Homework #3: NURBS surfaces; surface curvature; programming with OpenMesh [70 points]
Due Date: Wednesday, 6 March 2017

Theory and implementation

This is a mixed theory/implementation homework. Problem 1 provides some practice on calculations with NURBs surfaces (non-uniform B-splines). Problem 2 gives you chance to actually model and visualize the NURBs surface in Problem 1. Problem 3 asks you to derive the theory behind an algorithm for computing principal curvatures and directions on a meshed surface, and problem 4 asks you to implement this algorithm. As explained in the guidelines given with Homework 1, you may hand in a joint write-up for Problem 2 and 4 (implementation problems). You must still hand in individual write-ups for solutions to problems 1 and 3, however (theory problems). Recall that maximum group size is three.

A goal of this problem is also to introduce you to two standard programs for playing with surface meshes, **OpenMesh***,* `http://www.openmesh.org/` *and* **Meshlab***,* `http://meshlab.sourceforge.net/`*. You can download all helper code for this problem form the Handouts class web page.*

Problem 1. [10 points]

Let r and R be real numbers with $0 < r < R$. As the angles α and β vary, the varying point $V = (X, Y, Z)$ given by

$$\begin{aligned} X &:= (R + r\cos\alpha)\cos\beta \\ Y &:= (R + r\cos\alpha)\sin\beta \\ Z &:= r\sin\alpha \end{aligned}$$

traces out a *torus*, that is, the surface of an ideal bagel. The torus has rotational symmetry about the Z axis, which passes through the middle of the torus hole. Any plane π through the Z axis cuts the torus in two circles of radius r, whose centers lie along the line where π cuts the XY plane, R units on each side of the origin. Varying α moves the point V around one of those circles. Varying β rotates the plane π around the Z axis.

Find a biquadratic, rational parameterization of this torus. That is, express each of the homogeneous coordinates $[w; x, y, z]$ of the varying point V as a polynomial in two parameters that has degree at most 2 in each parameter when the other is held fixed. For consistency in notation, use Q and T as your two parameters, writing

$$V(Q;T) = [w(Q;T); x(Q;T), y(Q;T), z(Q;T)]$$

for certain polynomials w, x, y, and z. Hint: Let $Q := \tan(\alpha/2)$ and $T := \tan(\beta/2)$.

Homogenize and polarize your parameterization. You may carry out these two steps in either order; the result will be the same. If you homogenize first, use p as the weight coordinate for the Q parameter, writing $Q = q/p$, and use s as the weight coordinate for the T parameter, writing $T = t/s$. If you polarize first, split the T parameter into two separate parameters T_1 and T_2, and split the Q parameter into Q_1 and Q_2. The homogenized polar form v of V will have the form

$$v((p_1;q_1),(p_2;q_2);(s_1;t_1),(s_2;t_2)) = [w;x,y,z],$$

where each coordinate w, x, y, and z is a polynomial in the eight variables p_1, q_1, p_2, q_2, s_1, t_1, s_2, and t_2.

One quarter of the torus consists of points that have Y and Z positive. (If you followed the hint above, those points will correspond, under your parametrization, to parameter pairs $(Q;T)$ where both Q and T are positive.) Describe this portion of the torus as a biquadratic, rational Bézier surface patch, that is, as a rectangular, tensor-product patch of degree $(2;2)$. In particular, give the coordinates of the nine Bézier sites of this patch. (If you followed the hint above, the patch will be $V([0\,..\,\infty] \times [0\,..\,\infty])$.) Hint: Don't be distressed if one of the nine Bézier sites turns out to be the zero site, that is, the site all four of whose coordinates are zero.

Problem 2. [15 points]

In this problem, we will use the surface from Problem 1 to model a torus using OpenMesh. The file you will edit is called `main-torus.cpp` in the `RenderTorus` folder provided.

We have provided a bare-bones 3D viewing tool as part of your starter code. The controls are as follows:

- Dragging the left mouse button rotates the coordinate frame
- Dragging the middle mouse button pans the camera
- Dragging the right mouse button zooms

We have also provided code for rendering your mesh as wireframe, so that you can focus on the shape modeling and not have to worry about orientations and normals.

2(a). [Evaluating a point on the surface, 5 points] We start with only the quarter of the torus from Problem 1, which we modeled using a biquadratic tensor-product surface. First, fill in the code section labeled STUDENT CODE SECTION 1 with the control points you found in Problem 1. Recall that we can evaluate a point using the Bézier sites by interpolating along the Q and T directions separately. Given a parameter value of $u \in [0,1]$ in the Q direction and $v \in [0,1]$ in the T direction, fill in the function $Calculate(u,v,k)$ in STUDENT CODE SECTION 2, which should return the homogeneous coordinates for a point on the surface (k is the index for the quarter of the torus we are modeling; for now just set it to 0).

2(b). [Construct a mesh. 5 points] We have defined a set of points on the surface patch and now we want to connect them to construct a mesh. This can be done using OpenMesh. In `generateMesh()`, as we evaluate points on the surface, we insert them into the mesh as vertices and store a 2D array of their associated *vertex handles*. Fill in STUDENT CODE SECTION 3 to construct a face by listing the face's vertex handles and then inserting the face into the mesh. Once you set up the vertices and faces correctly, OpenMesh will take care of the rest for you (i.e. connectivity between faces). In addition, OpenMesh supports both triangle meshes and more general polygon meshes, so you can use whichever you prefer for this problem (quad meshes are a natural choice for tensor product surfaces).

2(c). [Time for a whole torus, 5 points] Now you should have successfully rendered a quarter of the torus. Determine the control points for the other three quarters (using either intuition and symmetry or explicit calculation) and repeat the same approach to construct the rest of the torus. You can do this by varying the k parameter in the code. For full credit, you can simply construct the four parts separately and put them all in one big mesh. However, note that the vertices on the boundaries will have duplicates in the mesh data structure (two if they connect two quarters of the torus and four if they connect all four). Therefore, the quarters are internally disconnected on the boundaries even though they appear to connect on display.

2(d). [Extra Credit, 10 points]

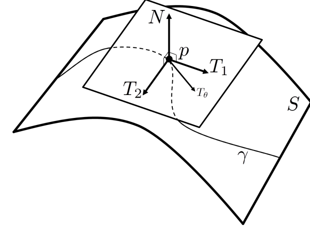
- The program saves your mesh into a file `torus.off`, so you can load and view the mesh using other software (e.g., MeshLab) or the program for Problem 4. Construct the mesh with correct orientation, so that it can be rendered correctly with lighting. [5 points]

- Stitch the quarters of the torus together to address the vertex duplication problem described above. You can use MeshLab or the executable from Problem 4 to view the mesh with lighting. You should see cusps along the boundaries of the quarters initially (using smooth shading). Your task is to get rid of these cusps. [5 points]

Problem 3. [25 points]

In the remainder of the homework, you'll derive the theory behind the paper titled "Estimating the Tensor of Curvature of a Surface from a Polyhedral Approximation," by Gabriel Taubin (available from the course schedule web page), and then implement the algorithm it describes using OpenMesh. You are welcome to read that paper and use his development to guide your answers to the homework problems, but you must show your work for each part.

In a past lecture we explored the differential geometry of space curves and showed how curvature κ and torsion τ can be used to characterize their shapes. Just as we extended our constructions of curves to yield methods for designing surfaces, so we can make use of notions from the differential geometry of curves to understand the shapes of surfaces.



In particular, suppose we have a patch of a surface S as shown on the right, and take a point $p \in S$. While we haven't yet developed a formal way to understand how S bends, we could draw a curve γ along S through p and use the curvature of γ as a proxy for understanding the curvature of S.

Just as the Frenet frame allowed us to organize our understanding of shape for curves, we can separate out the curvature vector k of γ into two components: one parallel to the surface normal N (not necessarily the normal to γ!) and one perpendicular to N. Although we will not ask you to prove them here, the parallel or "normal component" $k \cdot N$ of the curvature vector of γ at p satisfies a number of amazing properties:

- It depends only on the tangent direction of γ at p.
- The directions of maximum and minimum normal curvature, which we will call T_1 and T_2, must be orthogonal. We'll call the curvatures in these two directions κ_1 and κ_2, resp.
- For a general tangent direction $T_\theta = T_1 \cos\theta + T_2 \sin\theta$, the curvature in the T direction satisfies the relationship
$$\kappa_\theta = \kappa_1 \cos^2\theta + \kappa_2 \sin^2\theta.$$

With these basic facts in hand, we can begin to derive Taubin's classic algorithm for finding T_1, T_2, κ_1, and κ_2, the so-called "principal directions" and "principal curvatures."

3(a). [5 points] Define the tangent vector $T_\theta = T_1 \cos\theta + T_2 \sin\theta$ and the matrix

$$M = \frac{1}{2\pi} \int_{-\pi}^{\pi} \kappa_\theta T_\theta T_\theta^\top \, d\theta.$$

Show that N is in the *kernel* of M; that is, show that the surface normal N is an eigenvector of M with eigenvalue 0.

3(b). [5 points] Show that $T_1^\top M T_2 = 0$. Notice that an identical argument will show that $T_2^\top M T_1 = 0$. *[Hint: Recall that T_1 and T_2 are orthogonal.]*

3(c). [5 points] Show that $T_1^\top M T_1 = \frac{3}{8}\kappa_1 + \frac{1}{8}\kappa_2$. By symmetry this must imply that $T_2^\top M T_2 = \frac{1}{8}\kappa_1 + \frac{3}{8}\kappa_2$.

3(d). [5 points] Show how to find the principal curvatures κ_1 and κ_2 from the values $m_{11} = T_1^\top M T_1$ and $m_{22} = T_2^\top M T_2$. Furthermore, argue that T_1 and T_2 are eigenvectors of M with eigenvalues m_{11} and m_{22}, resp.

Effectively, the facts above show that if we can estimate the matrix M at a point p on a surface, we easily can figure out the principal directions and curvatures of the surface at p.

Next we will develop our tool for approximating M. Suppose at point $p \in S$ we choose a tangent direction T. Draw a curve $\gamma(s)$ along the surface through p such that $\gamma(0) = p$ and $\gamma'(0) = T$. We can approximate γ as

$$\gamma(s) \approx \gamma(0) + \gamma'(0)s + \frac{1}{2}\gamma''(0)s^2.$$

3(e). [5 points] Use our discussion of normal curvature and the approximation of $\gamma(s)$ above to derive the relationship

$$\kappa(T) \approx \frac{2N^\top(\gamma(s) - p)}{\|\gamma(s) - p\|^2},$$

where $\kappa(T)$ is the normal curvature of the surface at p in the T direction. As an aside, a slightly more formal development of this approximation shows

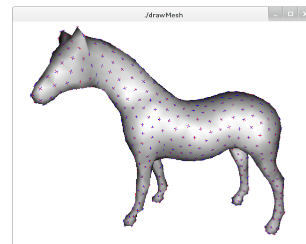
$$\kappa(T) = \lim_{s \to 0} \frac{2N^\top(\gamma(s) - p)}{\|\gamma(s) - p\|^2}.$$

[Hint: Assume terms cubic or higher in s can be ignored. Take the inner product of our approximation of γ with N and also find an approximation of $\|\gamma(s) - p\|^2$.]

Problem 4. [20 points]

Now that you understand the theory behind Taubin's algorithm, it is time to implement it. We will be using the C++ programming language equipped with the OpenMesh and Eigen libraries for loading meshed geometry and doing linear algebra, resp. We will spend section in recitation introducing these libraries and are happy to answer questions you might have about them during office hours and on Piazza.

Computing Principal Directions and Curvatures



Now you will apply your mesh navigation skills to computing the principal curvatures and directions of an input mesh using the Taubin algorithm we discussed in the previous problem. Your job is to complete the `computeCurvature` function in `src/curvature.cpp`. In particular, you should set the `curvature` property of each vertex with the appropriate `CurvatureInfo` struct (as defined in `include/curvature.h`).

For convenience, we recommend using the Eigen library for linear algebra rather than OpenMesh's more limited linear algebra capabilities. To do so, you'll have to convert from OpenMesh's `Vec3f` to Eigen's `Vector3d` at the beginning of your inner loop and back at the end — an example is included in the code.

See Section 4 of Taubin's paper for the particular algorithm you should implement. Although you are welcome to implement the algorithm exactly as described, we'll allow for a few simplifications:

- You can use the `mesh.normal(vertexHandle)` method to get the per-vertex normal N_{v_i} for vertex v_i.
- Once you compute the matrix $\tilde{M}_{v_i}$, you can use Eigen's 3×3 matrix eigenvector methods to find E_1 and E_2 rather than using the Householder/Givens technique described.
- Once you've computed the principal directions, fill the variable `info` in the code accordingly.

What to hand in

To submit your code, please first read `README` file in the given code. Make a `.tar` or `.zip` archive of all your files (except library files) and e-mail it to the course CA. Be sure to include your own (replaced) `README` file that gives the names of your group members and an index to your program files and functions. In addition, please hand in with your paper-and-pencil homework a short write-up about how your code works.

Recall that teams of up to three students are allowed to work as a group in solving homework problems and a single submission is allowed for the programming problems.