CGGS Coursework 1 (Geometry): Reconstruction from Point Clouds

Amir Vaxman, Thomas M. Walker

Errata

- Currently None

Introduction

This is the 1^{st} coursework for the "geometry" choice. The purpose is to write Python code for reconstructing a mesh, through an implicit function, from a set of points with oriented normals. Our algorithm of choice is that of *Radial Basis Function Reconstruction*, mostly following the material of Lecture 6, which is a revised and simplified version of [1]. Visualization will be done with PolyScope as in Practical 0. The concrete objectives are:

- Implement the basic RBF algorithm for reconstruction from a small set of points and examine both the function and the resulting mesh.
- Extend the algorithm by adding different RBF basis functions and a global polynomial.
- Explore solving the problem in least-squares with centre reduction.
- Sparsify the computation to be able to scale to larger point clouds.

General guidelines

Please update to Python 3.11; some features of PolyScope depend on it.
The practical uses the same Python + PolyScope setup as in Practical 0. In addition, we make use of the `scikit-image` package for marching cubes, and some other sub-packages of `scipy` such as `scipy.spatial` for computing pairwise distances efficiently, and `scipy.sparse` for handling sparse linear algebra.

Your job is essentially to complete the functions `compute_RBF_weights()`, which sets up the RBF system and computes the weights, and the consequent `evaluate_RBF()` that evaluates the resulting implicit function on input locations in space. They have some default parameters which are supposed to induce the

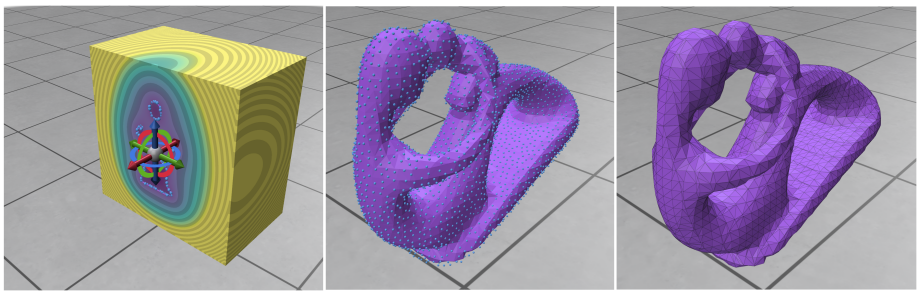

Figure 1: Reconstruction of the `fertility-2500` mesh. Left: the (plane-sliced) implicit function. Middle: the marching-cubes mesh with the input point clouds. Right: the mesh with edges marked on it, showing visible artefacts of the marching cubes algorithm.

basic behaviour (of Section 1). Altering them "unlocks" the further features of the more advanced sections.

You are also provided with the `Reconstruction.py` script that deals with the loading of point clouds, visualization, and meshing. Explore its code documentation and options to see how you modify it to your settings—it does not get graded, but rather only the functions above.

1 The Basic Version

Given a set of points $\{p_i\}$ (as `inputPoints`) with corresponding oriented normals (as `inputNormals`), a choice of RBF $\varphi(r)$ (as `RBFFunction`), and the off-points coefficient ϵ (as `epsilon`). You are to use `compute_RBF_weights()` to return the weights w (as `w`), and centres c_i (as `RBFCentres`) that solve the RBF interpolation system $Aw = b$ (Lecture 6 slide 26). You should create $\pm\epsilon$ off-surface points, and feed them to the function. The ordering has to be all original points, then all $+\epsilon$, then all $-\epsilon$ for the grader. The centres of the RBFs will be exactly the data points in that same ordering.

Next, you need to implement the function `evaluate_RBF()` that accepts a set of evaluation points `xyz`, the RBF centres, chosen basis function `RBFFunction`, and computed weights `w`, and evaluates the implicit function F (returned as `values`). The script will automatically generate the evaluation points on a lattice, feed them to PolyScope, and run the marching cubes algorithm to obtain a mesh.

Solving this would produce the results you see in Figure 1. Note that PolyScope shows both the implicit function (that you can slice into), and a marching cubes result[1].

[1]PolyScope has an "isosurface" option for its implicit functions, but we found it to be buggy, so only rely on the one we use from `scikit-image`.

Grading: Grading will be first done by a perfect result (and pro-rata wrong results) obtained from the grading script `BasicVersion.py` in the `grading` folder. You should, as the grader does, experiment with different values of ϵ, and different choices of $\phi(r)$: try the following:

1. Biharmonic spline: $\varphi(r) = r$.
2. Triharmonic spline: $\varphi(r) = r^3$
3. Wendland function: $\varphi(r) = \frac{1}{12}(1-\beta r)_+ \cdot (1-3\beta r)$. For this, experiment with different β values. To be able to feed the β hyperparameter through to the computation and evaluation functions without changing their signature, use `functools.partial` to create a function argument that only accepts r.

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 with regards to the effects of parameter exploration on the result; include imagery and explanations of what you see. That means this section will bring you to a total of 40%. The grader will take all options default and therefore should not invoke any extensions—you should take care that any such extensions, in the next Sections, only work with non-default arguments.

Coding advice: use `scipy.spatial.distance.cdist` to compute a matrix of pairwise distances r_{ij} between data points and centres, and then "activate" them by sending this matrix to $\phi(r)$. Here you can assume the resulting matrix is dense and square, so you can solve it directly or by using LU decomposition through `scipy.linalg.lu_factor` and `scipy.linalg.lu_solve`.

2 Extension: adding a global polynomial

In several situations, we would like to get *polynomial precision*. That is, to be able to reproduce point clouds that are (more or less) coming from a polynomial of degree l:

$$q_l(x,y,z) = \sum_{0\leq i,j,k,\ i+j+k\leq l} a_{ijk} x^i y^j z^k.$$

This will for instance reproduce perfect planes for $l = 1$. It moreover has the potential of reducing artefacts from spurious floating zero sets (see Figure 2). As such, we are after the following augmented implicit function $F(p = x, y, z)$:

$$F(p) = q_l(p) + \sum_{n=1}^{3N} w_n \varphi(|p - c_n|)$$

Our job is to find the parameters $\{a_{ijk}\}$ alongside the weights w_n that we found in the basic formulation, where we need again to fit all original and off-surface points. To form the equations, we denote L as the number of all possible permutations of

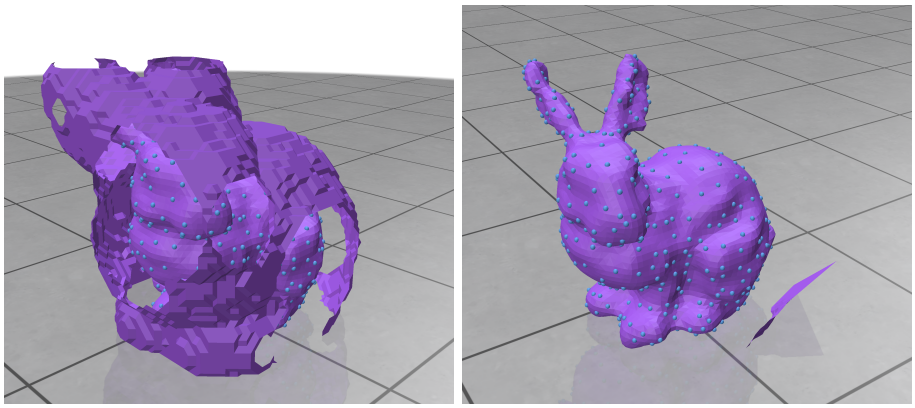

Figure 2: The basic result with Wendland function (left) has many spurious zero sets, since the functions decay far enough from the object. Adding an $l = 1$-degree polynomial (right), i.e., $a_{000} + a_{100}x + a_{010}y + a_{001}z$ helped resolve these zero sets (though notice the single "floater" for the zero hyperplane of the polynomial).

indices ijk (this is 4 for $l = 1$, where they are $\{(0,0,0),(1,0,0),(0,1,0),(0,0,1)\}$, and 10 for $l = 2$, for instance). Then we construct the matrix $Q : 3N \times L$ as follows:

$$Q = \begin{pmatrix} 1 & \cdots & x_0^i y_0^j z_0^k & \cdots \\ 1 & \cdots & x_1^i y_1^j z_1^k & \cdots \\ 1 & \cdots & x_N^i y_N^j z_N^k & \cdots \end{pmatrix} \tag{1}$$

Thus, for a vector of coefficients $a = (a_{ijk})$ (ordered arbitrarily, but consistent to the column order in Q), we have that $Q \cdot a$ is the vector of values $q(p_n)$ for all centres $n \in [1, N]$. Thus, we have that:

$$F(p_n) = b = Q \cdot a + A \cdot w.$$

One problem is that we added L more variables for a total of $3N + L$ variables, but only have $3N$ equations. However, we also need to make sure that the problem is well-posed; that is, that there is a single solution, where the space represented by Q is orthogonal to that represented by A. We achieve that by demanding $Q^T w = 0$. Then, our new entire system for this task is:

$$\begin{pmatrix} A & Q \\ Q^T & 0 \end{pmatrix} \begin{pmatrix} w \\ a \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}$$

You should "unlock" this feature by providing a value for the parameter `l` other than the default -1, for both computation and evaluation functions. See Figure 2 for different results. Specifically, try the inputs `plane` and `hyperbolic-paraboloid` that are perfect samplings of $l = 1$ and $l = 2$ zero-set surfaces, respectively; you'll see a major difference with Wendland RBFs.

Grading The grading for this feature is worth 15%, where 10% are by a perfect result in the grading script `PolynomialPrecision.py`, and 5% are for the analysis in the report—choose examples from the benchmark that clearly demonstrates this benefit.

Coding advice: Write a small function that for a given l will generate all index triplets $\{ijk\}$. `numpy.meshgrid` will come in handy. Remember the order doesn't matter here. Use `numpy.block` to create block matrices, use the same A from Section 1, where only Q is new.

3 Extension: Centre Reduction

Using all original and off-centre points is not always a great idea. First, the system is $3N \times 3N$ and dense. Second, we cannot set ϵ too small, since the 3 equations associated with a single original point become very similar to each other, and by so greatly deteriorating the conditioning of matrix A. Furthermore, we sometimes don't even want to use all points, since it might be a waste if we have many of them covering a rather smoothly-varying surface. We thus allow the case where we have M RBF centres (=functions), and our usual $3N$ data points, and where $M \leq 3N$. As such, our matrix A is now of sizes $3N \times M$ and is overdetermined; meaning that we will not interpolate the data points exactly. We solve the system using least squares:

$$A^T A w = A^T b.$$

There are two possible reductions that we invoke. One is through the argument `UsingOffPoints`, which is `True` by default. if given `False`, you should not create RBF centres for the $\pm\epsilon$ points, and use N centres for the original points alone. *Independently*, the argument `RBFCentreIndices` allows you to prescribe a subset of indices from $[0, N-1]$ which will be used as centres. The default empty value makes the function use the full set of N. Note that the behaviour of `usingOffPoints` should be set to include (or exclude) the $\pm\epsilon$ points *only* of the prescribed centres in `RBFCentreIndices`.

You can assume this feature is mutually exclusive with that of Section 2, and that is since the definition there assumes a full square matrix, which takes a bit of effort to generalize to this case. Also note that, unlike the cited paper [1], we do not keep A square by also disregarding the data points, but rather always consider *all* data points.

Grading: The grading for this feature is worth 20%, where 10% is based on the automatic grading of script `CentreReduction.py` and 10% on the description of a feature in the report. You should compare all possible options and draw insights from them. To exemplify the effect of `RBFCentreIndices`, write a script that chooses a small subset of original points randomly, and iteratively augments them with new points for which the fitting error is the largest until some tolerance in size or error is achieved.

4 Sparsification

This section is a more advanced extension of the practical. If we use the Wendland RBF, A can technically be sparse since the RBF support is local. This might allow us to scale to bigger point clouds. That means you need to write a version of the computation (that is called from within the main computation function; don't change signatures, as it will invalidate the grader for the other sections!) that efficiently creates a sparse A and solves the system with sparse solvers. This should be triggered by setting the argument `sparsify=True`.

One challenge is to study and apply sparse solvers in Python; another is to avoid creating the dense matrix of 0s to begin with, since it might be prohibitive. For this, you will have to think of a way to avoid computing all pairwise distances. This will probably be in the form of using loops and aggregating values; however remember that loops are slow in Python, and you should use at most a single loop on a small variable (like the number of centres), and vector operations on the big number (number of points).

This section will not be automatically graded since the results may vary on the design decisions here. The worth of this section is 25%, and it will be manually graded upon the convincing ability of the sparsification to both solve the problem and scale for bigger points cloud. General tip: if you can make it into the 10-50K centres it's great; it's not likely that you can do more without more specialized methods, but you are welcome to surprise! You'll find some big point clouds in the data folder (note that any point cloud above 4500 points is automatically skipped by all graders).

5 Submission

The report must be at most 3-pages long including all figures. All functionality should be done by changing the function bodies in `ReconstructionFunctions.py`. The submission will be in the official place on Learn, where you should submit a ZIP file of only the affected code scripts, any auxiliary data you would like to share, and a PDF of your report.

References

[1] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 67–76, 2001.