CGGS Coursework 2 (Geometry): Discrete Analysis and Parameterization

Amir Vaxman, Thomas M. Walker

Errata

- None.

Introduction

This is the 2^{nd} coursework for the "geometry" choice. The purpose is to write Python code for obtaining and visualizing the curvatures of a triangle mesh, and work with the Laplace operator (Laplacian) for further mean-curvature flow and convex Tutte mesh parameterization. This practical mostly follows the material of Lectures 10–12. Visualization will be done with PolyScope as in Practical 1. The concrete objectives are:

- Compute and visualize Gaussian and mean curvatures on a triangle mesh.
- Compute the Laplacian operator and the Voronoi areas.
- Use the Laplacian to flow mean curvature so as to smooth the surface.
- Use the Laplacian to do Tutte UV parameterization and visualize it as texture mapping.
- *Advanced:* compute and visualize principal curvatures and directions.

General guidelines

Please update to Python 3.11; some features of PolyScope depend on it.
The practical uses the same Python + PolyScope setup as in Practical 1. In addition, we use the `scipy.sparse` for handling sparse linear algebra.

All the functionality should be filled into the signatures of `DAFunctions.py`. Note that *all* `TODO` functions need to be properly filled so that the graders can operate correctly. However, every section will use different visualization scripts, as per the below. These scripts don't get graded, so you can modify them as you wish.

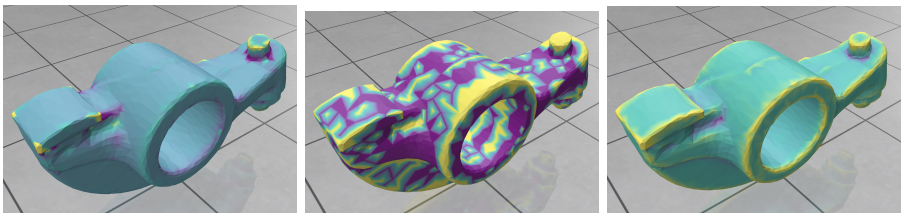

Figure 1: Left to right: Gaussian curvature (sensitivity adapted), curvature regions by the sign of Gaussian curvature (see evident noise in more or less parabolic regions), depicting elliptic and hyperbolic regions, and mean curvature.

1 Computing Curvatures

This section computes three basic quantities from which we can compute curvatures: the discrete angle defect, the Voronoi areas of every vertex, and the cot-weight Laplace operator, or Laplacian (Lecture 11).

Angle defect The discrete angle defect is to be computed through the function `compute_angle_defect()`. It should return an array G of size $|V|$ (number of vertices), where it outputs:

$$G(v) = \begin{cases} 2\pi - \sum_{f \in N(v)} \alpha_{f,v} & v \in v_I \\ \pi - \sum_{f \in N(v)} \alpha_{f,v} & v \in v_B \end{cases}.$$

v_I is the set of inner vertices, and v_B is the set of boundary vertices (represented through `boundVertices`). $\alpha_{f,v}$ is the angle in face f at vertex v, and $N(v)$ is the set of all the faces adjacent to vertex v.

Voronoi area You should return an array A of size $|V|$ that represents the "control area" that belongs to each vertex v. That is:

$$A(v) = \frac{1}{3} \sum_{f \in N(i)} A(f),$$

where $A(f)$ is the area of face f. Boundary vertices are treated the same.

Cot-weight Laplacian You should compute the *integrated* Laplacian operator of the form:

$$L = d_0^T W d_0,$$

where d_0 is the *differential* matrix, the same one you computed in Practical 0. That is, d_0 is a $|E| \times |V|$ sparse matrix defined as follows:

$$d_0(i,j) = \begin{cases} -1 & j = source(e_i) \\ 1 & j = target(e_i) \\ 0 & otherwise \end{cases}$$

W is a *sparse diagonal* matrix of edge-based cot weights. For any edge $e = ik$ neighbouring on faces ijk (left) and kli (right), you should compute:

$$W(e,e) = \frac{1}{2}(cot(\alpha_{ijk}) + cot(\alpha_{kli})),$$

where α_{ijk} is the angle at vertex j, and α_{kli} is the angle at vertex l. If the edge is a boundary edge and you only have, without loss of generality, a single adjacent triangle ijk, you should produce:

$$W(e,e) = \frac{1}{2}cot(\alpha_{ijk})$$

Assembling everything both the Voronoi areas and the Laplacian should be output through `compute_laplacian()` function. It gets two helpful parameters: `edgeBoundMask`, which is an $|E|$-sized array which is 1 for boundary edges and 0 otherwise, and a $|E| \times 4$ matrix EF where the first column is the index (into `faces`) of the left face ijk, the second column is the location of the relative vertex j in ijk (either 0,1,2; for instance if the face is arranged ijk, the result is 1). The third and fourth columns of `EF` are the same for triangle kli.

Having completed both functions `compute_angle_defect()` as well as `compute_laplacian()`, the visualization script will compute the pointwise Gaussian curvature $K = \frac{G}{A(v)}$ and display it; it will further display curvature regions according to the sign of K. Note they might be quite noisy if the mesh is not entirely smooth around parabolic regions; try to play with the script to show them only above some tolerance for more insight!

To compute mean curvature, you have to implement another stage of the algorithm: complete the `compute_mean_curvature_normal()` function that obtains the cot Laplacian `L` and Voronoi areas `vorAreas` as input. The mean curvature normal formula is:

$$H\hat{n} = \frac{Lv}{2A(v)},$$

where Lv is L applied to the $|V| \times 3$ matrix of the vertex coordinates. Note: the result is then a $|V| \times 3$ matrix as well. Separating the mean curvature H from the normal $\hat{n}$ is not obvious since the sign is ambiguous. Therefore, we take the following strategy: compute independent vertex normals $n(v)$ by averaging surrounding face normals and normalizing:

$$n(v) = \frac{\sum_{f \in N(v)} n(f)}{|\sum_{f \in N(v)} n(f)|}$$

We then treat H as positive if $H\hat{n}$ and $n(v)$ are co-aligned, and negative otherwise:

$$H(v) = sign(H\hat{n} \cdot n(v)) \cdot |H|.$$

the visualization script will then show both H and $H\hat{n}$. (Figure 1).

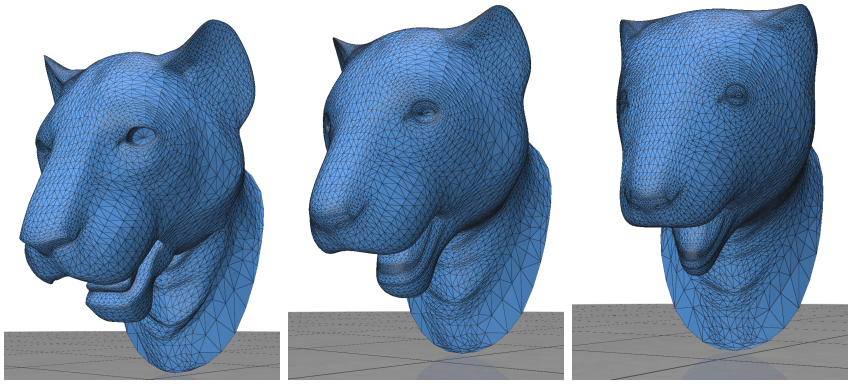

Figure 2: Left: the original mesh. Middle: after a few iterations of explicit flow with $0.1 \cdot min(A(v))$ little progress was made, but it's not possible to go faster for stability reasons. Right: with an implicit flow at $2.0 \cdot min(A(v))$, we can make far more progress in a shorter time.

Grading: Grading will be first done by a perfect result (and pro-rata wrong results) obtained from the grading script `DAGrader.py` in the `grading` folder. The automatic grader for this section will constitute 25% of your grade; an extra 10% will be graded on a report of your (concise) insights about the results you see for several meshes, with regard to your expectations. That means this section will bring you to a total of 35%.

Coding advice: use the provided function `accumarray()` (equivalent to a celebrated MATLAB function) that gets a matrix of indices and a matrix of values, and computes a new array in which the positions corresponding to the input indices get the corresponding values, where repeating indices result in summing up values. This is very helpful in aggregating angle defects and Voronoi areas from faces and corners to vertices.

2 Mean curvature flow

We next make use of the quantities you computed in the previous section for a new application: mesh smoothing by mean curvature flow. This is controlled by the script `MeanCurvatureFlow.py`. We know that Lv encodes the mean-curvature normal, which is also the direction from the perfect (weighted) average of the neighbors of a vertex v to its actual location. Thus, we aim to solve the following differential equation:

$$\frac{dv}{dt} = -\Delta v,$$

Δv being the continuous laplacian of the points, discretized by $M^{-1}L$ for a sparse diagonal mass matrix $M : |V| \times |V|$ with the Voronoi areas on the diagonal and the cot-Laplacian L. Following this equation, the vertices of the mesh keep flowing to the average of their neighbors, and the mesh appears to smooth away—for the most part. As you will see (for instance, in the `horsers` mesh), this flow can develop thin singularities. The stationary final point of this flow is when

$$\Delta v = 0,$$

meaning you will have computed a minimal surface. In case there are boundary vertices v_B, we keep them fixed (not changing with the flow). Otherwise, to avoid the mesh smoothing away to 0, we scale it uniformly (meaning multiplying all coordinates by a scalar), so that it has the original total surface area after each smoothing step (it might jitter a bit visually because of this).

Explicit Flow We will experiment with two types of time discretizations for integrating the flow. The flow should be implemented in the function `mean_curvature_flow()`, where the parameter `isExplicit=True` means you should use an explicit time integration (Lecture 7). You are then to implement a *single* explicit Euler integration step as follows:

$$v(t + \Delta t) = v(t) - \delta t \cdot M^{-1} L v(t),$$

δt is given as `flowRate`. You can avoid constructing M or M^{-1} explicitly, and equivalently divide Lv by `vorAreas` similar to what you did to get the mean curvature normal in the previous section. You will see, by playing with the visualization script, that explicit flow works well, but only for quite small flow rates δt, usually of the magnitude of no more than $\delta t = 0.1 \cdot min(A(v))$. Try to go above $\delta t = 0.5 \cdot min(A(v))$ and it will usually explode; this is the nature of explicit Euler integration.

Implicit flow Given `isExplicit=False`, we will instead do a single step of *implicit* Euler integration (Lecture 7). That means, in each invocation of `mean_curvature_flow()`, you should solve the linear system:

$$\begin{aligned} v(t + \Delta t) - v(t) &= -\delta t (M^{-1} L v(t + \Delta t)) \\ \Rightarrow (M + \delta t \cdot L) v(t + \Delta t) &= M \cdot v(t) \end{aligned}$$

to get $v(t + \Delta t)$. Use `sparse.linalg.splu`, as the left-hand size $M + \delta t L$ is symmetric (it's also positive-definite, but SciPy doesn't have native Cholesky decomposition for no apparent reason...). You will see that this is a big improvement: you can use almost unlimited δt and it will be unconditionally stable, albeit will create some artifacts when this is too large. Smoothing problems like this one are the quintessential applications for implicit time integration. See results in Figure 2. Whether explicit or implicit, remember that if the mesh has boundary vertices (check `boundVertices`), you should set them as fixed (you can do it after smoothing, for a simpler code; you don't need to exclude them

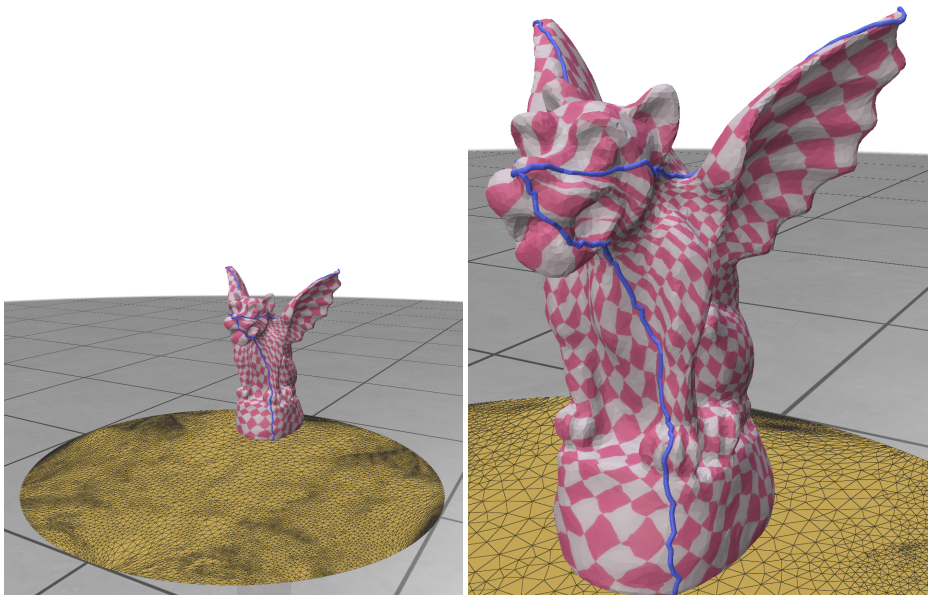

Figure 3: Tutte parameterization. The orange mesh shows the explicit UV mesh, the texture and the seam are shown on the object (with zoom-in on the right).

from the linear equation apriori). Again, if the mesh doesn't have any boundary vertices, scale it unifromly so that its new total surface area is like its original one (measured as `np.sum(vorAreas)`)

Discussion We are actually cheating here; this flow should be done when L and M *keep changing* along with the smoothed mesh, and we are always using the original L and M without changing in each iteration; however that is a reasonable approximation that is considerably more stable.

Grading The grading for this feature is worth 20%, where 15% are by a perfect result in the grading script `MCFlowGrader.py`, and 5% are for the analysis in the report—choose examples from the benchmark that clearly demonstrates difference between explicit and implicit, and discuss artifacts and the flow rate.

3 Tutte Parameterization

As a different application of the same operators, we do the convex Tutte parameterization (Lecture 12), to be able to do UV mapping of a 3D mesh of disc topology, and consequently texture it with a quad pattern, done automatically by PolyScope, that shows the quality of the mapping. This Section uses a different

benchmark, under the subfolder `param` within the `data` folder, and it visualized by the script `Parameterization.py`. Tutte parameterization involves two steps:

Setting the boundary The mesh is already cut so that it has a single boundary loop, and also you can assume that the loop is already sorted in `boundVertices` (this is already coded for you in `compute_edge_list()`). Tutte parameterization comprises then the two following steps

Setting the boundary You need to implement `compute_boundary_embedding()` to generate UV coordinates for the `boundVertices`. We will do so by setting them in a circle of radius r (as an input parameter), where the sector angle each boundary edge occupies is proportional to its relative length. Specifically, denote the sorted boundary loop B of size $|B|$ as $\left\{v_0, v_1, \cdots, v_{|B|}\right\}$. We then define the individual sector angles as:

$$\psi(v_i, v_{i+1}) = 2\pi \frac{|v_{i+1} - v_i|}{\sum_i |v_{i+1} - v_i|}$$

The sum is taken including the last edge $(v_{|B|}, v_0)$. Our UV embedding of each boundary vertex is then done by a cumulative sum of the individual angles:

$$UV_i = r \left(cos \left(\sum_{j=0}^{i} \psi(v_j, v_{j+1}) \right), sin \left(\sum_{j=0}^{i} \psi(v_j, v_{j+1}) \right) \right).$$

Intuitively, we put UV_0 at $r(cos(\psi(v_0, v_1)), sin(\psi(v_0, v_1)))$, and then keep adding the angles until $UV_{|B|} = (r, 0)$. Use `np.cumsum()` to do the cumulative sum of ψ.

Setting the inner vertices We will next solve for the harmonic $Lv = 0$ for the inner vertices, given the boundary UV. This is exactly Section 3 of Practical 0 again, which comprises a linear least-squares system with fixed variables, and should be implemented in function `compute_tutte_embedding()`. What you need to do is separate, by column slicing, d_0 into $d_{0,I}$ for the inner vertices v_I, and $d_{0,B}$ for the boundary vertices v_B, and solve for:

$$d_{0,I}^T W d_{0,I} UV_I = -d_{0,I}^T W d_{0,B} UV_B.$$

v_B are the UV values computed in the boundary-setting step. Note that you can solve directly for a right-hand side which is a matrix, where we have that UV_B is of size $|B| \times 2$; it will automatically solve individually for U and for V which is correct. Having gotten UV_I, reintegrate it with UV_B to the full UV for the output. The visualizer will show you:

- The UV mesh, which is your original mesh but with UV as its coordinates. It will be a flat orange disc on the plane.

- The original mesh with a seam, and with a textured quad mesh on it. You can play with the `period` parameter in PolyScope to get more or less resolution.

Note that you do not use the explicit L here, but rather d_0 and W, since you need to slice d_0; they should too be the output of `compute_laplacian()`.

Grading The grading for this feature is worth 20%, where 15% are by a perfect result in the grading script `ParamGrader.py`, and 15% are for the analysis in the report—show nice examples that clearly show distortion, and discuss the reasons for this distortion.

4 Advanced Analysis and Parameterization

This section is a more advanced extension of the practical. You should chose *only one* of each of these following extension options:

4.1 Computing principal curvatures and directions

We would like to compute the two vector fields of principal directions, and the consequent scalar values of principal curvatures. There are many ways to do so, but a simple, yet computationally expensive, way is that of *quadratic fitting*. Consider a vertex v and its normal n. You can rotate the local environment so that $n = \hat{z}$ and the vertex is at the origin. Then, treat the 1-ring as a height function, and fit, by least squares, the entire 1-ring with a single best-fit quadratic function

$$z_v(x, y) = ax^2 + by^2 + cxy + dx + ey + f.$$

Given this continuous function at v, find out the principal directions and curvatures, and assign them to v. This will require a bit of pen-and-paper work to figure out exactly how, but not much implementation. Note that you might not have enough vertices in the 1-ring for the fitting, which means you should employ the minimum 2-norm solution. Write a script that shows these quantities clearly on the benchmark, and document this in your report. You should use PolyScope vector field visualization to show fields, as you did for the normals.

4.2 LCSM Parameterization

To counter the rather large distortion artefacts of Tutte parameterization, you should instead compute the Least-Squares Conformal-Mapping parameterization method learned in class, which only pegs down two vertices and allows the boundary to naturally emerge, where the objective is that triangles are mapped as conformally as possible from the mesh to the plane. Write a variation of the parameterization script to compare between the two results. The challenge in

this one is properly constructing the left-hand side matrix, but it's again not much of an implementation challenge, but more of writing it down properly.

This section will not be automatically graded since the results may vary on the design decisions here. The worth of this section is 25%, and it will be manually graded upon the convincing demonstration in the code and in the report of the results.

5 Submission

The report must be at most 3-pages long including all figures. All functionality should be done by changing the function bodies in `DAFunctions.py`. The submission will be in the official place on Learn, where you should submit a ZIP file of only the affected code scripts, any auxiliary data you would like to share, and a PDF of your report.