HOMEWORK 3

Surface Fairing

Earlier we mentioned that the Laplace-Beltrami operator (commonly abbreviated as just the *Laplacian*) plays a fundamental role in a variety of geometric and physical equations. In this chapter we'll put the Laplacian to work by coming up with a discrete version of the *Poisson equation* for triangulated surfaces. As in the chapter on vertex normals, we'll see that the same discrete expression for the Laplacian (via the *cotan formula*) arises from two very different ways of looking at the problem: using *test functions* (often known as *Galerkin projection*), or by integrating differential forms (often called *discrete exterior calculus*).

6.1. Basic Properties

Before we start talking about discretization, let's establish a few basic facts about the Laplace operator Δ and the standard *Poisson problem*

$$\Delta\phi = \rho.$$

Poisson equations show up all over the place—for instance, in physics ρ might represent a mass density in which case the solution ϕ would (up to suitable constants) give the corresponding gravitational potential. Similarly, if ρ describes an charge density then ϕ gives the corresponding electric potential (you'll get to play around with these equations in the code portion of this assignment). In geometry processing a surprising number of things can be done by solving a Poisson equation (e.g., smoothing a surface, computing a vector field with prescribed singularities, or even computing the geodesic distance on a surface).

Often we'll be interested in solving Poisson equations on a compact surface M without boundary.

EXERCISE 18. A twice-differentiable function $\phi : M \to \mathbb{R}$ is called *harmonic* if it sits in the kernel of the Laplacian, i.e., $\Delta\phi = 0$. Argue that the *only* harmonic functions on a compact domain without boundary are the *constant* functions.

Your argument does not have to be incredibly formal—there are just a couple simple observations that capture the main idea. This fact is quite important because it implies that we can add a constant to any solution to a Poisson equation. In other words, if ϕ satisfies $\Delta\phi = \rho$, then so does $\phi + c$ since $\Delta(\phi + c) = \Delta\phi + \Delta c = \Delta\phi + 0 = \rho$.

EXERCISE 19. A separate fact is that on a compact domain without boundary, constant functions are not in the image of Δ. In other words, there is no function ϕ such that $\Delta\phi = c$. Why?

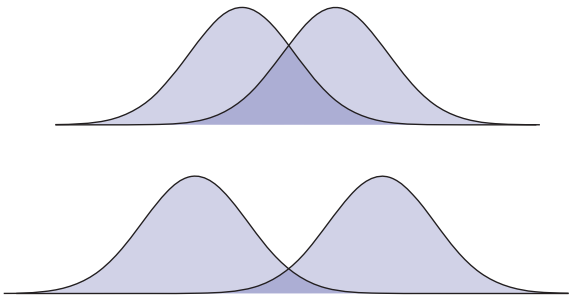
This fact is also important because it tells us when a given Poisson equation admits a solution. In particular, if ρ has a constant component then the problem is not well-posed. In some situations,

however, it may make sense to simply remove the constant component. I.e., instead of trying to solve $\Delta\phi = \rho$ one can solve $\Delta\phi = \rho - \bar{\rho}$, where $\bar{\rho} := \int_M \rho \, dV / |M|$ and $|M|$ is the total volume of M. However, you *must* be certain that this trick makes sense in the context of your particular problem!

When working with PDEs like the Poisson equation, it's often useful to have an *inner product* between functions. An extremely common inner product is the L^2 *inner product* $\langle \cdot, \cdot \rangle$, which takes the integral of the pointwise product of two functions over the entire domain Ω:

$$\langle f, g \rangle := \int_\Omega f(x) g(x) dx.$$

In spirit, this operation is similar to the usual *dot product* on $\mathbb{R}^n$: it measures the degree to which two functions "line up." For instance, the top two functions have a large inner product; the bottom two have a smaller inner product (as indicated by the dark blue regions):



Similarly, for two vector fields X and Y we can define an L^2 inner product

$$\langle X, Y \rangle := \int_\Omega X(x) \cdot Y(x) dx$$

which measures how much the two fields "line up" at each point.

Using the L^2 inner product we can express an important relationship known as *Green's first identity*. Green's identity says that for any sufficiently differentiable functions f and g

$$\langle \Delta f, g \rangle = -\langle \nabla f, \nabla g \rangle + \langle N \cdot \nabla f, g \rangle_\partial,$$

where $\langle \cdot, \cdot \rangle_\partial$ denotes the inner product on the boundary and N is the outward unit normal.

EXERCISE 20. Using exterior calculus, show that Green's identity holds. Hint: apply Stokes' theorem to the 1-form $g \star df$.

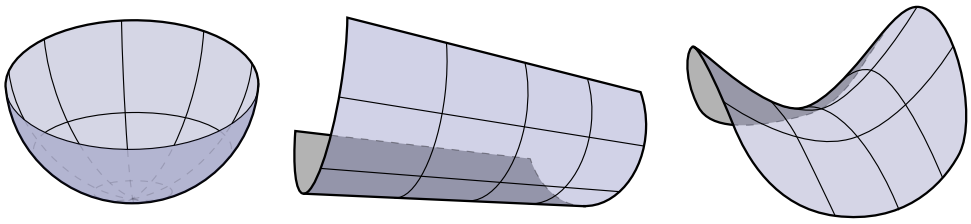
One last key fact about the Laplacian is that it is *positive-semidefinite*, i.e., Δ satisfies

$$\langle \Delta\phi, \phi \rangle \geq 0$$

for all functions ϕ. (By the way, why isn't this quantity *strictly* greater than zero?) Words cannot express the importance of (semi)definiteness. Let's think about a very simple example: functions of the form $\phi(x, y) = ax^2 + bxy + cy^2$ in the plane. Any such function can also be expressed in matrix form:

$$\phi(x, y) = \underbrace{\begin{bmatrix} x & y \end{bmatrix}}_{\mathbf{x}^T} \underbrace{\begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x \\ y \end{bmatrix}}_{\mathbf{x}} = ax^2 + bxy + cy^2,$$

and we can likewise define positive-semidefiniteness for A. But what does it actually look like? As depicted below, positive-definite matrices ($\mathbf{x}^T A\mathbf{x} > 0$) look like a bowl, positive-semidefinite matrices ($\mathbf{x}^T A\mathbf{x} \geq 0$) look like a half-cylinder, and indefinite matrices ($\mathbf{x}^T A\mathbf{x}$ might be positive or negative depending on $\mathbf{x}$) look like a saddle:



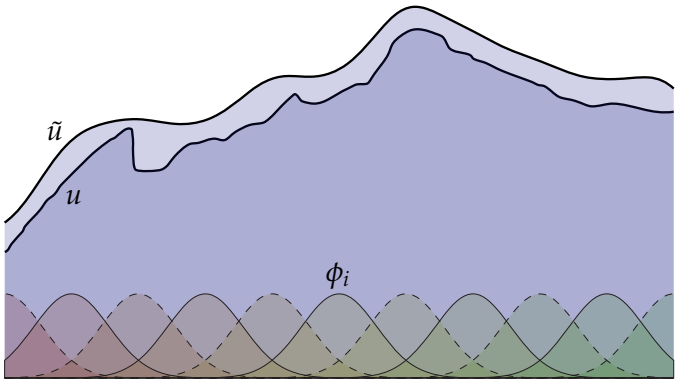
$x^2 + y^2$ *definite* | x^2 *semidefinite* | $x^2 - y^2$ *indefinite*

Now suppose you're a back country skiier riding down this kind of terrain in the middle of a howling blizzard. You're cold and exhausted, and you know you parked your truck in a place where the ground levels out, but where exactly is it? The snow is blowing hard and visibility is low—all you can do is keep your fingers crossed and follow the slope of the mountain as you make your descent. (Trust me: this is really how one feels when doing numerical optimization!) If you were smart and went skiing in Pos Def Valley then you can just keep heading down and will soon arrive safely back at the truck. But maybe you were feeling a bit more adventurous that day and took a trip to Semi Def Valley. In that case you'll still get to the bottom, but may have to hike back and forth along the length of the valley before you find your car. Finally, if your motto is "safety second" then you threw caution to the wind and took a wild ride in Indef Valley. In this case you may never make it home!

In short: positive-semidefinite matrices are nice because it's easy to find the minimum of the quadratic functions they describe—many tools in numerical linear algebra are based on this idea. Same goes for positive-semidefinite *linear operators* like the Laplacian Δ, which can often be thought of as sort of infinite-dimensional matrices (if you take some time to read about the spectral theorem, you'll find that this analogy runs even deeper). Given the ubiquity of Poisson equations in geometry and physics, it's a damn good thing Δ is positive-semidefinite!

EXERCISE 21. Using Green's first identity, show that Δ is negative-semidefinite on any compact surface M without boundary. From a practical perspective, why are negative semi-definite operators just as good as positive semi-definite ones?

6.2. Discretization via FEM

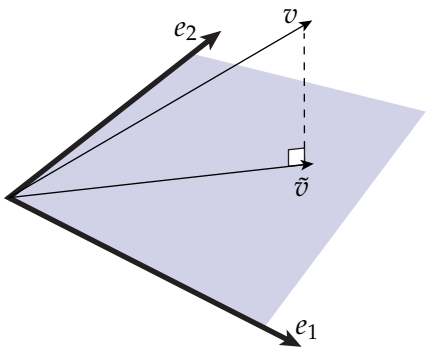


The solution to a geometric or physical problem is often described by a *function*: the temperature at each point on the Earth, the curvature at each point on a surface, the amount of light hitting each point of your retina, etc. Yet the space of *all possible* functions is mind-bogglingly large—too large to be represented on a computer. The basic idea behind the *finite element method* (FEM) is to pick a smaller space of functions and try to find the best possible solution from within this space. More specifically, if u is the true solution to a problem and $\{\phi_i\}$ is a collection of *basis functions*, then we seek the linear combination of these functions

$$\tilde{u} = \sum_i x_i \phi_i, \; x_i \in \mathbb{R}$$

such that the difference $||\tilde{u} - u||$ is as small as possible with respect to some norm. (Above we see a detailed curve u and its best approximation $\tilde{u}$ by a collection of bump-like basis functions ϕ_i.)

Let's start out with a very simple question: suppose we have a vector $v \in \mathbb{R}^3$, and want to find the best approximation $\tilde{v}$ within a plane spanned by two basis vectors $e_1, e_2 \in \mathbb{R}^3$:



Since $\tilde{v}$ is forced to stay in the plane, the best we can do is make sure there's error *only* in the normal direction. In other words, we want the error $\tilde{v} - v$ to be orthogonal to both basis vectors e_1 and e_2:

$$\begin{aligned} (\tilde{v} - v) \cdot e_1 &= 0, \\ (\tilde{v} - v) \cdot e_2 &= 0. \end{aligned}$$

In this case we get a system of two linear equations for two unknowns, and can easily compute the optimal vector $\tilde{v}$.

Now a harder question: suppose we want to solve a standard Poisson problem

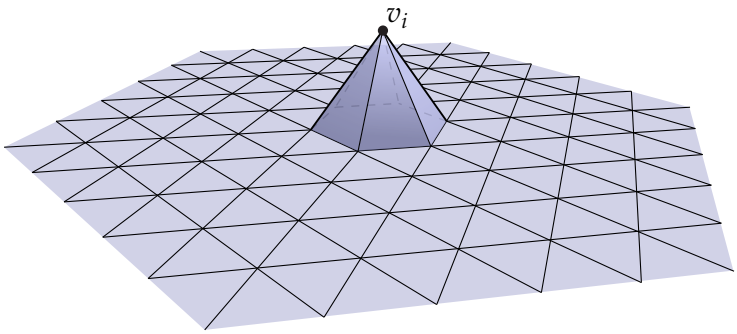
$$\Delta u = f.$$

How do we check whether a given function $\tilde{u}$ is the best possible solution? The basic picture still applies, except that our bases are now *functions* ϕ instead of finite-dimensional vectors e_i, and the simple vector dot product $\cdot$ gets replaced by the L^2 *inner product*. Unfortunately, when trying to solve a Poisson equation we don't know what the correct solution u looks like (otherwise we'd be done already!). So instead of the error $\tilde{u} - u$, we'll have to look at the *residual* $\Delta\tilde{u} - f$, which measures how closely $\tilde{u}$ satisfies our original equation. In particular, we want to "test" that the residual vanishes along each basis direction ϕ_j:

$$\langle \Delta\tilde{u} - f, \phi_j \rangle = 0,$$

again resulting in a system of linear equations. This condition ensures that the solution behaves just as the true solution would over a large collection of possible "measurements."

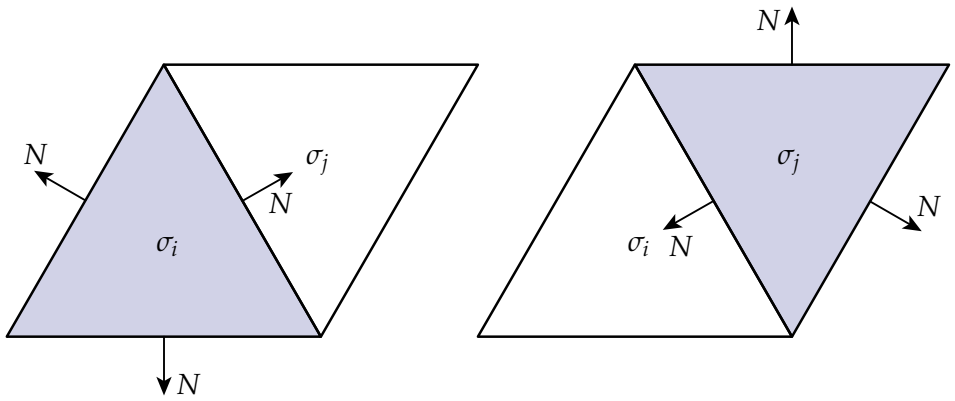
Next, let's work out the details of this system for a triangulated surface. The most natural choice of basis functions are the piecewise linear *hat functions* ϕ_i, which equal one at their associated vertex and zero at all other vertices:



At this point you might object: if all our functions are linear, and Δ is a *second* derivative, aren't we just going to get zero every time we evaluate Δu? Fortunately we're saved by Green's identity—let's see what happens if we apply this identity to our triangle mesh, by breaking up the integral into a sum over individual triangles σ:

$$\begin{aligned} \langle \Delta u, \phi_j \rangle &= \textstyle\sum_k \langle \Delta u, \phi_j \rangle_{\sigma_k} \\ &= \textstyle\sum_k \langle \nabla u, \nabla \phi_j \rangle_{\sigma_k} + \sum_k \langle N \cdot \nabla u, \phi_j \rangle_{\partial\sigma_k}. \end{aligned}$$

If the mesh has no boundary then this final sum will disappear since the normals of adjacent triangles are oppositely oriented, hence the boundary integrals along shared edges cancel each-other out:



In this case, we're left with simply

$$\langle \nabla u, \nabla \phi_j \rangle$$

in each triangle σ_k. In other words, we can "test" Δu as long as we can compute the gradients of both the candidate solution u and each basis function ϕ_j. But remember that u is itself a linear combination of the bases ϕ_i, so we have

$$\langle \nabla u, \nabla \phi_j \rangle = \left\langle \nabla \sum_i x_i \phi_i, \nabla \phi_j \right\rangle = \sum_i x_i \langle \nabla \phi_i, \nabla \phi_j \rangle.$$

The critical thing now becomes the inner product between the gradients of the basis functions in each triangle. If we can compute these, then we can simply build the matrix

$$A_{ij} := \langle \nabla \phi_i, \nabla \phi_j \rangle$$

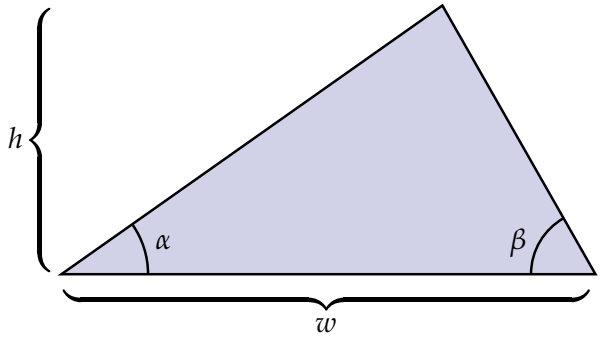
and solve the problem

$$Ax = b$$

for the coefficients x, where the entries on the right-hand side are given by $b_i = \langle f, \phi_i \rangle$ (i.e., we just take the same "measurements" on the right-hand side).

EXERCISE 22. Show that the aspect ratio of a triangle can be expressed as the sum of the cotangents of the interior angles at its base, i.e.,

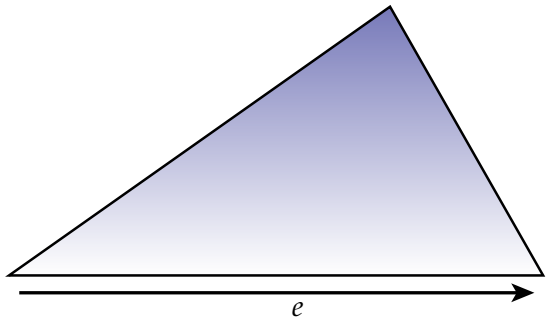
$$\frac{w}{h} = \cot \alpha + \cot \beta.$$



EXERCISE 23. Let e be the edge vector along the base of a triangle. Show that the gradient of the hat function ϕ associated with the opposite vertex is given by

$$\nabla\phi = \frac{e^\perp}{2A},$$

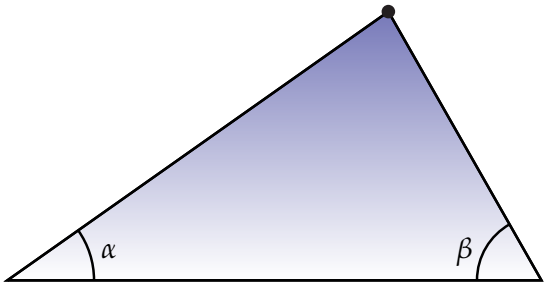
where $e^\perp$ is the vector e rotated by a quarter turn in the counter-clockwise direction and A is the area of the triangle.



EXERCISE 24. Show that for any hat function ϕ associated with a given vertex

$$\langle \nabla\phi, \nabla\phi \rangle = \frac{1}{2}(\cot\alpha + \cot\beta)$$

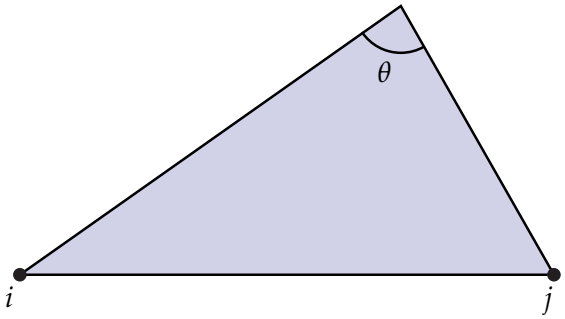
within a given triangle, where α and β are the interior angles at the remaining two vertices.



EXERCISE 25. Show that for the hat functions ϕ_i and ϕ_j associated with vertices i and j (respectively) of the same triangle, we have

$$\langle \nabla\phi_i, \nabla\phi_j \rangle = -\frac{1}{2}\cot\theta,$$

where θ is the angle between the opposing edge vectors.



Putting all these facts together, we find that we can express the Laplacian of u at vertex i via the infamous *cotan formula*

$$(\Delta u)_i = \frac{1}{2}\sum_j (\cot \alpha_j + \cot \beta_j)(u_j - u_i),$$

where we sum over the immediate neighbors of vertex i.

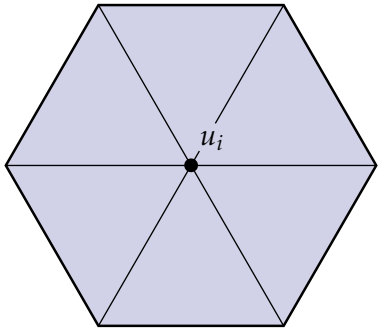
6.3. Discretization via DEC

The FEM approach reflects a fairly standard way to discretize partial differential equations. But let's try a different approach, based on discrete exterior calculus (DEC). Interestingly enough, although these two approaches are quite different, we end up with exactly the same result!

Again we want to solve the Poisson equation $\Delta u = f$, which (if you remember our discussion of differential operators) can also be expressed as

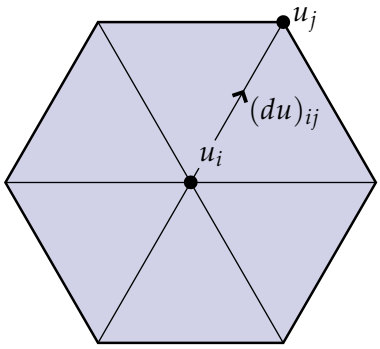
$$\star d \star du = f.$$

We already outlined how to discretize this kind of expression in the notes on discrete exterior calculus, but let's walk through it step by step. We start out with a 0-form u, which is specified as a number u_i at each vertex:



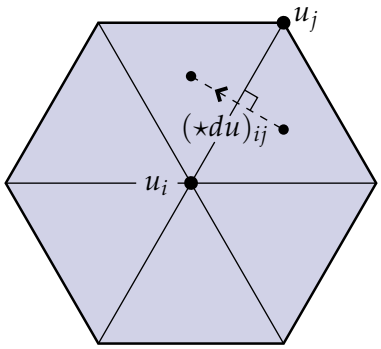
Next, we compute the discrete exterior derivative du, which just means that we want to *integrate* the derivative along each edge:

$$(du)_{ij} = \int_{e_{ij}} du = \int_{\partial e_{ij}} u = u_j - u_i.$$



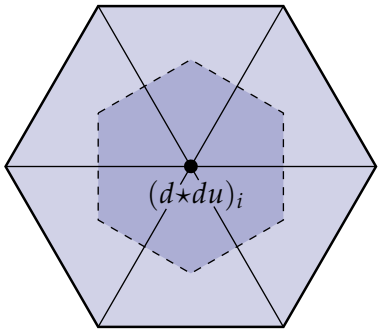
(Note that the boundary ∂e_{ij} of the edge is simply its two endpoints v_i and v_j.) The Hodge star converts a circulation along the edge e_{ij} into the flux through the corresponding dual edge $e^\star_{ij}$. In particular, we take the *total circulation* along the primal edge, divide it by the edge length to get the *average pointwise circulation*, then multiply by the dual edge length to get the *total flux* through the dual edge:

$$(\star du)_{ij} = \frac{|e^\star_{ij}|}{|e_{ij}|}(u_j - u_i).$$



Here $|e_{ij}|$ and $|e^\star_{ij}|$ denote the length of the primal and dual edges, respectively. Next, we take the derivative of $\star du$ and integrate over the whole dual cell:

$$(d \star du)_i = \int_{C_i} d \star du = \int_{\partial C_i} \star du = \sum_j \frac{|e^\star_{ij}|}{|e_{ij}|}(u_j - u_i).$$



The final Hodge star simply divides this quantity by the area of C_i to get the average value over the cell, and we end up with a system of linear equations

$$(\star d \star du)_i = \frac{1}{|C_i|} \sum_j \frac{|e^\star_{ij}|}{|e_{ij}|}(u_j - u_i) = f_i$$

where f_i is the value of the right-hand side at vertex i. In practice, however, it's often preferable to move the area factor $|C_i|$ to the right hand side, since the resulting system

$$(d \star du)_i = \sum_j \frac{|e^\star_{ij}|}{|e_{ij}|}(u_j - u_i) = |C_i| f_i$$

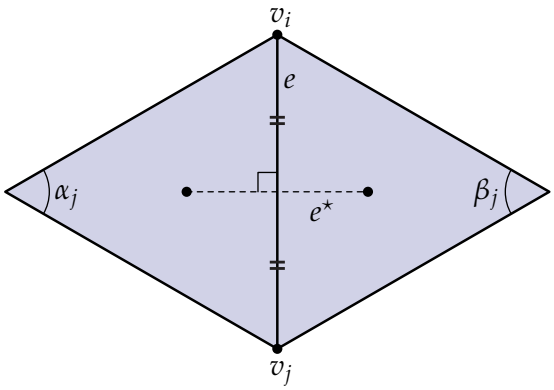
can be represented by a *symmetric* matrix. (Symmetric matrices are often easier to deal with numerically and lead to more efficient algorithms.) Another way of looking at this transformation

is to imagine that we discretized the system

$$d \star du = \star f.$$

In other words, we converted an equation in terms of 0-forms into an equation in terms of n-forms. When working with surfaces, the operator $d \star d$ is sometimes referred to as the *conformal Laplacian*, because it does not change when we subject our surface to a conformal transformation. Alternatively, we can think of $d \star d$ as simply an operator that gives us the value of the Laplacian integrated over each dual cell of the mesh (instead of the pointwise value).

EXERCISE 26. Consider a simplicial surface and suppose we place the vertices of the dual mesh at the circumcenters of the triangles (i.e., the center of the unique circle containing all three vertices):



Demonstrate that the dual edge $e^\star$ (i.e., the line between the two circumcenters) bisects the primal edge orthogonally, and use this fact to show that

$$\frac{|e^\star_{ij}|}{|e_{ij}|} = \frac{1}{2}(\cot \alpha_j + \cot \beta_j).$$

Hence the DEC discretization yields precisely the same "cotan-Laplace" operator as the Galerkin discretization.

6.4. Meshes and Matrices

So far we've been giving a sort of "algorithmic" description of operators like Laplace. For instance, we determined that the Laplacian of a scalar function u at a vertex i can be approximated as

$$(\Delta u)_i = \frac{1}{2}\sum_j (\cot \alpha_j + \cot \beta_j)(u_j - u_i),$$

where the sum is taken over the immediate neighbors j of i. In code, this sum could easily be expressed as a *loop* and evaluated at any vertex. However, a key aspect of working with discrete differential operators is building their *matrix representation*. The motivation for encoding an operator as a matrix is so that we can solve systems like

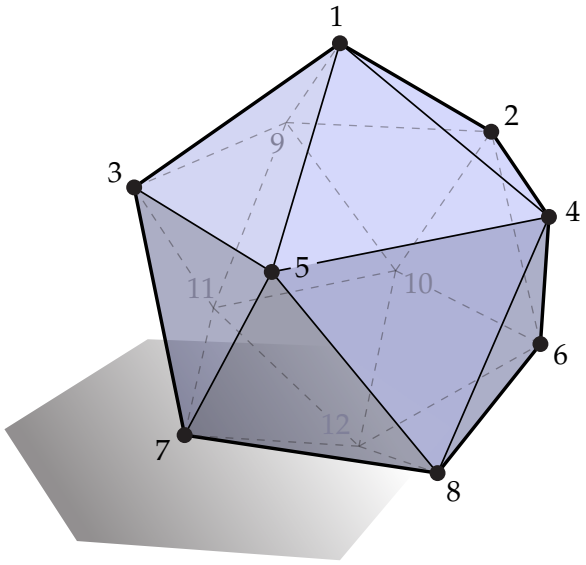
$$\Delta u = f$$

using a standard numerical linear algebra package. (To make matters even more complicated, some linear solvers are perfectly happy to work with algorithmic representations of operators called *callback functions*—in general, however, we'll need a matrix.)

In the case of the Poisson equation, we want to construct a matrix $L \in \mathbb{R}^{|V| \times |V|}$ (where $|V|$ is the number of mesh vertices) such that for any vector $u \in \mathbb{R}^{|V|}$ of values at vertices, the expression Lu effectively evaluates the formula above. But let's start with something simpler—consider an operator B that computes the sum of all neighboring values:

$$(Bu)_i = \sum_j u_j$$

How do we build the matrix representation of this operator? Think of B a machine that takes a vector u of input values at each vertex and spits out another vector Bu of output values. In order for this story to make sense, we need to know which values correspond to which vertices. In other words, we need to assign a unique *index* to each vertex of our mesh, in the range $1, \ldots, |V|$:



It doesn't matter which numbers we assign to which vertices, so long as there's one number for every vertex and one vertex for every number. This mesh has twelve vertices and vertex 1 is next to vertices 2, 3, 4, 5, and 9. So we could compute the sum of the neighboring values as

$$(Bu)_1 = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \\ u_{12} \end{bmatrix}.$$

Here we've put a "1" in the jth place whenever vertex j is a neighbor of vertex 1 and a "0" otherwise. Since this row gives the "output" value at the first vertex, we'll make it the first row of the matrix B. The entire matrix looks like this:

$$B = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(You could verify that this matrix is correct, or you could go outside and play in the sunshine. Your choice.) In practice, fortunately, we don't have to build matrices "by hand"—we can simply start with a matrix of zeros and fill in the nonzero entries by looping over local neighborhoods of our mesh.

Finally, one important thing to notice here is that many of the entries of B are *zero*. In fact, for most discrete operators the number of zeros far outnumbers the number of nonzeros. For this reason, it's usually a good idea to use a *sparse matrix*, i.e., a data structure that stores only the location and value of nonzero entries (rather than explicitly storing every single entry of the matrix). The design of sparse matrix data structures is an interesting question all on its own, but conceptually you can imagine that a sparse matrix is simply a list of triples (i, j, x) where $i, j \in \mathbb{N}$ specify the row and column index of a nonzero entry and $x \in \mathbb{R}$ gives its value.

6.5. The Poisson Equation

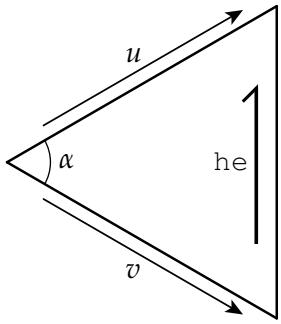
In the first part of the coding assignment you'll build the cotan-Laplace operator and use it to solve the scalar Poisson equation

$$\Delta\phi = \rho$$

on a triangle mesh, where ρ can be thought of as a (mass or charge) density and ϕ can be thought of as a (gravitational or electric) potential. Once you've implemented the methods below, you can visualize the results via the Viewer. (If you want to play with the density function ρ, take a look at the method `Viewer::updatePotential`.)

CODING 7. Assigns a unique ID to each vertex of the mesh, in the range $0, \ldots, |V| - 1$. (Details on exactly how and where to specify these indices can be found in the online writeup.)

CODING 8. Derive an expression for the cotangent of a given angle purely in terms of the two incident edge vectors and the standard Euclidean dot product $(\cdot)$ and cross product $(\times)$. Implement a method that computes the cotangent weight for a given edge, using this formula.

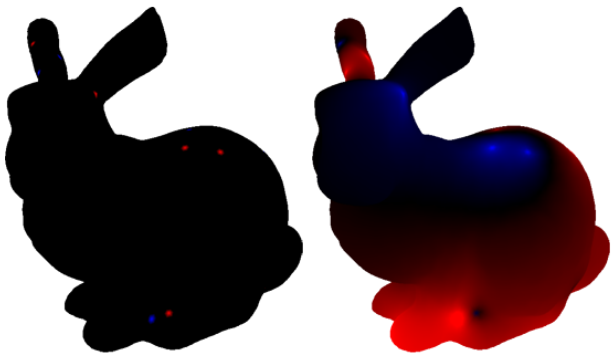


CODING 9. Build a diagonal mass matrix, where the diagonal entries are given by the dual vertex areas. For the dual area of a vertex you can simply use one-third the area of the incident faces—you do not need to compute the area of the circumcentric dual cell. (This choice of area will not affect the order of the rate of convergence.)

CODING 10. Using the methods you've written so far, build a matrix representing the cotan-Laplace operator.

CODING 11. Implement a method that solves the problem $\Delta\phi = \rho$ where ρ is a scalar density on vertices. Be careful about appropriately incorporating the mass matrix into your computations; also remember that the right-hand side cannot have a constant component (else a solution does not exist!).

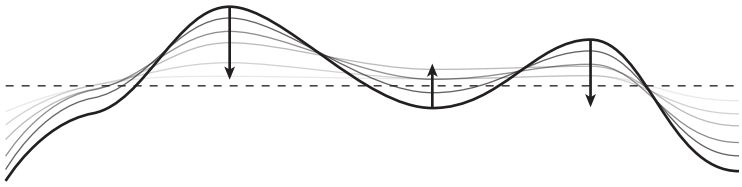
You should verify that your code produces results that look something like these two images (density on the left; corresponding potential on the right), though the color scheme may be a bit different:



6.6. Implicit Mean Curvature Flow

Next, you'll use nearly identical code to smooth out geometric detail on a surface mesh (also known as *fairing* or *curvature flow*). The basic idea is captured by the *heat equation*, which describes the way heat diffuses over a domain. For instance, if u is a scalar function describing the temperature at every point on the real line, then the heat equation is given by

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}.$$



Geometrically this equation simply says that concave bumps get pushed down and convex bumps get pushed up—after a long time the heat distribution becomes completely flat. We also could have written this equation using the Laplacian: $\frac{\partial u}{\partial t} = \Delta u$. In fact, this equation is exactly the one we'll use to smooth out a surface, except that instead of considering the evolution of temperature, we consider the flow of the surface $f : M \to \mathbb{R}^3$ itself:

$$\frac{\partial f}{\partial t} = \Delta f.$$

Remember from our discussion of vertex normals that $\Delta f = 2HN$, i.e., the Laplacian of position yields (twice) the mean curvature times the unit normal. Therefore, the equation above reads, "move the surface in the direction of the normal, with strength proportional to mean curvature." In other words, it describes a *mean curvature flow*.

So how do we compute this flow? We already know how to discretize the term Δf—just use the cotangent discretization of Laplace. But what about the time derivative $\frac{\partial f}{\partial t}$? There are all sorts of interesting things to say about discretizing time, but for now let's use a very simple idea: the change over time can be approximated by the *difference* of two consecutive states:

$$\frac{\partial f}{\partial t} \approx \frac{f_h - f_0}{h},$$

where f_0 is the initial state of our system (here the initial configuration of our mesh) and f_h is the configuration after a mean curvature flow of some duration $h > 0$. Our discrete mean curvature flow then becomes

$$\frac{f_h - f_0}{h} = \Delta f.$$

The only remaining question is: which values of f do we use on the right-hand side? One idea is to use f_0, which results in the system

$$f_h = f_0 + h\Delta f_0.$$

This scheme, called *forward Euler*, is attractive because it can be evaluated directly using the known data f_0—we don't have to solve a linear system. Unfortunately, forward Euler is not numerically stable, which means we can take only very small time steps h. One attractive alternative is to use f_h as the argument to Δ, leading to the system

$$\underbrace{(I - h\Delta)}_{A} f_h = f_0,$$

where I is the identity matrix (try the derivation yourself!) This scheme, called *backward Euler*, is far more stable, though we now have to solve a linear system $Af_h = f_0$. Fortunately this system is highly *sparse*, which means it's not too expensive to solve in practice. (Note that this system is not much different from the Poisson system.)

CODING 12. Build a matrix representing the flow operator for a single time step; this matrix should not be much different from the one you used for the Poisson problem.

CODING 13. Implement a method which takes a single step of implicit mean curvature flow. Note that you can treat each of the components of f (x, y, and z) as separate scalar quantities, i.e., you can solve for the components one at a time.

You should verify that your code produces results that look something like these two images (original mesh on the left; smoothed mesh on the right):