CGGS Coursework 1 (Geometry): Reconstruction from Point Clouds

Amir Vaxman

Errata

- Section 1: The minimium number of points is $|a|$ (the number of polynomial coefficients), not $N+1$).
- Section 1: advice to use SVD rather than LDLT.

Introduction

This is the 1^{st} coursework for the "geometry" choice. The purpose is to write `C++` code for reconstructing a mesh, through an implicit function, from a set of points with oriented normals. Our algorithm of choice is that of *Moving Least Squares* (MLS), mostly following the material of Lecture 6. Visualization will be done with PolyScope as in Practical 0. The concrete objectives are:

- Implement the basic MLS algorithm (scalar polynomial) for reconstruction from a small set of points and examine both the function and the resulting mesh.
- Implement the alternative version (blending functions).
- Extend the method to become more efficient.

General guidelines

The practical uses the same Eigen + PolyScope setup as in Practical 0. Your job is essentially to complete the functions `compute_scalar_mls()`, for Section 1, and `compute_function_mls()` for Section 2. Both of which are in the eponymous header files. The `main.cpp` code in the respective subprojects will call on them, and operate the visualization. Reconstruction is controlled by a few parameters:

1. `gridSize`: the resolution of a grid in space that contains the point cloud.
2. `h`: the support parameter for the Wendland function.

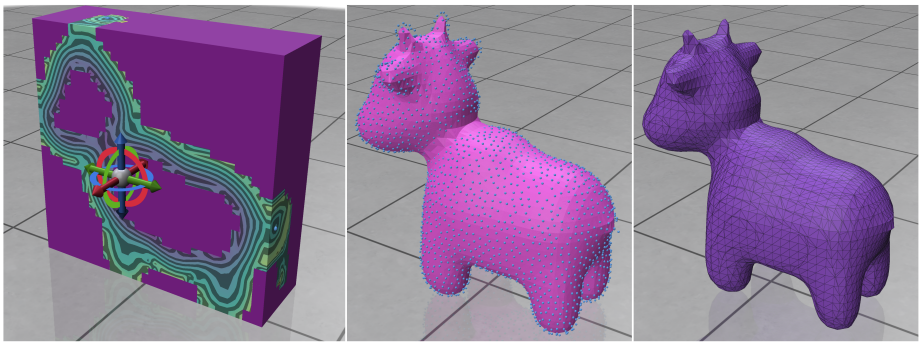

Figure 1: Reconstruction of the `Spot-2000` mesh. Left: the (plane-sliced) implicit function. Middle: the reconstructed mesh with the input point clouds. Right: the mesh with edges marked on it, showing visible artefacts of the marching cubes algorithm. Parameters are `h=0.05, N=1, gridRes=32, epsNormal = 0.01`.

3. `epsNormal`: the ϵ parameter that controls off-surface points (just for Section 1).
4. `N`: the degree of the polynomial (just for Section 1).

ϵ and h are automatically scaled according to the mesh diagonal, you don't have to do that. Point clouds (`pointCloud`) are loaded from `.OFF` files, and each has an auxiliary normal (`pcNormals`). Polyscope will present the grid, with the options of showing the isosurface (zero level set), and slicing it with a plane (see Fig. 1). You can then effectively see both the implicit result, and the explicit results (Polyscope automatically runs marching cubes for you).

1 Scalar Polynomial MLS

Given a set of points $\{p_i\}$ (as `inputPoints`) with corresponding oriented normals (as `inputNormals`), your task is to fit a polynomial $F(x, y, z)$ of degree N to the original and auxiliary $\pm\epsilon$ off-surface points:

$$F(x, y, z) = \sum_{0 \geq i,j,k, i+j+k \leq N} a_{ijk} x^i y^j z^k$$

for each point on the grid (that's as many different polynomials as grid points!). `compute_scalar_mls()` gets `gridLocations`, which is a matrix of all grid locations. You are returning `MLSValues`, which is the evaluation of the polynomial you computed at that point (by its coordinates). For each point, you will be solving a least-squares system, as explained in class and the lecture nodes. For the weights, you should use the Wendland function with the parameter `h`. You should *only* use the points for which $r \leq h$ (the distance from the grid point to the point-cloud point). If you don't have enough points, that is $\geq |a|$, the

number of coefficients a, so that you have at least as many equations as variables, then you should put a `nanValue`. You must do that since the grader checks that all the NaNs are in the correct place. The envelope code will automatically generate the evaluation points on a lattice, feed them to PolyScope, and run the marching cubes algorithm to obtain a mesh. See Fig. 2 for examples.

Grading: The `Grading1` subproject will thoroughly run the code against all examples in the `data` folder, and combinations of parameters which it will print. You will gain points for each correct result, and your automatic-grader grade will be shown eventually. You should, as the grader does, experiment with different values of ϵ, h, N, and the grid resolution.

The automatic grader for this section will constitute 30% of your grade; an extra 10% will be graded on a report of your (concise) insights about the effects of parameter exploration on the result. This includes experimentation with different weight functions: Gaussian and singular edge function (with the same ϵ). Include imagery and explanations of what you see. That means this section will bring you to a total of 40%. Use `SVD` solver to solve the least-squares system, as it mitigates tiny weights by producing a minimum 2-norm solution.

Important Note: The grader will time your functions, and will fail a test if its duration is more than 10× the duration of our code (we compile your code on the same machine so it's a fair comparison). If your code contains infinite loops or anything that gets stuck indefinitely, we will **fail the automatic part of the section altogether**. This is to prevent you from using extremely bad code-writing practices.

Optimization tip: You'll notice that the powers $x^i y^j z^k$ only depend on the point-cloud coordinates and not the grid point you are currently on. That means you can first compute *all* of them for the points cloud (given N), and only then do a loop on all grid points, rather than recompute these costly powers over and over again. Also, avoid constructing the diagonal W explicitly, and instead multiply the rows of C by the scalar weights. Finally, avoid excessive computations of r that involve expensive square roots: you can compare r^2 to h^2 equivalently. An even better optimization is to form the difference vector $\vec{r}$, where $r = ||\vec{r}||^2$, and first check that $max(|r|) < h$ (the maximum absolute coefficient of $\vec{r}$) before checking the norm; it's like bounding the sphere with a box.

2 Function-Blending MLS

We next implement the alternative version of MLS, where we define distance functions per point i in the point cloud, and then blend them using the Wendland

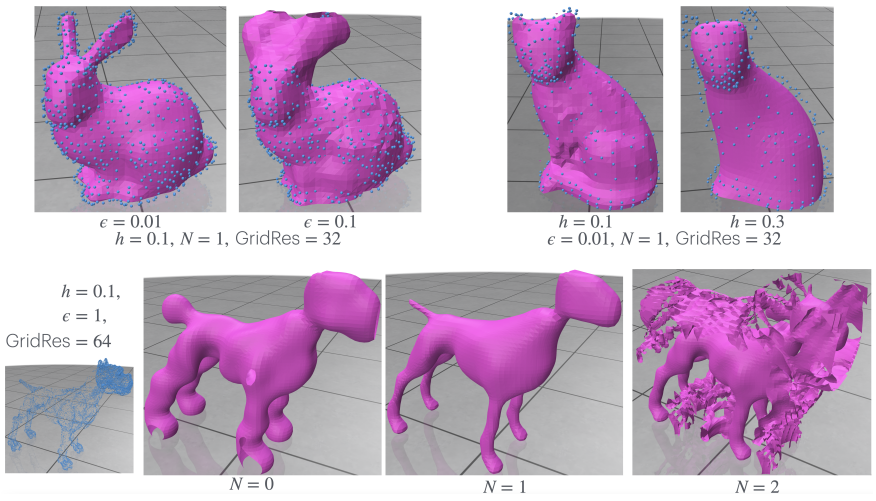

Figure 2: Ablating the scalar MLS with ϵ (top left), h (top right), and N (bottom). For ϵ, a large value results in over-connectivity across the ears. For h, a large value results in over-smoothing. For N, a small value results in poor reconstruction, while $N = 2$ results in relative adherence to features, but in many "floaters" due to non-linearity.

functions as a partition of unity:

$$F(x, y, z) = \frac{\sum_i \phi(|(x, y, z) - p_i|) n_i^T ((x, y, z) - p_i)}{\sum_i \phi(|(x, y, z) - p_i|)}.$$

This is equivalent to MLS with $N = 0$, but blending functions instead of scalar values. This will also not use $\pm\epsilon$ points, so it's a more lightweight algorithm without linear systems. Otherwise, you still follow all the rules of Section 1, including NaN values for when there are no points in which $r \geq h$.

Grading The grading for this Section is worth 30%, where 20% are automatically graded with subproject `Grading2`, and 10% on a report that details your results. Compare between the results of Section 1 and 2 and give your insights when one is better than other, if anything.

3 Efficient MLS

The bottleneck in the algorithm is the need to look up every pair of grid-point and point-cloud point, which is likely a double loop in your code. This becomes unsustainable for big point clouds and fine grids. In the extension section, worth the distinction extra 30%, you should devise an efficient way to work this out. This will likely include some sort of efficient spatial lookup structure. To get

a good score on this, you will need to research a proper way to do it, and demonstrate a considerable scaleup (for instance, with at least $100K$ points and a grid of resolution 128). Demonstrate this in your report (there is no automatic grader). You should also look up nice, and a bit more massive, point clouds to demonstrate as your input. You should submit full code to reproduce your results.

4 Submission

The report must be at most 3-pages long (all sections together) including all figures. All functionality should be done by changing the function bodies in the respective header files. You must **only** submit these header files; we will ignore any other input for Sections 1 and 2. For Section 3, you are welcome to submit anything relevant that will help us reproduce your result.

The submission will be in the official place on Learn, where you should submit a ZIP file code and the PDF of the report.