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 tetrahedra 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 (they are used just to estimate surface normals at vertices).

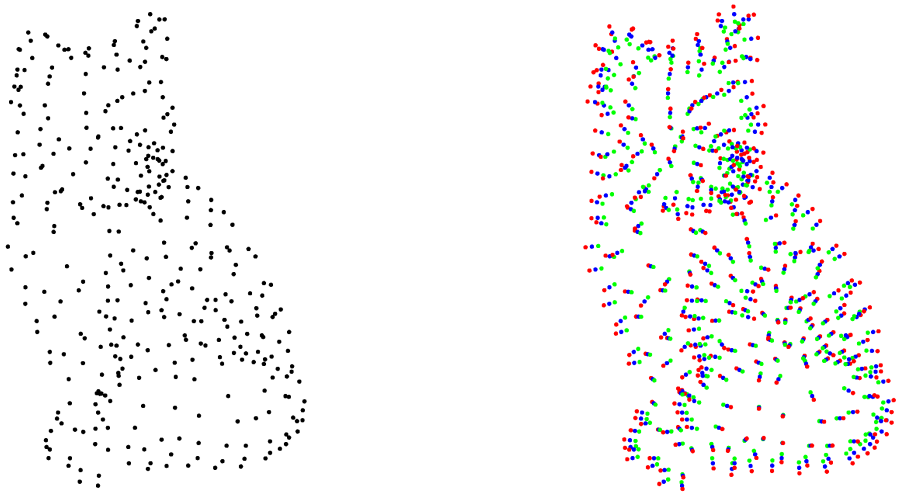
1. Setting up the Constraints

Your first step is to build the set of constraint equations by choosing constraint locations and values. 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 $f(\mathbf{p}_i) = d_i = 0$.
- Fix an ε value, e.g. $\varepsilon = 0.01 \times \text{bounding\_box\_diagonal}$.
- 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 at $\mathbf{p}_i$. 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 $f(\mathbf{p}_{i+N}) = d_{i+N} = \varepsilon$.
- Repeat the same process for $-\epsilon$, i.e., add equations of the form $f(\mathbf{p}_{i+2N}) = d_{i+2N} = -\epsilon$. Do not forget to check each time that $\mathbf{p}_i$ is the closest point to $\mathbf{p}_{i+2N}$.
- Append the three vectors $\mathbf{p}_i$, $\mathbf{p}_{i+N}$ and $\mathbf{p}_{i+2N}$ and their corresponding values f to unique vectors $\mathbf{p}$ and $\mathbf{f}$, respectively.

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

Important: explicitly write a function `find_closed_point(point, points)` that retrieves the index of the closest point to `point` in `points`. You can start with a naive brute-force search. For an efficient implementation based on a spatial index see the Optional tasks below.



(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.

Relevant numpy functions: `max, min, argmin, linalg.norm, zeros`.

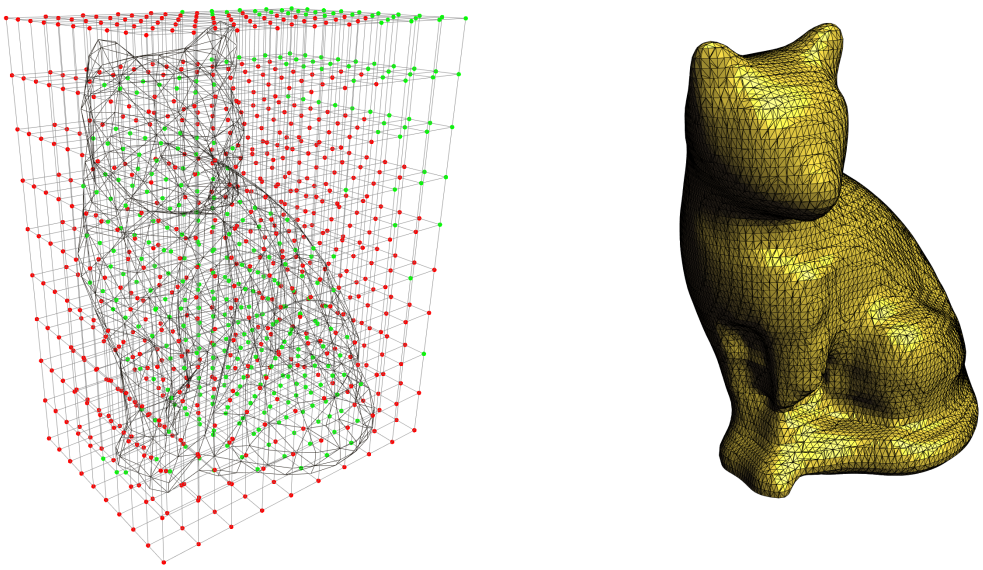
Required output of this section:

- Screenshot (using meshplot) of the point cloud with additional points shaded with green, blue, and red as in Fig. 1. In meshplot, use parameter `c=col` where `col` is a vector of RGB colors of the same size of the point cloud.

2. USE MLS INTERPOLATION TO EXTEND FUNCTION f

The provided mockup code computes and visualizes the grid values for an implicit function representing a sphere (MLS wasn't used in this case since the formula is known analytically). You shall substitute that code with the proper code, implementing the steps explained in the following. For a result using MLS see Fig. 2 (A). It is sufficient to visualize the nodes of the grid colored properly; grid edges and the superimposed mesh are optional.

2.1. **Create a grid sampling the 3D space.** Create a regular volumetric grid of tetrahedra around your point cloud: compute the axis-aligned bounding box of the point cloud, enlarge it slightly, and divide it into uniform cells (cubes), each of which is divided into six tetrahedra in turn. The grid resolution is configured by the global variable `resolution`, which can be changed. Note that the function `tet_grid` to generate the grid is provided. We call the grid vertices `x` and the tets connecting them `T`.



(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.

2.2. **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 `xi` in $\mathbf{x}$, 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 `xi`.

Complete the appropriate source code sections to evaluate the MLS function at every node `xi` of a regular volumetric grid containing the input point cloud. More specifically, for each grid node of the grid, evaluate the implicit function $f(\texttt{xi})$, 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`. Only use the constraint points with nonzero weight (i.e., points `p` with `||xi - p|| < wendlandRadius`). If the number of constraint points is less than twice the number of polynomial coefficients, (i.e., 1 for `k = 0`, 4 for `k = 1`, and 10 for `k = 2`), you can assign a large positive (outside) value to the grid point.

Store the field value `fx = f(xi)` in a numpy.array, using the same ordering as in `x`. Render these values by coloring each grid point red/green depending if they are inside/outside (i.e., depending on the sign of `fx`. You can use `meshplot.plot(.../ c=color)` where `color` is a `n x 3` matrix containing

RGB values. 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).

Important: explicitly write a function `closest_points(point, points, h)` that retrieves the indices all points in `points` that are at distance less than `h` from `point`. Similar to the previous: implement brute-force search first; optimized version with spatial index is optional.

Relevant numpy functions: `argwhere, diag, linalg.solve`.

Required output of this section:

- Screenshots of the grid points $\mathbf{x}$ colored according of being inside or outside the input cloud.

3. Extracting the surface

You can now use marching tets to extract the zero isosurface from your grid. The extraction has already been implemented and the surface is displayed. The implicit function obtained from MLS might be noisy and the reconstructed mesh will contain several pieces. Filter out and keep the largest component. For an example result, see Fig. 2 (B).

Relevant `igl` *functions:* `marching_tets`, `face_components`.

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, and polynomial degree.

Optional tasks

3.1. **Optional Task 1 - 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. **Hint:** You may use a 3D array of buckets, where each entry contains an array of indices of input points within the bucket. Or you may want to experiment with a hash table, since most buckets will be empty.

3.2. **Optional Task 2 - 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. **Hint:** Principal component analysis of the point cloud can provide

the directions for the box. You can alternatively rotate the input points to align such directions with the cardinal direction, and use an AABB as before; or build the grid aligned with the eigenvectors of the PCA.