$. As the tangent of a curve is perpendicular to the normal, $\gamma'(0) \cdot N_p = 0$. Differentiation yields $\gamma''(0) \cdot N_p + \gamma'(0) \cdot dN_p(\gamma'(0)) = 0$. Hence,

$$\gamma''(0) \cdot N_p = -\gamma'(0) \cdot dN_p(\gamma'(0)). \tag{6}$$

The inner product $\gamma''(0) \cdot N_p$ is the component of acceleration of the curve γ that is normal to the surface, and is therefore known as the *normal curvature*, $\kappa_n(v)$, where $v = \gamma'(0)$ [15, pg.202]. Meusnier's Theorem states that all curves with tangent v at a point p have the same normal curvature [4, pg.142]. Moreover, from equation (6) it can be seen that the normal curvature is given by the second fundamental form: $\kappa_n(v) = \mathrm{I\!I}_p(v, v)$.

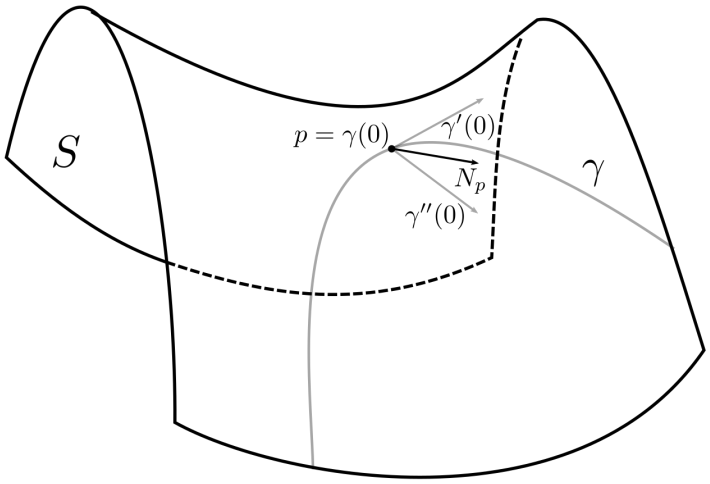


Figure 5: The normal curvature at a point

As the tangent vector v varies, so does the normal curvature. There is a beautiful result that allows us to calculate the maximum and minimum curvatures, known as the *principal curvatures*, and the tangent vectors that yield them, the *principal directions*.

Theorem 2.3. *At a point, the curvature κ_n has a minimum of k_1 in one direction and maximum of k_2 in a perpendicular direction. These curvatures are given by the eigenvalues of $-dN_p$, and their directions by the corresponding eigenvectors [18, pg.127].*

Proof. Since $-dN_p$ is self adjoint, it can be represented by a quadratic form. As a result, there exists an orthonormal basis $\{e_1, e_2\}$ of T_pS such that

$$-dN_p(e_1) = k_1 e_1 \quad \text{and} \quad -dN_p(e_2) = k_2 e_2, \tag{7}$$

where k_1, k_2 are the eigenvalues of $-dN_p$ and e_1, e_2 are their respective eigenvectors. It is customary to take use $-dN_p$ so that the correct sign is obtained for the final curvature values.

Moreover, any unit vector $v \in T_pS$ can be written as $v = e_1 \cos\theta + e_2 \sin\theta$, where θ measures the angle between v and e_1. Now,

$$\begin{aligned}
\kappa_n(v) &= \mathrm{II}(v, v) \\
&= -dN_p(v) \cdot v \\
&= (k_1(\cos\theta)e_1 + k_2(\sin\theta)e_2) \cdot ((\cos\theta)e_1 + (\sin\theta)e_2) \\
&= k_1 \cos^2\theta + k_2 \sin^2\theta.
\end{aligned}$$

Assuming $k_1 \leq k_2$, applying simple calculus shows that a maximum of k_2 is achieved when $\theta = 0$ and a minimum of k_1 is achieved when $\theta = \frac{\pi}{2}$. □

Corollary. *Let $e_1, e_2 \in T_pS$ be orthonormal, then the principal curvatures k_1 and k_2 are the eigenvalues of $\mathrm{II}_p(e_1, e_2)$.*

Proof. This follows directly from Proposition 2.2 and Theorem 2.3. □

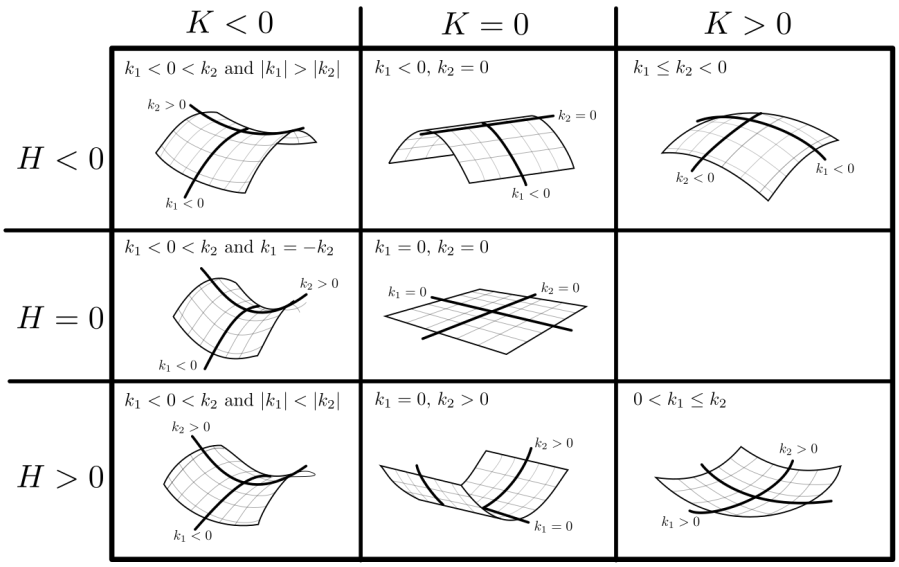


Figure 6: Surfaces obtained for different signs of Gaussian and mean curvature

Definition 2.5 (Gaussian and mean curvatures)**.** The Gaussian and mean curvatures at a point are defined as $K = k_1 k_2$ and $H = \frac{1}{2}(k_1 + k_2)$ respectively, where k_1 and k_2 are the eigenvalues of $-dN_p$ [15, pg.209].

It follows directly from this definition that for an orthonormal basis $\{e_1, e_2\}$ of T_pS, $K = \det(\mathrm{I\!I}_p(e_1, e_2))$ and $H = \mathrm{tr}(\mathrm{I\!I}_p(e_1, e_2))$.

The Gaussian and Mean curvatures fully describe local bending, as seen in Figure 6. A more thorough discussion can be found in *Elementary Differential Geometry* [15, pg.209-211].

3 Approximating Curvature on Triangle Meshes

Triangle meshes are a computer based data structure used for many purposes. In this section, we build on the results from Section 2 to develop a measure of curvature that works on discrete surfaces. Details are based on the method provided by Szymon Rusinkiewicz in his paper *Estimating Curvatures and their Derivatives on Triangle Meshes* [16]. Corresponding code was written in MATLAB by the author of this essay, and is included in the Appendix.

In this essay, we will be working with a finite collection of triangles that approximate a closed surface and call this a triangle mesh. Our approximation must conserve the general properties of a smooth surface in $\mathbb{R}^3$. To ensure this, the following definition, from *Graphs, Surfaces and Homology* [5, pg.38-40], must be satisfied.

Definition 3.1. A closed surface is a finite collection F of triangles such that

(i) M satisfies the intersection condition, namely that two triangles are either:

  (a) disjoint,

  (b) have one vertex in common,

  (c) have two vertices and the edge joining them in common.

(ii) M is connected, that is, there exists a path along the edges of the triangles from any vertex to any other vertex.

(iii) For every vertex v of a triangle of M, the edges opposite v (known as the link) form a simple closed polygon.

We are only considering closed surfaces in this essay so that we do not need to deal with boundary conditions or infinite surfaces.

3.1 Programming with Triangle Meshes

The most common way to represent a triangle mesh is as an indexed face set. This stores the mesh as two arrays. The first is a $V \times 3$ array of the coordinates of the vertices. The second is a $F \times 3$ array of the three vertices that make up each face, according to their index in the first array [1, pg.89-90].

As with smooth surfaces, triangle meshes have a notion of orientability. If a triangle has ordered vertices $\{i, j, k\}$ then writing these vertices such that we move around the triangle in a clockwise manner gives a clockwise orientation, and similarly for an anticlockwise orientation [5, pg.44].

Definition 3.2 (Coherently oriented)**.** Two triangles with a common edge are said to be coherently oriented if they both have the same (clockwise or anticlockwise) orientation [5, pg.44].

A closed surface is said to be orientable if all its triangles can be given an orientation such that two triangles with a common edge are oriented coherently [5, pg.44]. Otherwise, then the surface is non-orientable.

In Figure 7 we have a simple triangle mesh with clockwise orientation. It is represented by the two matrices

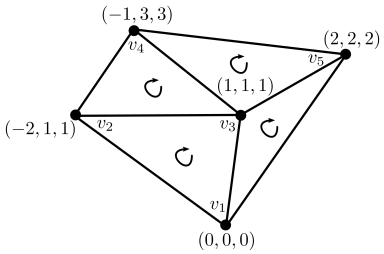


Figure 7: A simple triangle mesh

$$V = \begin{pmatrix} 0 & 0 & 0 \\ -2 & 1 & 1 \\ 1 & 1 & 1 \\ -1 & 3 & 3 \\ 2 & 2 & 2 \end{pmatrix} \quad \text{and} \quad F = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 3 \\ 3 & 4 & 5 \\ 1 & 3 & 5 \end{pmatrix}.$$

This mesh is not a closed surface, but the link of v_3 is the quadrilateral $v_1v_2v_4v_5$.

Given any face of an oriented triangle mesh, its unit normal is the vector perpendicular to it, defined as the normalised cross product of two of its edges. However, there is no clear notion of a normal at edges or vertices. Per-vertex normals are an important concept as vertices are where we actually see the surface changing shape. Therefore, we want to find an interpolation that gives us a suitable per vertex normal for our mesh. There are several methods for doing so. To name a few, per vertex normals can be calculated by averaging over angles adjacent to the vertex [1, pg.145], averaging over the areas of triangles adjacent to the vertex [7, pg.72], or averaging over the areas of the triangles divided by the square of their edge lengths [12].

Code for calculating per-face and per-vertex normals (using the final method) is provided in Appendix Subsections A.1 and A.2 respectively.

3.2 The Discrete Second Fundamental Form

Now that we understand the basics of a triangle mesh, we can consider a discrete analogue to the second fundamental form, presented in *Estimating Curvatures and their Derivatives on Triangle Meshes* [16]. From, Proposition 2.2 we know that for an orthonormal basis of the tangent space, $\{u, v\}$, and a vector s in the tangent space,

$$\mathrm{I\!I}_p s = -dN_p s = -\partial_s N_p, \tag{8}$$

where ∂_s represents a derivative in the direction of s. Because of the way surfaces are parametrised, it is customary to take inward facing normals for closed surfaces. On triangle meshes, we want to instead take an outwards facing normal, and retain the same (signed) curvature values. Negating one side of (8) resolves this and gives

$$\mathrm{I\!I}_p s = \partial_s N_p. \tag{9}$$

Consider a general triangle in some mesh. As shown in Figure 8, an orthonormal basis $\{u_f, v_f\}$ can be defined by normalising an edge to find u_f, and applying the Gram-Schmidt process to a second edge to find v_f. Code to calculate the face coordinate system can be found in Appendix Subsection A.3.

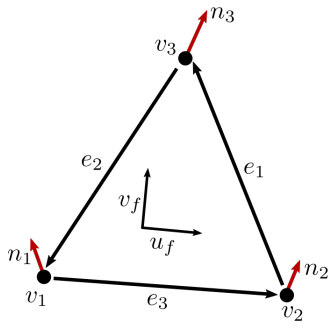


Figure 8: One face of a mesh

On our triangle, the three oriented edges give three well defined directions, which can be written in the $\{u_f, v_f\}$ basis as $(e_i \cdot u_f, e_i \cdot v_f)$ for $1 \leq i \leq 3$. Moreover, the difference between adjacent per-vertex normals gives a derivative of the normal in the direction of an edge [16]. These 'directional derivatives' are then projected onto the u_f, v_f plane since the derivative of the normal must lie in the tangent space. So in the u_f, v_f basis, the directional derivative of the normal is given by $((n_{i+1} - n_i) \cdot u_f, (n_{i+1} - n_i) \cdot v_f)$, where $1 \leq i \leq 3$ and $n_4 = n_1$ [16].

Based on Definition 2.4 of the smooth second fundamental form, the second fundamental form on each face can be represented by the symmetric matrix

$$\mathrm{I\!I}_f = \begin{pmatrix} L_f & M_f \\ M_f & N_f \end{pmatrix}.$$

From equation (9) we have the following equations:

$$\begin{pmatrix} L_f(e_1 \cdot u_f) + M_f(e_1 \cdot v_f) \\ M_f(e_1 \cdot u_f) + N_f(e_1 \cdot v_f) \end{pmatrix} = \begin{pmatrix} L_f & M_f \\ M_f & N_f \end{pmatrix} \begin{pmatrix} e_1 \cdot u_f \\ e_1 \cdot v_f \end{pmatrix} = \begin{pmatrix} (n_3 - n_2) \cdot u_f \\ (n_3 - n_2) \cdot v_f \end{pmatrix},$$
$$\begin{pmatrix} L_f(e_2 \cdot u_f) + M_f(e_2 \cdot v_f) \\ M_f(e_2 \cdot u_f) + N_f(e_2 \cdot v_f) \end{pmatrix} = \begin{pmatrix} L_f & M_f \\ M_f & N_f \end{pmatrix} \begin{pmatrix} e_2 \cdot u_f \\ e_2 \cdot v_f \end{pmatrix} = \begin{pmatrix} (n_1 - n_3) \cdot u_f \\ (n_1 - n_3) \cdot v_f \end{pmatrix},$$
$$\begin{pmatrix} L_f(e_3 \cdot u_f) + M_f(e_3 \cdot v_f) \\ M_f(e_3 \cdot u_f) + N_f(e_3 \cdot v_f) \end{pmatrix} = \begin{pmatrix} L_f & M_f \\ M_f & N_f \end{pmatrix} \begin{pmatrix} e_3 \cdot u_f \\ e_3 \cdot v_f \end{pmatrix} = \begin{pmatrix} (n_2 - n_1) \cdot u_f \\ (n_2 - n_1) \cdot v_f \end{pmatrix}.$$

These equations can be rewritten as a linear system

$$\begin{pmatrix} e_1 \cdot u_f & e_1 \cdot v_f & 0 \\ 0 & e_1 \cdot u_f & e_1 \cdot v_f \\ e_2 \cdot u_f & e_2 \cdot v_f & 0 \\ 0 & e_2 \cdot u_f & e_2 \cdot v_f \\ e_3 \cdot u_f & e_3 \cdot v_f & 0 \\ 0 & e_3 \cdot u_f & e_3 \cdot v_f \end{pmatrix} \begin{pmatrix} L_f \\ M_f \\ N_f \end{pmatrix} = \begin{pmatrix} (n_3 - n_2) \cdot u_f \\ (n_3 - n_2) \cdot v_f \\ (n_1 - n_3) \cdot u_f \\ (n_1 - n_3) \cdot v_f \\ (n_2 - n_1) \cdot u_f \\ (n_2 - n_1) \cdot v_f \end{pmatrix}.$$

It is trivial to solve the above for L_f, M_f, N_f using a least squares method on a computer. Code for this process can be found in Appendix Subsection A.5.

The second fundamental form has been found in terms of basis vectors lying on the face of each triangle. However, we see the surface actually changing shape at the vertices, so we want to find the second fundamental form $\mathrm{I\!I}_p$ in terms of a basis $\{u_p, v_p\}$ at each vertex of the triangle, perpendicular to the vertex normal n_p. Given $\mathrm{I\!I}_p$ for one vertex of a triangle, we can average it with contributions from all triangles sharing that vertex to give the second fundamental form at the vertex itself.

The vertex coordinate system can be calculated by taking the cross product of n_p with an arbitrary unit vector (with different direction to n_p) to find u_p, and $n_p \times u_p$ then gives v_p. Since we are only using unit vectors, v_p will naturally be a unit vector. Code to calculate the vertex coordinate system can be found in Appendix Subsection A.4.

Trying to compute the second fundamental form using the coordinate system at each vertex causes problems, since $\mathrm{I\!I}_f$ only accepts vectors on the triangle's face as inputs, but $\{u_p, v_p\}$ is not necessarily in that subspace. Two cases must therefore be considered: when the face and vertex normals are equal, and when they are not.

When the face and vertex normals are equal, then the planes spanned by $\{u_f, v_f\}$ and $\{u_p, v_p\}$ are coplanar. So a usual basis change can be used to to write u_p and v_p in the $\{u_f, v_f\}$ coordinate system. That is $u'_p = (u_p \cdot u_f, u_p \cdot v_f)$. Since

$$L_p = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} L_p & M_p \\ M_p & N_p \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

in the $\{u_p, v_p\}$ coordinate system, it follows that

$$L_p = (u'_p)^T \mathrm{I\!I}_f u'_p = \begin{pmatrix} u_p \cdot u_f \\ u_p \cdot v_f \end{pmatrix}^T \mathrm{I\!I}_f \begin{pmatrix} u_p \cdot u_f \\ u_p \cdot v_f \end{pmatrix}$$

in the $\{u_f, v_f\}$ coordinate system. Similarly, in the coordinate system of the face, $M_p = (u'_p)^T \mathrm{I\!I}_f v'_p$ and $N_p = (v'_p)^T \mathrm{I\!I}_f v'_p$ [16].

It is more likely that the face and vertex normals are not equal. In this case, the face and vertex coordinate systems lie on completely different planes. There is no way to write the $\{u_p, v_p\}$ basis in terms of the basis of the face, thus $\mathrm{I\!I}_f$ is not at all meaningful. However, if the basis at the vertex is first rotated onto the plane defined by the triangle, then $\mathrm{I\!I}_f$ can be applied to u_r and v_r, our rotated basis vectors. The change of basis detailed above can be performed to find $\mathrm{I\!I}_p$.

To rotate a vector u around an axis a and through an angle θ, the Rodrigues formula

$$u_r = u \cos\theta + (a \times u) \sin\theta + a(a \cdot u)(1 - \cos\theta)$$

can be used. The proof is beyond the scope of this essay and can be found in [11]. Applying the Rodrigues formula to u_p and v_p with rotation axis $n_p \times n_f$ and angle $\theta = \arccos(n_p \cdot n_f)$ gives u_r and v_r as desired. Code for rotating the coordinate system is given in Appendix Subsection A.6.

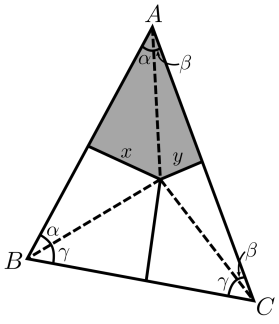


Figure 9: Voronoi region of an acute triangle

We have now found the second fundamental form at each of the vertices of each triangle in our mesh. Since more than one triangle shares each vertex, the data needs to be interpolated. This can be done using Voronoi areas as weights. The Voronoi area of a vertex is the area of the triangle closest to a vertex [14, pg.43]. This is given by the perpendicular bisectors of the two sides adjacent to the vertex. For the triangle ABC in Figure 9, the Voronoi region is shaded in grey, its area is $A_V = \frac{1}{4}(|AB| \cdot x + |AC| \cdot y)$, where $|AB|$ denotes the length of side AB. Using a bit of trigonometry,

$$x = \frac{1}{2}|AB| \tan(\alpha) \quad \text{and} \quad y = \frac{1}{2}|AC| \tan(\beta). \tag{10}$$

Since, $2(\alpha + \beta + \gamma) = \pi$, it follows that

$$\alpha = \frac{\pi}{2} - \angle C \quad \text{and} \quad \beta = \frac{\pi}{2} - \angle B. \tag{11}$$

Combining the equations in (10) and (11) gives

$$x = \frac{1}{2}|AB| \cot(\angle C) \quad \text{and} \quad y = \frac{1}{2}|AC| \cot(\angle B).$$

Hence, $A_V = \frac{1}{8}\big(|AB|^2 \cot(\angle C) + |AC|^2 \cot(\angle B)\big)$.

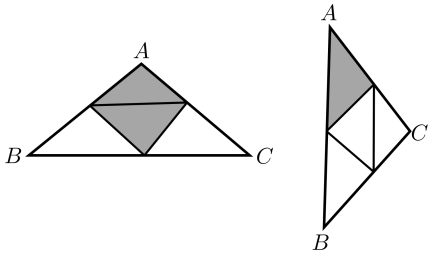


Figure 10: Voronoi regions for obtuse triangles

For an obtuse triangle, the perpendicular bisectors of the sides adjacent to the obtuse angle intersect outside of the triangle, so the midpoint of the edge opposite the obtuse angle is used instead of the intersection of the perpendicular bisectors [14, pg.43]. Figure 10 shows the Voronoi regions for obtuse triangles.

Proposition 3.1. *Connecting the midpoints of the sides of a triangle divides the triangle into four congruent triangles.*

The proof is extraneous and can be found in [3, pg.18]. By Proposition 3.1, if the angle at the vertex we are considering is obtuse then $A_V = \frac{1}{2}A_T$, where A_T is the total area of the triangle. Otherwise, $A_V = \frac{1}{4}A_T$.

Let $T(i)$ denote the triangles adjacent to vertex v_i. Let the second fundamental form at the vertex v_i of triangle $t \in T(i)$ have be the matrix $\mathrm{I\!I}_{t,i}$. The weighted version of the second fundamental form at v_i is given by

$$\mathrm{I\!I}_i = \frac{\sum_{t \in T(i)} A_{V_{t,i}} \mathrm{I\!I}_{t,i}}{\sum_{t \in T(i)} A_{V_{t,i}}},$$

where $A_{V_{t,i}}$ is the Voronoi weight for vertex i of triangle $t \in T(i)$. Code for the calculating the second fundamental form at each vertex is included in Appendix Subsection A.7.

By Corollary 2.4 the eigenvalues of $\mathrm{I\!I}_i$ give the principal curvatures and the eigenvectors give the principal directions at vertex v_i. The code for the final curvature computations is given in Appendix Subsection A.8.

4 Results

To test the method detailed above, we take the (smooth) surfaces of a torus and a sphere and calculate their curvatures everywhere. These values are then compared with values obtained by running code over more and more refined meshes of these surfaces. The vertices of the meshes coincide with points lying on the corresponding smooth surface. Meshes for the sphere and torus were obtained from the gptoolbox library [6].

4.1 Results on a Sphere

A sphere of radius 1 parametrised by $\sigma(u, v) = (\cos u \sin v, \sin u \sin v, \cos v)$ has tangent space spanned by

$$\sigma_u = (-\sin u \sin v, \cos u \sin v, 0) \quad \text{and} \quad \sigma_v = (\cos u \cos v, \sin u \cos v, -\sin v).$$

The unit normal is $N(u, v) = (-\cos u \sin v, -\sin u \sin v, -\cos v)$, with derivatives

$$N_u = (\sin u \sin v, -\cos u \sin v, 0) = -\sigma_u,$$
$$N_v = (-\cos u \cos v, -\sin u \cos v, \sin v) = -\sigma_v,$$

in the $\{\sigma_u, \sigma_v\}$ basis of the tangent space. So we can represent $-dN_p$ as the 2×2 identity matrix. This has repeated eigenvalue 1, so the Gaussian and mean curvatures are both 1 everywhere.

Figure 11 shows a graph of the maximum error plotted against the number of faces in the mesh of the sphere. Both quantities are are taken as a log to base 10 for visualisation purposes. The maximum error over all vertices decreases as the number of vertices gets larger.

A sequence (x_n) converging to x has asymptotic rate of convergence α if $\lim_{n \to \infty} \frac{|x_{n+1} - r|}{|x_n - r|^\alpha} = \lambda$ for some λ [17]. It can be approximated by

$$\alpha \approx \frac{|\log e_{n+1} / \log e_n|}{|\log e_n / \log e_{n-1}|}$$

where e_n is the error of the n^{th} term [17]. In this example, the asymptotic rate of convergence is linear, which shows that both curvatures do in fact converge to 1. Calculations can be found in Appendix Subsection B.1.

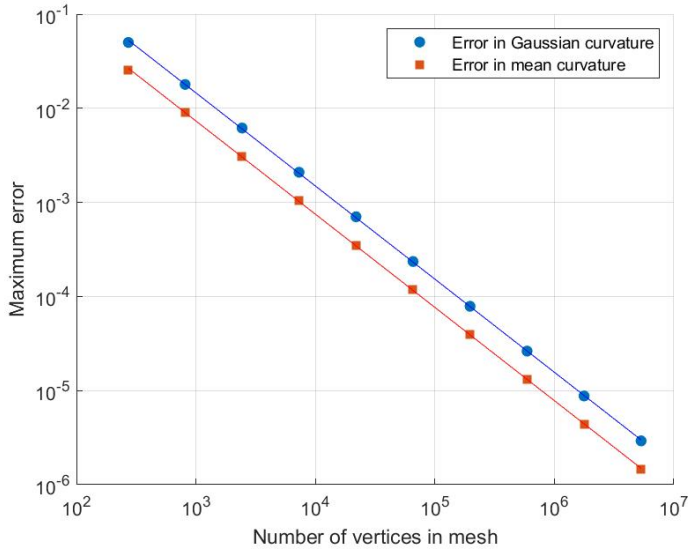


Figure 11: Absolute global error of numerical curvatures on a sphere

4.2 Results on a Torus

A torus of inner radius a and outer radius c is parametrised by

$$\sigma(u,v) = ((c + a\cos v)\cos u, (c + a\cos v)\sin u, a\sin v),$$

with a basis of the tangent space given by

$$\begin{aligned}\sigma_u &= (-(c + a\cos v)\sin u, (c + a\cos v)\cos u, 0),\\ \sigma_v &= (-a\cos u\sin v, -a\sin u\sin v, a\cos v).\end{aligned}$$

The torus has unit normal $N(u,v) = (\cos u\cos v, \sin u\cos v, \sin v)$, with

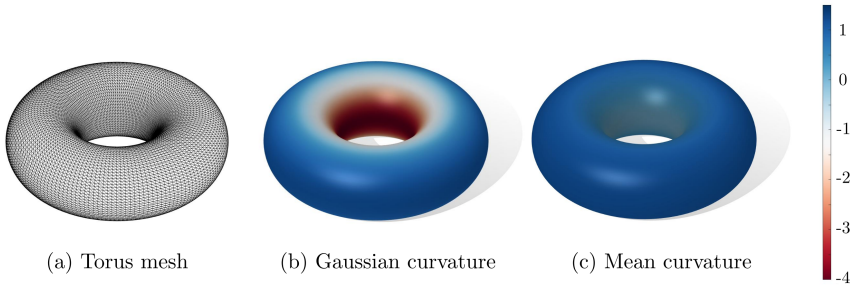
$$\begin{aligned}N_u &= (-\sin u\cos v, \cos u\cos v, 0) = -\frac{\cos v}{c + a\cos v}\sigma_u,\\ N_v &= (-\cos u\sin v, -\sin u\sin v, \cos v) = -\frac{1}{a}\sigma_v,\end{aligned}$$

in the $\{\sigma_u, \sigma_v\}$ basis of the tangent space. In this basis $-dN_p$ is represented by the matrix

$$-dN_p = \begin{pmatrix} \frac{\cos v}{c+a\cos v} & 0 \\ 0 & \frac{1}{a} \end{pmatrix}.$$

Hence, the Gaussian curvature is given by $K = \frac{\cos v}{a(c+a\cos v)}$ and the mean curvature by $H = \frac{c+2a\cos v}{2a(c+a\cos v)}$.

Figure 12 shows a plot of Gaussian and mean curvatures on a torus with inner radius 0.5 and outer radius 1, and a total of 11250 vertices. The torus mesh used was obtained from the gptoolbox library [6]. Referring back to Figure 6 we can intuitively see that the figure below accurately represents surface curvature on a torus.



(a) Torus mesh (b) Gaussian curvature (c) Mean curvature

Figure 12: Curvatures on a torus

For increasingly refined tori, we get the error plot in Figure 13. The number of vertices and maximum error have been taken as a log to base 10 for visualisation purposes. Compared to the sphere, we have much larger errors, especially for small numbers of vertices. However, we still have a linear asymptotic rate of convergence, with calculations in Appendix Subsection B.2.

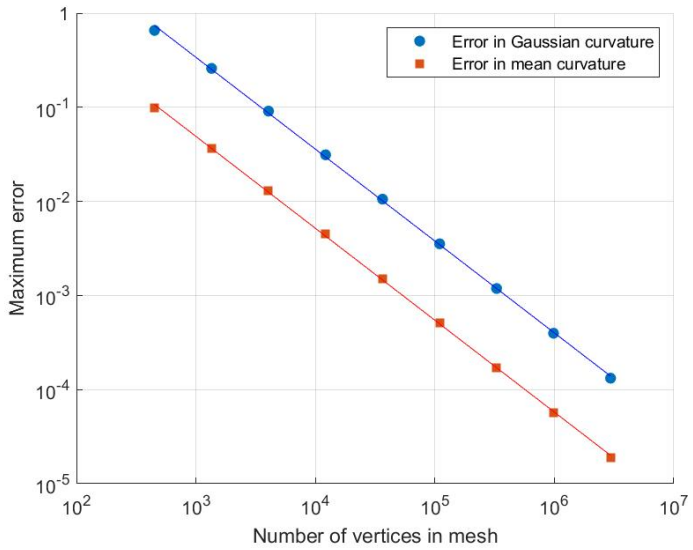
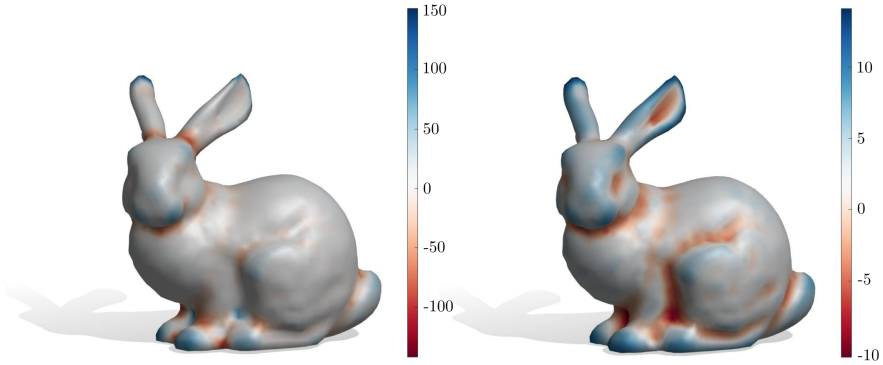


Figure 13: Absolute global error curvature approximation on torus

5 Summary

This essay has been a brief exploration of one method for calculating curvatures on smooth surfaces and its extension to triangle meshes. We implemented the algorithm given in *Estimating Curvatures and Their Derivatives on Triangle Meshes* [16] and saw that it gave a linear asymptotic rate of convergence for both the sphere and torus, making it an accurate method for approximating curvatures.

The significance of this result is that it can be applied to triangle meshes of closed surfaces that do not have an explicit formula, for example the bunny in Figure 14, original mesh obtained from The Stanford 3D Scanning Repository [8]. It is also great for visualising curvature, something that is extremely difficult with just pen and paper.



(a) Gaussian curvature (a) Mean curvature

Figure 14: Curvatures on a Stanford bunny

There are many other methods for calculating curvature on a discrete representation of a surface, each with different benefits. In fact, curvature estimation is an example of the 'no free lunch' theorem, where if algorithm A can outperform algorithm B in certain aspects, then it will fall short in others. For instance, methods such as the Gauss-Bonnet scheme are better at preserving the intrinsic structure of the original smooth surface. Other methods, such as Taubin's matrix, discretise an alternative method for calculating curvature on smooth surfaces. For a discussion of alternative methods the reader is referred to *A Comparison of Gaussian and Mean Curvature Estimation Methods on Triangular Meshes of Range Image Data* [10].

References

[1] Jakob Andreas Bærentzen, Jens Gravesen, François Anton, and Henrik Aanæs. *Guide to Computational Geometry Processing: Foundations, Algorithms, and Methods*. Springer, London, 2012.

[2] Cédric Clouchoux, Dimitri Kudelski, Ali Gholipour, Simon K. Warfield, Sophie Viseur, Marine Bouyssi-Kobar, Jean-Luc Mari, Alan C. Evans, Adre J. du Plessis, and Catherine Limperopoulos. Quantitative in vivo mri measurement of cortical development in the fetus. *Brain Structure and Function*, 217:129–139, 2012.

[3] H.S.M Coxeter and S.L Greitzer. *Geometry Revisited*. The Mathematical Association of America, Washington D.C., 1967.

[4] Manfredo P. do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, New Jersey, 1976.

[5] Peter Giblin. *Graphs, Surfaces and Homology*. Cambridge University Press, Cambridge, 3 edition, 2010.

[6] Alec Jacobson et al. gptoolbox: Geometry processing toolbox. Available at `http://github.com/alecjacobson/gptoolbox`, Accessed March 2022.

[7] Shuangshuang Jin, Robert R. Lewis, and David West. A comparison of algorithms for vertex normal computation. *The Visual Computer*, 21:71–82, 2005.

[8] Stanford Computer Graphics Laboratory. The stanford 3d scanning repository. Available at `http://graphics.stanford.edu/data/3Dscanrep/#uses`, Accessed January 2022.

[9] K. Lawonn, T. Moench, and B. Preim. Streamlines for illustrative real-time rendering. *Computer Graphics Forum*, 32(3pt3):321–330, 2013.

[10] Evgeni Magid, Octavian Soldea, and Ehud Rivlin. A comparison of gaussian and mean curvature estimation methods on triangular meshes of range image data. *Computer Vision and Image Understanding*, 107(3):139–159, 2007.

[11] Jon Mathews. Coordinate-free rotation formalism. *American Journal of Physics*, 21:1210, 1976.

[12] Nelson Max. Weights for computing vertex normals from facet normals. *Journal of Graphics Tools*, 4(2):1–6, 1999.

[13] John McCleary. *Geometry from a Differential Viewpoint*. Cambridge University Press, New York, 2 edition, 2013.

[14] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete

differential geometry: Geometry operators for triangulated 2-manifolds. In Hans-Christian Hege and Konrad Polthier, editors, *Visualization and Mathematics III*, chapter 2, pages 35–58. Springer-Verlag, Berlin, 1 edition, 2003.

[15] Barrett O'Neill. *Elementary Differential Geometry.* Academic Press, San Diego, 2 edition, 1997.

[16] Szymon Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. In *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization and Transmission*, pages 486–493, 2004.

[17] Jonathan R Senning. Computing and estimating the rate of convergence. Available at `http://www.math-cs.gordon.edu/courses/ma342/handouts/rate.pdf`, Accessed January 2022.

[18] Michael Spivak. *A Comprehensive Introduction to Differential Geometry*, volume 2. Publish or Perish, Houston, 3 edition, 1999.

[19] Dirk J. Struik. *Lectures on Classical Differential Geometry.* Dover Publications, New York, 2 edition, 1961.

[20] Yinhang Tang, Huibin Li, Xiang Sun, Jean-Marie Morvan, and Liming Chen. Principal curvature measures estimation and application to 3d face recognition. *Journal of Mathematical Imaging and Vision*, 59:211–233, 2017.

Many thanks to Silvia Sellán for all her help with writing and debugging the code for this essay.

Code was integrated into this essay using the mcode package developed by Florian Knorn, copyright (c) 2014.

Appendix

Table of Contents

A MATLAB code **19**

A.1 Calculating per-face normals 19
A.2 Calculating per-vertex normals 20
A.3 Finding coordinate system on faces 21
A.4 Finding coordinate system on vertices 22
A.5 Computing second fundamental form on faces 23
A.6 Rotating coordinate system from vertices to faces 25
A.7 Computing second fundamental form on vertices 26
A.8 Calculating principal curvatures at vertices 30

B Test functions **31**

B.1 Calculating maximum error and rate of convergence for sphere 31
B.2 Calculating maximum error and rate of convergence for torus 33
B.3 Visualisation of curvatures 36

A MATLAB code

This appendix includes the functions written to calculate curvature at the vertices of triangle meshes, based on the method provided in *Estimating Curvatures and Their Derivatives on Triangle Meshes* [16]. All code was written by the author of this essay.

A.1 Calculating per-face normals

```
function [FN] = perFaceNormals(V,F)
% INPUT:
% F - #Fx3 array of faces
% V - #Vx3 array of vertices
%
% OUTPUT:
% FN - #Fx3 array of face normals

% Find edges adjacent to first vertex of each face, with ...
    direction away
% from vertex (#Fx3 array)
e1 = V(F(:,3),:) - V(F(:,1),:);
e2 = V(F(:,2),:) - V(F(:,1),:);

% Calculate face normal as cross product of edges, then ...
    normalise (#Fx3 array)
FN = cross(e2,e1);
FN = normr(FN);

end
```

A.2 Calculating per-vertex normals

```
function [VN] = perVertexNormals(V,F,FN)
% INPUT:
% F - #Fx3 array of faces
% V - #Vx3 array of vertices
% FN - #Fx3 array of face normals
%
% OUTPUT:
% VN - #Fx3 array of vertex normals

% Find edges opposite to vertices of each face (#Fx3 arrays)
e1 = V(F(:,3),:) - V(F(:,2),:);
e2 = V(F(:,1),:) - V(F(:,3),:);
e3 = V(F(:,2),:) - V(F(:,1),:);

% Array of edge lengths squared (#Fx3 array)
e1sq = sum(e1.^2,2);
e2sq = sum(e2.^2,2);
e3sq = sum(e3.^2,2);

% Find areas of each face at each vertex of triangle (#Fx1 array)
area = 0.5*sqrt(sum(cross(e3,-e2).^2,2));

% Find the weighting for each vertex of each face (#Fx3 array)
W1 = area ./ (e2sq .* e3sq);
W2 = area ./ (e1sq .* e3sq);
W3 = area ./ (e1sq .* e2sq);
W = [W1 W2 W3];

% Calculate vertex normals (#Vx3 array)
% We do this by calculating a #Vx#F matrix where the (i,j)th ...
    entry contains
% the weighting of the ith vertex (out of all vertices) on the ...
    jth face.
% This can be done using a sparse matrix, detailed below.
% s = sparse(i,j,k,m,n) generates a m x n sparse matrix such ...
    that s(i,j)=k.
% F(:) gives a #3Fx1 matrix where the first #F entries are the first
% vertices, etc.
% repmat(1:size(F,1),1,3) gives a 1x#3F matrix where the with ...
    the numbers 1
% to #F repeated 3 times, once for each vertex.
% W(:) = gives a #3Vx1 matrix where the first #V entries are the ...
    weightings
% of the first vertex of each face etc.
W = ...
    sparse(F(:),(repmat(1:size(F,1),1,3))',W(:),size(V,1),size(F,1));
VN = normr(W*FN);

end
```

A.3 Finding coordinate system on faces

```
function [uf,vf] = faceCoordinateSystem(V,F)
% INPUT:
% F - #Fx3 array of faces
% V - #Vx3 array of vertices
%
% OUTPUT:
% uf - #Fx3 array of the first basis vector of each face
% vf - #Fx3 array of the second basis vector of each face

% We take the first basis vector as the edge opposite vetex 1 ...
    (#Fx3 array)
uf = V(F(:,3),:) - V(F(:,2),:);
uf = normr(uf);

% To find the second basis vector, we take the edge opposite ...
    vertex 3, in
% the direction away from vertex 2 and apply the Gram-Schmidt ...
    process
% (#Fx3 array)
vf1 = V(F(:,1),:) - V(F(:,2),:);
vf = vf1 - dot(uf,vf1,2) .* uf;
vf = normr(vf);

end
```

A.4 Finding coordinate system on vertices

```
function [up,vp] = vertexCoordinateSystem(V,VN)
% INPUT:
% V - #Vx3 array of vertices
% VN - #Fx3 array of vertex normals
%
% OUTPUT:
% up - #Vx3 array of the first basis vector of each vertex
% vp - #Vx3 array of the second basis vector of each vertex

% We want to find a vector perpendicular to the vertex normal, ... we can do
% this calculating the cross product between the vertex normal ... and an
% arbitrary vector, say (1,0,0). If the normal is equal to ... (1,0,0) or
% (-1,0,0) we run into an issue, if this is the case we use the ... vector
% (0,1,0) instead.
% (#Vx3 array)
v = [1,0,0];
w = [0,1,0];

% Checking the difference between normal vectors and v
diff = sum((VN - v).^2,2);
diff1 = sum((VN + v).^2,2);

% Finding the indices of the vertices where the normal is equal ... to v within
% a tolerance of 1e-3
a = find(diff < 1e-3); b = find(diff1 < 1e-3);
c = 1:size(V,1);
c([a; b]) = [];

% Creating vertices with v or w duplicated the appropriate ... number of times
% so cross product can be calculated
rep = repmat(v,size(V,1)-size(a,1)-size(b,1),1);
repa = repmat(w,size(a,1),1); repb = repmat(w,size(b,1),1);

% Constructing relevant basis vector using cross products, then ... normalising
up = zeros(size(V,1),3);
up(a,:) = cross(repa,VN(a,:),2);
up(b,:) = cross(repb,VN(b,:),2);
up(c,:) = cross(rep,VN(c,:),2);
up = normr(up);

% To find a second basis vector, calculate the cross product ... between the
% vertex normal and up (#Vx3 array)
vp = normr(cross(VN,up));

end
```

A.5 Computing second fundamental form on faces

```
function [Lf,Mf,Nf] = sffFace(V,F,VN)
% INPUT:
% F - #Fx3 array of faces
% V - #Vx3 array of vertices
% VN - #Fx3 array of vertex normals
%
% OUTPUT:
% Lf - #Fx1 array with the first component of the sff for each face
% Mf - #Fx1 array with the second component of the sff for each face
% Nf - #Fx1 array with the third component of the sff for each face

% Find edges opposite to vertices of each face (#Fx3 arrays)
e1 = V(F(:,3),:) - V(F(:,2),:);
e2 = V(F(:,1),:) - V(F(:,3),:);
e3 = V(F(:,2),:) - V(F(:,1),:);

% Find basis of each face (#Fx3 arrays)
[uf,vf] = faceCoordinateSystem(V,F);

% Find edges in the uf,vf coordinate system (#Fx3 arrays)
eu = [dot(e1,uf,2) dot(e2,uf,2) dot(e3,uf,2)];
ev = [dot(e1,vf,2) dot(e2,vf,2) dot(e3,vf,2)];

% Find directional differences in vertex normals (#Fx1 arrays)
dn1 = VN(F(:,3),:) - VN(F(:,2),:);
dn2 = VN(F(:,1),:) - VN(F(:,3),:);
dn3 = VN(F(:,2),:) - VN(F(:,1),:);

% Find directional differences in vertex normals in uf,vf ...
    coordinate system
% (#Fx3 arrays)
dnu = [dot(dn1,uf,2) dot(dn2,uf,2) dot(dn3,uf,2)];
dnv = [dot(dn1,vf,2) dot(dn2,vf,2) dot(dn3,vf,2)];

% Initialise the components of the second fundamental form as ...
    zero vectors
Lf = zeros(size(F,1),1);
Mf = zeros(size(F,1),1);
Nf = zeros(size(F,1),1);

% For each face we solve the system of equations specified in ...
    the essay
for i=1:size(F,1)
A = [eu(i,1) ev(i,1) 0;
     0 eu(i,1) ev(i,1);
     eu(i,2) ev(i,2) 0;
     0 eu(i,2) ev(i,2);
     eu(i,3) ev(i,3) 0;
     0 eu(i,3) ev(i,3)];

b = [dnu(i,1); dnv(i,1); dnu(i,2); dnv(i,2); dnu(i,3); dnv(i,3)];

% x is a vector in R^3
x = A\b;
```

```
52
53  Lf(i) = x(1);
54  Mf(i) = x(2);
55  Nf(i) = x(3);
56  end
57
58  end
```

A.6 Rotating coordinate system from vertices to faces

```
function [ur,vr] = rotateCoordinateSystem(up,vp,np,nf)
% INPUT:
% up - #Fx3 array of the first basis vector of vertex [1,2,3] on ...
    a face
% vp - #Fx3 array of the second basis vector of vertex [1,2,3] ...
    on a face
% np - #Fx3 array of the normal at vertex [1,2,3] on a face
% nf - #Fx3 array of the normal at a face adjacent to vertex [1,2,3]
%
% OUTPUT:
% ur = 1x3 array of the rotated first basis vector of the vertex
% vr = 1x3 array of the rotated second basis vector of the vertex

% Using Rodrigues' formula, the rotation around the axis a with ...
    angle
% \theta can be calculated:
% ur = u\cos\theta + (a \times u)\sin\theta + a(a \cdot ...
    u)(1-\cos\theta)
a = cross(np,nf);
cos = dot(np,nf,2);
sin = sqrt(1-cos.^2);

ur = up.*cos + cross(a,up).*sin + a.*(dot(a,up,2)).*(1-cos);
vr = vp.*cos + cross(a,vp).*sin + a.*(dot(a,vp,2)).*(1-cos);

end
```

A.7 Computing second fundamental form on vertices

```
function [Lp,Mp,Np] = sffVertex(V,F,VN)
% INPUT:
% F - #Fx3 array of faces
% V - #Vx3 array of vertices
% VN - #Fx3 array of vertex normals
%
% OUTPUT:
% Lp - #Vx1 array with the first component of the sff for each ...
    vertex
% Mp - #Vx1 array with the second component of the sff for each ...
    vertex
% Np - #Vx1 array with the third component of the sff for each ...
    vertex

% Find edges opposite to vertices of each face (#Fx3 arrays)
e1 = V(F(:,3),:) - V(F(:,2),:);
e2 = V(F(:,1),:) - V(F(:,3),:);
e3 = V(F(:,2),:) - V(F(:,1),:);

% Normalise edge vectors (#Fx3 arrays)
e1_norm = normr(e1);
e2_norm = normr(e2);
e3_norm = normr(e3);

% Array of edge lengths squared (#Fx1 array)
e1sq = sum(e1.^2,2);
e2sq = sum(e2.^2,2);
e3sq = sum(e3.^2,2);

% Find areas of each face at each vertex of triangle (#Fx1 array)
area = 0.5*sqrt(sum(cross(e3,-e2).^2,2));

% Calculate cosine of angles of each triangle
cosine = [dot(e2_norm,-e3_norm,2) ...
          dot(e1_norm,-e3_norm,2) ...
          dot(e1_norm,-e2_norm,2)];

% Calculate sine of angles of each triangle
x = sqrt(sum(cross(e2_norm,-e3_norm).^2,2));
y = sqrt(sum(cross(e1_norm,-e3_norm).^2,2));
z = sqrt(sum(cross(e1_norm,-e2_norm).^2,2));
sine = [x y z];

% Calulate cotangent of angles of each triangle
cotan = cosine./sine;

% Find maximum cosine of each face, subtracting pi/2 and then ...
    setting all
% positive values to 1 and all negative values to 0 gives an ...
    indicator for
% which faces are obtuse
maxcosine = max(cosine,[],2);
maxdiff = maxcosine - (pi/2);
a = find(maxdiff≤0);
```

```
b = find(maxdiff>0);

diff = cosine - (pi/2);
c1 = find(diff(:,1)>0);
c2 = find(diff(:,2)>0);
c3 = find(diff(:,3)>0);

ind1 = zeros(size(b,1),1);
ind1(c1) = 1;
ind2 = zeros(size(b,1),1);
ind2(c2) = 1;
ind3 = zeros(size(b,1),1);
ind3(c3) = 1;

% Finding voronoi weights for each face and vertex
W = zeros(size(3*F,1),1);
W(a) = 0.125 * (e2sq(a).*cotan(a,2) + e3sq(a).*cotan(a,3));
W(b) = ind1.*(area(b)/2) + ind2.*(area(b)/4) + ind3.*(area(b)/4);

W(size(F,1)+a) = 0.125 * (e1sq(a).*cotan(a,1) + ...
    e3sq(a).*cotan(a,3));
W(size(F,1)+b) = ind1.*(area(b)/4) + ind2.*(area(b)/2) + ...
    ind3.*(area(b)/4);

W(2*size(F,1)+a) = 0.125 * (e1sq(a).*cotan(a,1) + ...
    e2sq(a).*cotan(a,2));
W(2*size(F,1)+b) = ind1.*(area(b)/4) + ind2.*(area(b)/4) + ...
    ind3.*(area(b)/2);

% Sparse matrix where the (i,j)th entry gives the voronoi ...
    weighting for the
% ith vertex on the jth face (#Vx#F array)
voronoi = ...
    sparse(F(:),(repmat(1:size(F,1),1,3))',W(:),size(V,1),size(F,1));

% % The second fundamental form for each face as a matrix
[Lf, Mf, Nf] = sffFace(V,F,VN);

% Defining the two coordinate systems, uf and vf are #Fx3 arrays ...
    whilst
% up and vp are #Vx3 arrays
[uf,vf] = faceCoordinateSystem(V,F);
[up,vp] = vertexCoordinateSystem(V,VN);

% For each face, we get the coordinate frame at each of its vertices
% (#Fx3 arrays)
up1 = up(F(:,1),:);
vp1 = vp(F(:,1),:);
up2 = up(F(:,2),:);
vp2 = vp(F(:,2),:);
up3 = up(F(:,3),:);
vp3 = vp(F(:,3),:);

% Find per face normals (#Fx3 array)
nf = perFaceNormals(V,F);

% Find per vertex normals for each vertex on a face (#Fx3 arrays)
```

```
100 np = perVertexNormals(V,F,nf);
101 np1 = np(F(:,1),:);
102 np2 = np(F(:,2),:);
103 np3 = np(F(:,3),:);
104
105 % Calculate rotated coordinate systems of each face (#Fx3 arrays)
106 [ur1, vr1] = rotateCoordinateSystem(up1,vp1,np1,nf);
107 [ur2, vr2] = rotateCoordinateSystem(up2,vp2,np2,nf);
108 [ur3, vr3] = rotateCoordinateSystem(up3,vp3,np3,nf);
109
110 % Calculate Lp for each vertex on each face (#VxF array)
111 Lpinit1 = dot(ur1,uf,2).^2.*Lf + ...
        2.*dot(ur1,vf,2).*dot(ur1,uf,2).*Mf ...
112             + dot(ur1,vf,2).^2.*Nf;
113 Lpinit2 = dot(ur2,uf,2).^2.*Lf + ...
        2.*dot(ur2,vf,2).*dot(ur2,uf,2).*Mf ...
114             + dot(ur2,vf,2).^2.*Nf;
115 Lpinit3 = dot(ur3,uf,2).^2.*Lf + ...
        2.*dot(ur3,vf,2).*dot(ur3,uf,2).*Mf ...
116             + dot(ur3,vf,2).^2.*Nf;
117
118 Lpinit = [Lpinit1; Lpinit2; Lpinit3];
119
120 Lpinit = sparse(F(:),(repmat(1:size(F,1),1,3))', ...
121     Lpinit(:),size(V,1),size(F,1));
122
123 % Calculate Mp for each vertex on each face (#VxF array)
124 Mpinit1 = dot(ur1,uf,2).*dot(vr1,uf,2).*Lf + ...
        dot(ur1,vf,2).*dot(vr1,uf,2).*Mf ...
125             + dot(ur1,uf,2).*dot(vr1,vf,2).*Mf + ...
                    dot(ur1,vf,2).*dot(vr1,vf,2).*Nf;
126 Mpinit2 = dot(ur2,uf,2).*dot(vr2,uf,2).*Lf + ...
        dot(ur2,vf,2).*dot(vr2,uf,2).*Mf ...
127             + dot(ur2,uf,2).*dot(vr2,vf,2).*Mf + ...
                    dot(ur2,vf,2).*dot(vr2,vf,2).*Nf;
128 Mpinit3 = dot(ur3,uf,2).*dot(vr3,uf,2).*Lf + ...
        dot(ur3,vf,2).*dot(vr3,uf,2).*Mf ...
129             + dot(ur3,uf,2).*dot(vr3,vf,2).*Mf + ...
                    dot(ur3,vf,2).*dot(vr3,vf,2).*Nf;
130
131 Mpinit = [Mpinit1; Mpinit2; Mpinit3];
132
133 Mpinit = sparse(F(:),(repmat(1:size(F,1),1,3))', ...
134     Mpinit(:),size(V,1),size(F,1));
135
136 % Calculate Np for each vertex on each face (#VxF array)
137 Npinit1 = dot(vr1,uf,2).^2.*Lf + ...
        2.*dot(vr1,vf,2).*dot(vr1,uf,2).*Mf ...
138             + dot(vr1,vf,2).^2.*Nf;
139 Npinit2 = dot(vr2,uf,2).^2.*Lf + ...
        2.*dot(vr2,vf,2).*dot(vr2,uf,2).*Mf ...
140             + dot(vr2,vf,2).^2.*Nf;
141 Npinit3 = dot(vr3,uf,2).^2.*Lf + ...
        2.*dot(vr3,vf,2).*dot(vr3,uf,2).*Mf ...
142             + dot(vr3,vf,2).^2.*Nf;
143
144 Npinit = [Npinit1; Npinit2; Npinit3];
```

```
145
146 Npinit = sparse(F(:),(repmat(1:size(F,1),1,3))', ...
147     Npinit(:),size(V,1),size(F,1));
148
149 % Calculate matrices weighted by voronoi weights
150 Lp = sum(Lpinit.*voronoi,2)./sum(voronoi,2);
151 Mp = sum(Mpinit.*voronoi,2)./sum(voronoi,2);
152 Np = sum(Npinit.*voronoi,2)./sum(voronoi,2);
153
154 % Convert to full matrices
155 Lp = full(Lp);
156 Mp = full(Mp);
157 Np = full(Np);
158
159 end
```

A.8 Calculating principal curvatures at vertices

```
function [k1,k2,x1,x2] = findCurvature(V,up,vp,Lp,Mp,Np)
% INPUT:
% V - #Vx3 array of vertices
% up - #Vx3 array of the first basis vector of each vertex
% vp - #Vx3 array of the second basis vector of each vertex
% Lp - #Vx1 array with the first component of the sff for each ...
    vertex
% Mp - #Vx1 array with the second component of the sff for each ...
    vertex
% Np - #Vx1 array with the third component of the sff for each ...
    vertex
%
% OUTPUT:
% k1 - #Vx1 array of minimum curvature at each vertex
% k2 - #Vx1 array of maximum curvature at each vertex
% x1 - #Vx3 array principal directions of minimum curvature at ...
    each vertex
% x2 - #Vx3 array of pricipal direction of maximum curvature at ...
    each vertex

% Initialising empty arrays for the principal curvatures and ...
    directions
k1 = zeros(size(V,1),1);
k2 = zeros(size(V,1),1);
x1 = zeros(size(V,1),2);
x2 = zeros(size(V,1),2);

% Takes the second fundamental form of each face and calculates the
% maximum and minimum eigenvalues and their respective eigenvectors
for i=1:size(V,1)
    sff = [Lp(i), Mp(i);
           Mp(i), Np(i)];
    [V,D] = eig(sff);

    eigv = [D(1),D(4)];
    [k1(i), Ik1] = min(eigv,[],'all');
    [k2(i), Ik2] = max(eigv,[],'all');

    x1(i,:) = V(:,Ik1);
    x2(i,:) = V(:,Ik2);
end

% Takes the princpal directions calculated in the basis of the ...
    tangent
% space and finds them in R^3
x1 = x1(:,1).*up + x1(:,2).*vp;
x2 = x2(:,1).*up + x2(:,2).*vp;

end
```

B Test functions

This appendix contains code relevant to Section 4.

B.1 Calculating maximum error and rate of convergence for sphere

```
n = 10;
x = zeros(n,1);
gauss = zeros(n,1);
mean = zeros(n,1);

% Calculate the maximum error for spheres with an increasing ...
    number of
% vertices, gptoolbox [6] is required for the mesh
for i=1:n
[V,F] = subdivided_sphere(i+2,'SubdivisionMethod','sqrt3');

FN = perFaceNormals(V,F);
VN = perVertexNormals(V,F,FN);
[uf,vf] = faceCoordinateSystem(V,F);
[up,vp] = vertexCoordinateSystem(V,VN);

% Calculate second fundamental form on vertices
[Lp,Mp,Np] = sffVertex(V,F,VN);

% Calculate curvatures on vertices
[k1,k2,x1,x2] = findCurvature(V,up,vp,Lp,Mp,Np);

% Calculate Gaussian curvature
K = k1 .* k2;

% Calculate mean curvature
H = 0.5 * (k1 + k2);

% Find error between the theoretical and calculated Gaussian and ...
    mean
% curvature values
x(i) = size(V,1);

errgauss = abs(1 - K);
errmean = abs(1 - H);

gauss(i) = max(errgauss);
mean(i) = max(errmean);
end

% Plot log log graph of maximum error against number of vertices
figure1 = figure;
hold on;
grid on;

xlim([2 7])
xticks([2 3 4 5 6 7])
```

```
xticklabels({'10^2','10^3','10^4','10^5','10^6','10^7'})

ylim([-6,-1])
yticks([-6 -5 -4 -3 -2 -1])
yticklabels({'10^{-6}','10^{-5}','10^{-4}','10^{-3}','10^{-2}', ...
    '10^{-1}'})

xlabel('Number of vertices in mesh');
ylabel('Maximum error') ;

% Plot lines of best fit
x1 = log10(x);
y1 = log10(gauss);
y2 = log10(mean);

p1 = polyfit(x1,y1,1);
p2 = polyfit(x1,y2,1);

f1 = polyval(p1,x1);
f2 = polyval(p2,x1);

% Final plots
scatter(log10(x),log10(gauss),'filled');
scatter(log10(x),log10(mean),'s','filled');
plot(x1,f1,'b');
plot(x1,f2,'r');
legend({'Error in Gaussian curvature','Error in mean curvature'});

saveas(figure1,'sphere.jpg');

% Calculating asymptotic rates of convergence
eocgauss = zeros(n-2,1);
for i = 1:n-2
    eocgauss(i) = log(gauss(i+2)./gauss(i+1)) ./ ...
        log(gauss(i+1)./gauss(i));
end

eocmean = zeros(n-2,1);
for i = 1:n-2
    eocmean(i) = log(mean(i+2)./mean(i+1)) ./ ...
        log(mean(i+1)./mean(i));
end
```

Table 1: Rate of convergence at each plot point for sphere

Plot point	Rate of convergence (Gaussian)	Rate of convergence (mean)
3	1.0359	1.0305
4	1.0153	1.0135
5	1.0073	1.0067
6	1.0036	1.0034
7	1.0019	1.0019
8	1.0010	1.0010
9	1.0006	1.0006
10	1.0003	1.0003

The table above shows the rate of convergence at plot points. The fact that the values converge to 1 suggests a linear asymptotic rate of convergence.

B.2 Calculating maximum error and rate of convergence for torus

```
n = 9;
x = zeros(n,1);
gauss = zeros(n,1);
mean = zeros(n,1);

% Calculate the maximum error for tori with an increasing number of
% vertices,  gptoolbox [6] is required for the mesh
for i = 1:n
[V,F] = ...
    torus(round(10*sqrt(3)^(i+1)),round(5*sqrt(3)^(i+1)),0.5,'R',1);

FN = perFaceNormals(V,F);
VN = perVertexNormals(V,F,FN);
[uf,vf] = faceCoordinateSystem(V,F);
[up,vp] = vertexCoordinateSystem(V,VN);

% Calculate second fundamental form on vertices
[Lp,Mp,Np] = sffVertex(V,F,VN);

% Calculate curvatures on vertices
[k1,k2,x1,x2] = findCurvature(V,up,vp,Lp,Mp,Np);

% Calculate Gaussian curvature
K = k1 .* k2;

% Calculate mean curvature
H = 0.5 * (k1 + k2);

% Find error between the theoretical and calculated Gaussian and ...
    mean
% curvature values. Since we only have (x,y,z) coordinates and ...
    not the
% surface parameters (u,v), we need to work backwards to obtain ...
    v. We do
% not need the value of u since it is not included in curvature
% calculations. We can calculate the sine of v using the z ...
    coordinate. If
% we try to calculate v using this, we get only the principal ...
    values. For
% positive sin(v), we could have v or (pi-v), for negative ...
    sin(v), we could
% have (pi-v) or (2pi+v) since v will be negative. We also know that
% (1+0.5cosv)^2 = x^2 + y^2 by the equation for a torus. So for ...
    sin(v)
% positive and negative we can substitute in the possible values ...
    of v and
% test which are closese to x^2 + y^2. This will give us the v ...
    that we can
% do error analysis on.
x(i) = size(V,1);
sinv = 2.*V(:,3);
v = asin(sinv);
```

```
v1 = pi - v;
v2 = 2*pi + v;

ind = find(sinv > 0);
ind1 = find(sinv < 0);
ind2 = find(sinv == 0);

pos1 = V(ind,1).^2 + V(ind,2).^2 - (1 + 0.5.*cos(v(ind))).^2;
pos2 = V(ind,1).^2 + V(ind,2).^2 - (1 + 0.5.*cos(v1(ind))).^2;
pos = [pos1 pos2];
[¬, j] = min(abs(pos),[],2);
a = find(j == 2);
v(ind(a)) = v1(ind(a));

neg1 = V(ind1,1).^2 + V(ind1,2).^2 - (1 + 0.5.*cos(v1(ind1))).^2;
neg2 = V(ind1,1).^2 + V(ind1,2).^2 - (1 + 0.5.*cos(v2(ind1))).^2;
neg = [neg1 neg2];
[¬, k] = min(abs(neg),[],2);
y = find(k == 1);
z = find(k == 2);
v(ind1(y)) = v1(ind1(y));
v(ind1(z)) = v2(ind1(z));

zer1 = V(ind2,1).^2 + V(ind2,2).^2 - (1 + 0.5).^2;
zer2 = V(ind2,1).^2 + V(ind2,2).^2 - (1 - 0.5).^2;
zer = [zer1 zer2];
[¬, m] = min(abs(zer),[],2);
t = find(m == 1);
u = find(m == 2);
v(ind2(t)) = 0;
v(ind2(u)) = pi;

errgauss = abs(2*cos(v)./(1 + 0.5 .* cos(v)) - K);
errmean = abs((1 + cos(v))./(1 + 0.5 .* cos(v)) - H);

gauss(i) = max(errgauss);
mean(i) = max(errmean);
end

% Plot log log graph of maximum error against number of vertices
figure1 = figure;
hold on;
grid on;

xlim([2 7])
xticks([2 3 4 5 6 7])
xticklabels({'10^{2}','10^{3}','10^{4}','10^{5}', ...
    '10^{6}','10^{7}'})

ylim([-5 0])
yticks([-5 -4 -3 -2 -1 0])
yticklabels({'10^{-5}','10^{-4}','10^{-3}','10^{-2}', ...
    '10^{-1}','1'})
xlabel('Number of vertices in mesh');
ylabel('Maximum error');

% Plot lines of best fit
```

```
x1 = log10(x);
y1 = log10(gauss);
y2 = log10(mean);

p1 = polyfit(x1,y1,1);
p2 = polyfit(x1,y2,1);

f1 = polyval(p1,x1);
f2 = polyval(p2,x1);

% Final plots
scatter(log10(x),log10(gauss),'filled');
scatter(log10(x),log10(mean),'s','filled');
plot(x1,f1,'b');
plot(x1,f2,'r');
legend({'Error in Gaussian curvature','Error in mean curvature'});

saveas(figure1,'torus1.jpg');

% Calculating asymptotic rates of convergence
eocgauss = zeros(n-2,1);
for i = 1:n-2
    eocgauss(i) = log(gauss(i+2)./gauss(i+1)) ./ ...
        log(gauss(i+1)./gauss(i));
end

eocmean = zeros(n-2,1);
for i = 1:n-2
    eocmean(i) = log(mean(i+2)./mean(i+1)) ./ ...
        log(mean(i+1)./mean(i));
end
```

Table 2: Rate of convergence at each plot point for torus

Plot point	Rate of convergence (Gaussian)	Rate of convergence (mean)
3	1.1255	1.0625
4	1.0218	1.0312
5	1.0149	1.0108
6	1.0073	1.0095
7	1.0007	1.0009
8	1.0017	1.0023
9	1.0028	1.0032

The table above shows the rate of convergence at plot points. The fact that the values converge to 1 suggests a linear asymptotic rate of convergence.

B.3 Visualisation of curvatures

```
% Set [V,F] as mesh of your choosing, gptoolbox [6] is required ...
    for the
% meshes below
% [V,F] = subdivided_sphere(5,'SubdivisionMethod','sqrt3');
% [V,F] = torus(150,75,0.5,'R',1);

% Initial data
FN = perFaceNormals(V,F);
VN = perVertexNormals(V,F,FN);
[uf,vf] = faceCoordinateSystem(V,F);
[up,vp] = vertexCoordinateSystem(V,VN);

% Calculate second fundamental form on vertices
[Lp,Mp,Np] = sffVertex(V,F,VN);

% Calculate curvatures on vertices
[k1,k2,x1,x2] = findCurvature(V,up,vp,Lp,Mp,Np);

% Calculate Gaussian curvature
K = k1 .* k2;

% Calculate mean curvature
H = 0.5 * (k1 + k2);

% Visualisation of Gaussian and mean curvatures on mesh, ...
    uncomment as
% required; gptoolbox [6] is required for tsurf to run.
t = tsurf(F,V,'FaceColor','interp','Cdata', H, fsoft, fphong);
% t = tsurf(F,V,'FaceColor','interp','Cdata', H, fsoft, fphong);
% caxis([-100 100]);
% caxis([-4 1.5]);
axis equal;
colorbar;
colormap(cbrewer('RdBu',1000));
set(t, 'DiffuseStrength',0.5, 'SpecularStrength',0.2, ...
    'AmbientStrength',0.3);
t.EdgeColor = 'None';
set(gca,'visible','off');
set(gca, 'CameraPosition', [ 2.7487 13.231 2.7613]);
l = light('Position',[-8 18 20]);
set(gcf,'Color',[1,1,1]);
s = add_shadow(t,l,'Color',[1 1 1]*0.8,'BackgroundColor',[1 1 ...
    1], ...
    'Fade','infinite');
```