Assignment 2: Implicit Surface Reconstruction

In this exercise you will

- Compute an implicit MLS function approximating a 3D point cloud with given (but possibly unnormalized) normals.
- Sample the implicit function on a 3D volumetric grid.
- Apply the marching cubes algorithm to extract a triangle mesh of this zero level set.
- Experiment with various MLS reconstruction parameters.

Your main task is to construct an implicit function $f(\mathbf{x})$ defined on all $\mathbf{x} \in \mathbb{R}^3$ whose zero level set contains (or at least passes near) each input point. That is, for every point $\mathbf{p}_i$ in the point cloud, we want $f(\mathbf{p}_i) = 0$. Furthermore, ∇f (the isosurface normal) evaluated at each point cloud location should approximate the point's normal provided as input.

You will construct f by interpolating a set of target values, d_i, at "constraint locations," $\mathbf{c}_i$. The MLS interpolant is defined by minimization of the form $f(\mathbf{x}) = \operatorname{argmin}_\phi \sum_i w(\mathbf{c}_i, \mathbf{x})(\phi(\mathbf{c}_i) - d_i)^2$, where $\phi(\mathbf{x})$ lies in the space of admissible function (e.g., multivariate polynomials up to some degree) and w is a weight function that prioritizes each constraint equation depending on the evaluation point, $\mathbf{x}$.

Note: The datasets provided actually already include triangles. Ignore them for this assignment.

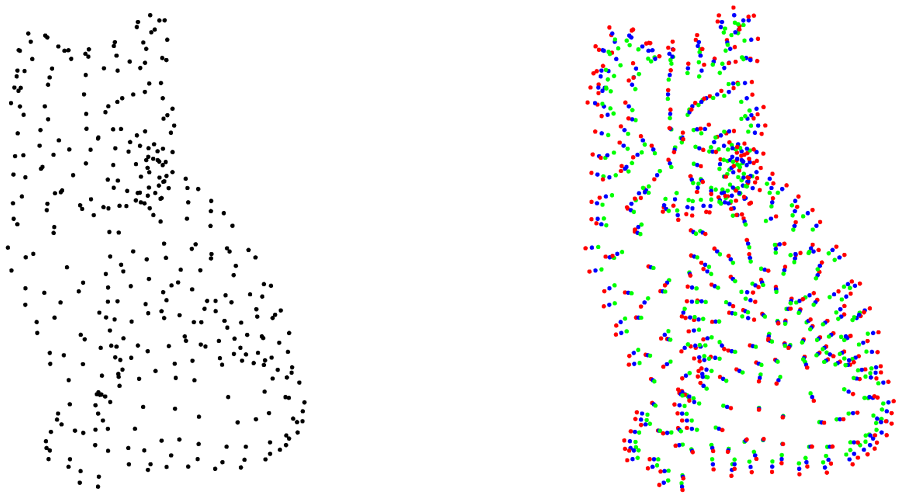
1. Setting up the Constraints

Your first step is thus to build the set of constraint equations by choosing constraint locations and values. Naturally, each point $\mathbf{p}_i$ in the input point cloud should contribute a constraint with target value $d_i = 0$. But these constraints alone provide no information to distinguish the object's inside (where we want $f < 0$) from its outside (where we want $f > 0$). Even worse, the minimization is likely to find the trivial solution $f = 0$ (if it lies in the space of admissible functions). To address these problems, we introduce additional constraints incorporating information from the normals as follows:

- For each point $\mathbf{p}_i$ in the point cloud, add a constraint of the form $f(\mathbf{p}_i) = 0$.
- Fix an ε value, for instance $\varepsilon = 0.01 \times \mathrm{bounding\_box\_diagonal}$.
- For each point $\mathbf{p}_i$ compute $\mathbf{p}_{i+N} = \mathbf{p}_i + \varepsilon \mathbf{n}_i$, where $\mathbf{n}_i$ is the normalized normal of $\mathbf{p}_i$. Check that $\mathbf{p}_i$ is the closest point to $\mathbf{p}_{i+N}$; if not, halve ε and recompute $\mathbf{p}_{i+N}$ until this is the case. Then, add another constraint equation: $f(\mathbf{p}_{i+N}) = \varepsilon$.
- Repeat the same process for $-\epsilon$, i.e., add equations of the form $f(\mathbf{p}_{i+2N}) = -\epsilon$. Do not forget to check each time that $\mathbf{p}_i$ is the closest point to $\mathbf{p}_{i+2N}$.

After these steps, you should have $3n$ equations for the implicit function $f(\mathbf{x})$.

1.1. **Creating the constraints.** For this task, you need to complete the appropriate sections (keyboard callback, keys '1' and '2') of `src/main.cpp`. Pressing key '1' displays the input point cloud (this part has already been completed). When key '2' is pressed, you must calculate and display the constraints (points and implicit function values), storing them in `constrained_points`, `constrained_values`.



(A) Input point cloud (B) Inward/Outward value constraints

FIGURE 1. Input point cloud for the *cat* mesh and inward/outward value constraints. In Fig. 1b, labels green, red and blue correspond to inside, outside and on the surface respectively. The blue points in Fig. 1b are the same as the black ones in Fig. 1a.

You should plot each constraint point in a color chosen based on its type (inside/outside/surface) as in Figure 1.

Relevant `libigl` *functions:* None.

1.2. **Implementing a spatial index to accelerate neighbor calculations.** To construct the MLS equations, you will perform queries to find, for a query point $\mathbf{q}$:

- the closest input point to $\mathbf{q}$ (needed while constructing inside/outside offset points); and
- all input points within distance h of $\mathbf{q}$ (needed to select constraints with nonzero weight).

Although a simple loop over all points could answer these queries, it would be slow for large point clouds. Improve the efficiency by implementing a simple spatial index (a uniform grid at some resolution). By this, we mean binning vertices into their enclosing grid cells and restricting the neighbor queries to visit only the grid cells that could possibly satisfy the query. You can debug this data structure by ensuring that it agrees with the brute-force for loop implementation.

Relevant `libigl` *functions:* None.

Required output of Section 1.

- Visualization of the constrained points.

2. MLS Interpolation

We now use MLS interpolation to construct an implicit function satisfying the constraints as nearly as possible. We won't define the function with an explicit formula; instead we characterize it as the linear combination of polynomial basis functions that best satisfies the constraints in some sense. At a given point $\mathbf{x}$, you evaluate this function by finding the "optimal" basis function coefficients (which will vary from point to point!) and using these to combine the basis function values at $\mathbf{x}$.

Complete the appropriate source code sections (inside the keyboard callback, key '3') to evaluate the MLS function at every node of a regular volumetric grid containing the point cloud. As an example, the provided code computes the grid values for an implicit function representing a sphere (MLS wasn't used in this case since the formula is known analytically). It corresponds to the point cloud `sphere.off`. For a result using MLS see Fig. 2 (A).

2.1. **Create a grid sampling the 3D space.** Create a regular volumetric grid around your point cloud: compute the axis-aligned bounding box of the point cloud, enlarge it slightly, and divide it into uniform cells (cubes). The grid resolution is configured by the global variable `resolution`, which can be changed in the GUI sidebar. The marching cubes library needs to know the location of each grid point, which you should store in the `grid_points` array. The library expects the points to be ordered lexicographically by their (z, y, x) grid index; see the example grid construction provided in `createGrid`.

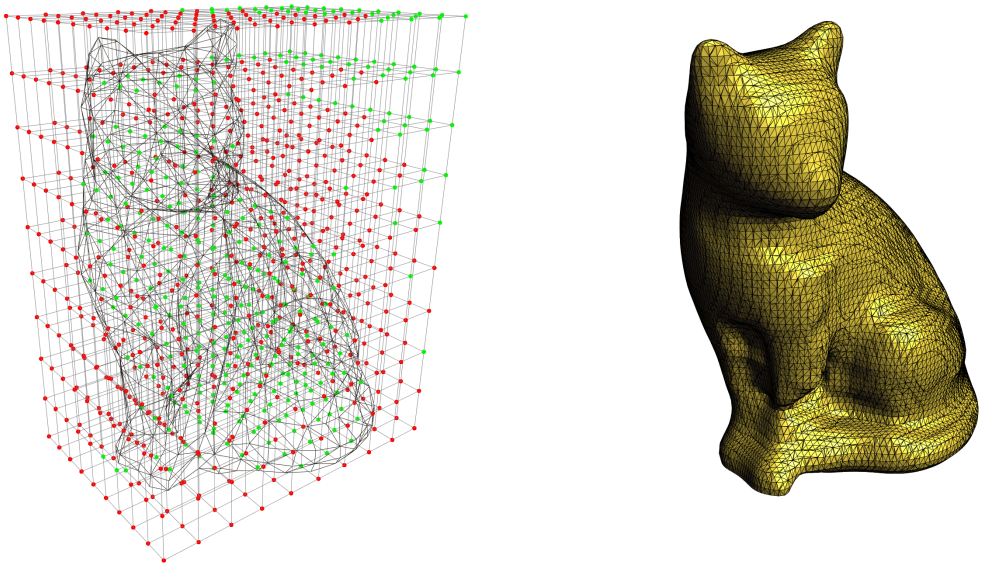
Relevant `libigl` *functions:* `igl::colon` can optionally be used to generate points.

2.2. **Evaluate the implicit function on the grid.** For each grid node of the grid, evaluate the implicit function $f(\mathbf{x})$, whose zero level set approximates the point cloud. Use the moving least squares approximation presented in class and in the tutoring session. You should use the Wendland weight function with radius configured by `wendlandRadius` and degree $k = 0, 1, 2$ polynomial basis functions configured by `polyDegree` (add these parameters to the GUI!). Only use the constraint points with nonzero weight (which you can find efficiently using the spatial index you created earlier). If no constraint points are within `wendlandRadius` of the evaluation grid point, you can assign a large positive (outside) value to the grid point.

Store the field value in the `grid_values` array, using the same ordering as in `grid_points`. Render these values by coloring each grid point red/green as already done for the sphere function (see Figure 2a). You can use the global variables `grid_colors` and `grid_lines` to store the colors and the lines of your grid. Code for displaying the grid is already provided (see function `getLines` and the callback function).

Relevant `libigl` *functions:* `igl::slice` might come in handy to extract the relevant constraint locations/values.

2.3. **Using a non-axis-aligned grid.** The point cloud `luigi.off` is not aligned with the canonical axes. Running reconstruction on an axis-aligned grid is wasteful in this case: many of the grid points will lie far outside the object. Devise an automatic (and general) way to align the grid to the data and implement it.



(A) Implicit function values at grid points

(B) Marching Cubes result

FIGURE 2. MLS implicit function constructed from the value constraints shown in Fig. 1 and final reconstruction using marching cubes. In Fig. 2a, green and red labels correspond to negative and positive values of the implicit function respectively.

Required output of this section:

- Screenshots of the grid with nodes colored according to their implicit function values.

3. EXTRACTING THE SURFACE

You can now use marching cubes to extract the zero isosurface from your grid. The extraction has already been implemented and the surface is displayed when key '4' is pressed. Add code to export your mesh in off format. For an example result, see Fig. 2 (B).

Relevant `libigl` *functions:* `igl::copyleft::marching_cubes`, `igl::writeOFF`.

Required output of this section:

- Screenshots of the reconstructed surfaces. Experiment with different parameter settings: grid resolution (also anisotropic in the 3 axes), Wendland function radius, polynomial degree. Add all these settings to the GUI to ease experimentation.
- The reconstructed model in `off` format for every point-cloud dataset provided.

Optional tasks

(1) *(2 points)* Compute the closed-form gradient of the MLS approximation. Suggestion: A good strategy to solve this exercise is to write MLS explicitly in matrix form and then compute its gradient (a good reference for differentiating expressions with matrices can be found in "The Matrix Cookbook").

(2) *(2 points)* In "Interpolating and Approximating Implicit Surfaces from Polygon Soup," normals are used differently to define the implicit surface. Instead of generating new sample points offset in the positive and negative normal directions, the paper uses the normal to define a linear function for each point cloud point: the signed distance to the tangent plane at the point. Then the values of these linear functions are interpolated by MLS. Implement Section 3.3 of the paper and append to your report a description of the method and how it compares to the original point-value-based approach.

(3) *(1 point)* Screened Poisson Surface Reconstruction is a more modern technique that avoids some of the pitfalls of local reconstruction methods.

An implementation is provided in MeshLab. A standalone implementation of this method is also provided by the authors here with accompanying usage instructions and datasets. Compare your MLS reconstruction results to the surfaces obtained with this method, and try to understand the differences. Report your findings.