Assignment 4: Mesh Parametrization

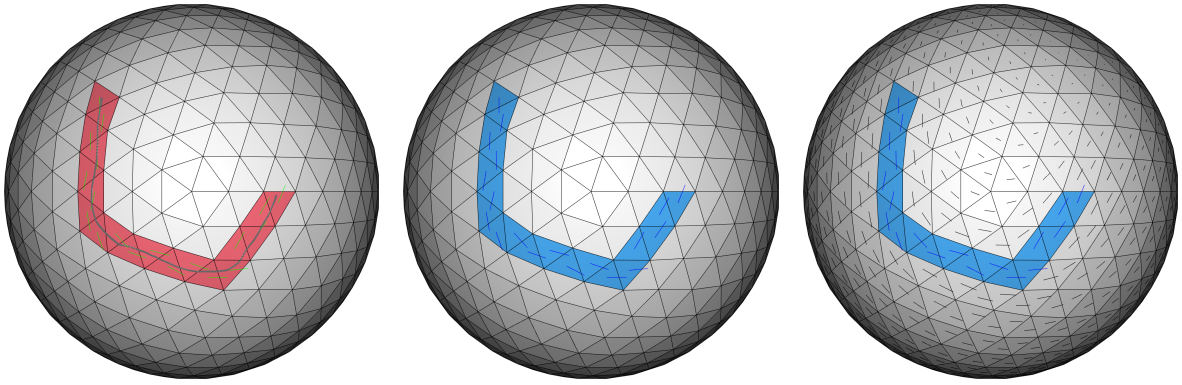
Handout date: 03/27/2017
Submission deadline: 4/11/2017, 11:59PM
Demo session: 4/26/2017, 2-3PM

In this exercise you will

- Familiarize yourself with vector field design on surfaces.
- Create scalar fields whose gradients align with given vector fields as closely as possible.
- Experiment with the `libigl` implementation of harmonic and least-squares conformal parameterizations.

1. Tangent vector fields for scalar field design

Our first task is to design smooth tangent vector fields on a surface; these will be "integrated" later to define a scalar field. A (piecewise constant) vector field on a triangle mesh is defined as an assignment of a single vector to each triangle such that each vector lies in the tangent plane containing the triangle. We will design fields to follow a set of alignment constraints provided by the user: the user specifies the field vectors at a subset of the mesh triangles (the constraints) and those constraints are interpolated smoothly throughout the surface to define a field.



(A) Selection (B) Constraints (C) Vector Field

Figure 1. Selecting faces and assigning vector constraints via the UI; interpolated vector field.

1.1. **Creating vector constraints.** The provided assignment code already implements a minimal, stroke-based interface for assigning vector constraints at faces. To use it, enable `selection_mode` from the Nanogui and drag with your mouse to draw a stroke over the surface. The faces below the

stroke are selected, and the path of the stroke is used to define a tangent vector on each selected face (tangent to the stroke along the direction of mouse motion); see Fig. 1a.

The selected faces, the vectors, and the stroke are saved into the `selected_faces`, `selected_vec3`, `selection_stroke_points` variables. You have the option to: (a) accept the selection and the assigned vectors as constraints by hitting the relevant Nanogui button, (b) replace the selection with a new one by drawing another stroke on the mesh, or (c) discard the selection with the `Clear selection` button. Once a selection is accepted, you can add more constraints by drawing more strokes. You can also clear all constraints and read/write them from a file. The constrained faces' indices and their painted values (vectors) are saved into the `constraint_fi` and `constraint_vec3` variables.

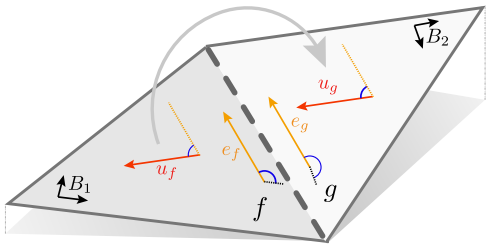
Pressing '1' displays the constraints and the current selection and stroke. Feel free to add more functionality for designing constraints as needed.

Relevant `libigl` *functions:* None.

1.2. **Interpolating the constraints.** Since each field vector $\mathbf{u}_f$ lies in the plane of its respective triangle f, it can be decomposed into the triangle's local basis and represented with two real coefficients: $\mathbf{u}_f = (x_f, y_f) \in \mathbb{R}^2$. Alternatively, we can identify the triangle plane with the complex plane, expressing the vector as a single complex number $u_f = x_f + iy_f \in \mathbb{C}$.

The interpolation produces a smooth vector field from the constraints by trying to make each vector as similar as possible to the adjacent triangles' vectors. However, since vectors u_f, u_g at adjacent triangles f, g are expressed with respect to the triangles' local bases, and the triangles generally have different bases, their complex expressions cannot be compared directly (see inset figure). So we must first express the vectors with respect to a common basis.

A simple way to choose a common basis is to use the shared (directed) edge vector as the new real (x) axis for both triangles. This implicitly also chooses the $90°$-counterclockwise-rotated edge vector as the imaginary axis, forming a complete basis. In this new common basis, the two triangles' vectors can be written as $\tilde{u}_f = u_f e_f^{-1} = u_f \overline{e_f}$, $\tilde{u}_g = u_g \overline{e_g}$. Where e_f, e_g are the shared edge vector expressed in each local basis. Here $\overline{e_f}$ denotes the complex conjugate of e_f, *and we assume* e_f, e_g *are normalized so that the conjugate is actually the inverse*—normalize them!



The difference ("non-smoothness") of the two vectors across the edge (f, g) can now be computed as $E_{fg}(u_f, u_g) = \|u_f \overline{e_f} - u_g \overline{e_g}\|^2$ (recall that $\|z\|^2 = z\overline{z}$ is a complex number z's squared magnitude). This is a quadratic form on the complex variables u_f, u_g and can be manipulated into the form $E_{fg}(u_f, u_g) = \begin{bmatrix}\overline{u_f} & \overline{u_g}\end{bmatrix} Q_{fg} \begin{bmatrix}u_f \\ u_g\end{bmatrix}$ for a particular complex matrix Q_{fg}. The full field's smoothness can then be written as the sum of all these per-edge energies: $\sum_{\text{edge}(f,g)} E_{fg}(u_f, u_g)$. This yields a sparse quadratic form $u^* Q u$, where the complex column vector u encodes each each per-face vector. The row vector u^* is u's adjoint (conjugate transpose), and Q is the appropriate combination of the matrices Q_{fg}. Thus, finding the smoothest field under the

prescribed constraints is equivalent solving

$$\begin{aligned} &\min\ u^* Q u \\ &\text{such that} \quad u|_{cf} = c, \end{aligned}$$

where cf are the constrained face indices and c are the prescribed vectors at those faces. We can differentiate the smoothness energy to find its minimum, thus obtaining a (complex) linear system

$$\begin{aligned} &Qu = 0 \\ &\text{such that } u|_{cf} = c. \end{aligned}$$

This system's solution describes the smoothly interpolated vector field (Fig. 1c). Your task is

- Determine how to construct the complex matrix Q,
- Solve the system under the prescribed constraints (which are specified via the UI as described above),
- Display the constraints and the interpolated field when '2' is pressed.

Note: Eigen supports sparse complex matrices and can factorize/solve linear systems with them (e.g. `SparseLU`), but feel free to convert this problem into real variables if you find that easier.

Relevant `libigl` *functions:* `igl::sparse`,`igl::colon`, `igl::slice`, `igl::slice_into`, `igl::cat`, `igl::kronecker_product`, `igl::speye`, `igl::repdiag` can be used for various sparse matrix construction and manipulation operations. `igl::local_basis` computes a basis for each triangle plane.

Required output for this section:

- Visualization of the constraints and interpolated field.
- An ASCII dump of the interpolated field ($\#F \times 3$ matrix, one vector per row) for the mesh `irr4-cyl2.off` and the input constraints in the provided file `irr4-cyl2.constraints`.

2. Reconstructing a scalar field from a vector field

Your task is now to find a scalar function $S(\mathbf{x})$ defined over the surface whose gradient fits a given vector field as closely as possible. The scalar field is defined by values on the mesh vertices that are linearly interpolated over each triangle's interior: for given vertex values s_i, the function $S(\mathbf{x})$ inside a triangle t is computed as $S_t(\mathbf{x}) = \sum_{\text{vertex } i \in t}^{3} s_i \phi_i^t(\mathbf{x})$, where $\phi_i^t(\mathbf{x})$ are the linear "hat" functions associated with each triangle vertex (i.e. $\phi_i^t(\mathbf{x})$ is linear and takes the value 1 at vertex i and 0 at all other vertices). Then the scalar function's (vector-valued) gradient is $\mathbf{g}_t = \nabla S_t = \sum_{\text{vertex } i \in t}^{3} s_i \nabla \phi_i^t$.

Since the "hat" functions are piecewise linear, their gradients $\nabla \phi_i^t$ are constant within each triangle, and so is $\mathbf{g}_t$ (the full scalar function's gradient). Specifically, $\mathbf{g}_t$ is a linear combination of the constant hat function gradients with the (unknown) values s_i as coefficients, meaning that we can write an expression of the form $g = Gs$, where s is a $\#V \times 1$ column vector holding each s_i, g is a column vector of size $3\#F \times 1$ consisting of the vectors $\mathbf{g}_t$ "flattened" and vertically stacked, and G is the so-called "gradient matrix" of size $3\#F \times \#V$.

Since there is no guarantee that our interpolated face-based field is actually the gradient of some function, we cannot attempt to integrate it directly. Instead, we will try to find $S(\mathbf{x})$ by asking its gradient to approximate the vector field u in the least-squares sense:

$$\min \sum_{\text{face } t} A_t \|\mathbf{g}_t - \mathbf{u}_t\|^2,$$

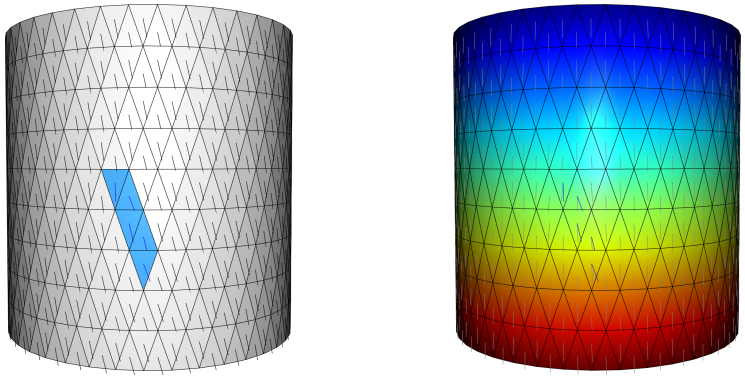
where A_t is triangle t's area, $\mathbf{g}_t$ is the (unknown) function gradient on the triangle, and $\mathbf{u}_t$ is the triangle's vector assigned by the guiding vector field. Using the linear relationship $g = Gs$, we can write this least-squares error as a (real) quadratic form:

$$s^T K s + s^T b + c$$

and minimize it by solving a linear system for the unknown s.

- Determine the matrix K and vector b in the above minimization (by expanding the least-squares error expression).
- Minimize by differentiating and equating the gradient to zero; this gives you a linear system to solve.
- Display the scalar function on the surface using a color map and overlay its gradient vectors when the user presses '3'.
- Plot the deviation between the input vector field and the solution scalar function's gradient (the "Poisson reconstruction error").

Note: the linear system is not full rank; K has a one dimensional nullspace corresponding to the constant function. This is because a scalar field can be offset by any constant value without altering its gradient. You will need to fix the value at one vertex (e.g., to zero) to solve the system.



(A) Vector Field (B) Scalar Field

FIGURE 2. A vector field and the reconstructed scalar function. Note the scalar function gradient's deviation from the input field.

Relevant `libigl` *functions:* `igl::grad` calculates the gradient matrix explained above. `igl::doublearea` can be used to compute triangle areas. `igl::sparse`, `igl::repdiag`, `igl::colon`, `igl::slice`, `igl::slice_into` might also be relevant here.

Required output for this section:

- Visualization of computed scalar function and its gradient.
- Plots of the Poisson reconstruction error
- An ASCII dump of the reconstructed scalar function ($\#V \times 1$ vector, one vertex value per row) for the mesh `irr4-cyl2.off` and the input constraints in `irr4-cyl2.constraints`.

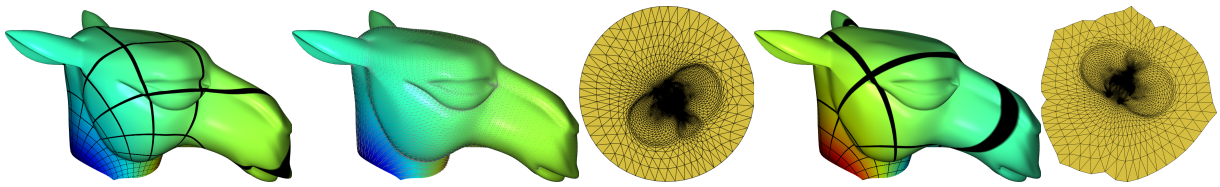
3. Harmonic and LSCM Parameterizations

For this task, you will experiment with flattening a mesh with a boundary onto the plane using two parameterization methods: *harmonic* and *Least Squares Conformal* (LSCM) parameterization. In both cases, two scalar fields, U and V, are computed over the mesh. The per-vertex (u, v) scalars defining these coordinate functions determine the vertices' flattened positions in the plane (the flattening is linearly interpolated within each triangle).

For the harmonic parametrization example, you will first map the mesh boundary to a unit circle in the UV plane centered at the origin. The boundary U and V coordinates are then "harmonically interpolated" into the interior by solving the Laplace equation with Dirichlet boundary conditions (setting the Laplacian of U equal to zero at each interior vertex, then doing the same for V). This involves two separate linear system solves (each with the same system matrix).

In LSCM, the boundary is free, with the exception of two vertices that must be fixed at two different locations in the UV-plane (to pin down a global position, rotation, and scaling factor). These vertices can be chosen arbitrarily. The process is again a linear system solve, but in this case the U and V functions are entwined into a single linear system.

- Use the `libigl` implementation of harmonic (key '4') and LSCM (key '5') mappings. Hitting these keys should trigger a visualization of one of the two mapping functions (U or V) as a color map over the surface, with the provided line texture overlaid.
- Compute the gradient of the selected mapping function and overlay it (key '6').



(A) Harmonic (B) Gradient of V (C) UV (D) LSCM (E) UV

Figure 3. A harmonic parameterization (A), the gradient of its V function (B) and a visualization of the flattened mesh on the UV plane (C). LSCM parameterization (D) and its UV domain (E).

Note: you may need to scale up the parameterization to better visualize the texture lines (parametric curves).

Relevant `libigl` *functions:* `igl::harmonic`, `igl::lscm`, `igl::boundary_loop`, `igl::map_vertices_to_circle`.

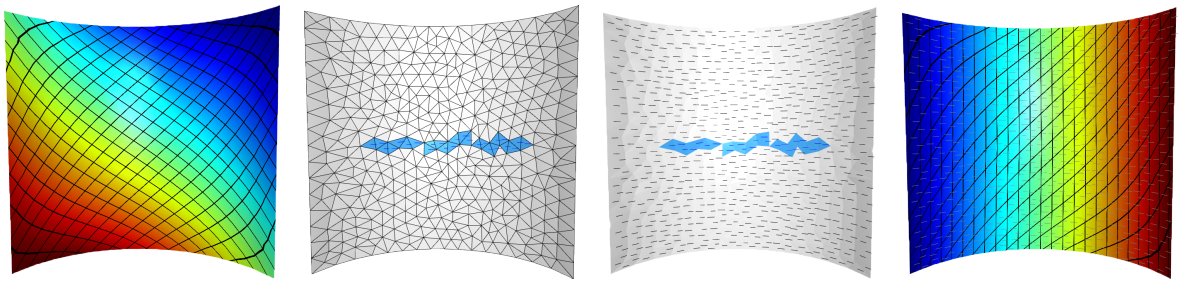
Required output for this section:

- Visualization of the computed mapping functions and their gradients for LSCM and harmonic mapping.

4. Editing a parameterization with vector fields

A parameterization consists of two scalar coordinate functions on the surface. As such, we can use vector fields to guide the parameterization: we can design a vector field and fit one of the coordinate functions' gradients to it.

4.1. **Editing the parameterization.** Starting with a harmonic/LSCM parameterization, use the results of the previous steps to replace one of the U, V functions with a function obtained from a smooth user-guided vector field. Visualize the resulting U or V replacement function and its gradient atop the mesh, and texture the mesh with the new parameterization (key '7').



(A) Harmonic (B) Constraints (C) Vector Field (D) Parameterization

FIGURE 4. An initial harmonic parameterization (A) is edited by designing a vector field. The user-provided constraints (B) are first interpolated (C), then a scalar function is reconstructed and used to replace the parameterization's V coordinate function. The new scalar function and its gradient are shown in (D) together with the newly textured mesh.

4.2. **Detecting problems with the parameterization.** It is possible for a parameterization created in this way to cause triangles to flip over as they are mapped into the UV plane. Determine a reliable criterion for detecting flipped triangles and visualize the planar mapped mesh with the flipped faces highlighted in red (key '8').

Required output for this section:

- Visualization of the edited parameterization.

- Visualization of flipped elements.
- An ASCII dump of the flipped triangle indices (if any) resulting from an edited harmonic parameterization of the mesh `irr4-cyl2.off`, where the parameterization's V coordinate is replaced with a scalar field designed from the gradient vector constraints provided in `irr4-cyl2.constraints`.