Mesh reconstruction from point cloud

Authors: Simone Piazza, Federica Valentini, Eugenio Varetti

September 2023

1. Introduction

Surface reconstruction from point cloud consists in a collection of algorithms that allow to obtain a continuous surface starting from a discrete set of points. When the point cloud given as input is regular and smooth, most of the methods work well and produce an accurate reconstruction of the mesh at issue. However, in practical applications the point cloud is usually obtained from scanning and this procedure causes the appearance of imperfections; in this cases, the geometry becomes really complex and many existing algorithms may fail in these critical points [5].
The aim of this work is to develop a robust method for reconstructing meshes starting from a point cloud, even in conditions characterized by complex geometries.

2. Surface mesh reconstruction methods

We first review the main algorithms and their open source implementations for the reconstruction of a mesh from a 3D point cloud: Ball Pivoting Algorithm (BPA) and Poisson Surface Reconstruction.

2.1. Ball Pivoting Algorithm

The Ball Pivoting Algorithm [3] is a method used to create a surface mesh from a set of unordered points representing a 3D object or surface; by using the idea of a ball rolling along the points, the algorithm efficiently constructs a triangular mesh that approximates the original surface.
The concept behind BPA is indeed as follows: imagine a 3D ball with a user-defined radius ρ being placed on a point cloud, beginning from a seed triangle. When the ball comes into contact with any three points in the cloud (without passing through those points), it creates a new triangle. The algorithm then pivots from the edges of the existing triangles, and each time it encounters another set of three points where the ball doesn't pass through, it generates an additional triangle. This process continues until all reachable edges have been attempted, and then moves on to start from a different seed triangle, repeating the process until all points in the point cloud have been considered. A 2D-sketch of how the algorithm works is shown in Figure 1(a), where a circle of radius ρ pivots from sample point to sample point, connecting them with edges.
It's worth noting that the success and quality of the BPA depend on the density and distribution of the input points in the point cloud and the choice of the initial seed triangle. Proper parameter tuning is essential to ensure accurate and reliable results in generating the surface mesh. Two possible related issues are shown in Figure 1(b)-(c): in (b) it is shown that if the sampling density is too low, some of the edges will not be created, leaving holes; in (c) it is instead represented the situation where the curvature of the manifold is greater than $1/\rho$, thus some of the sample points will not be reached by the pivoting ball, and features will be missed.

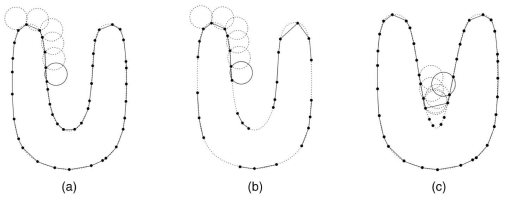


Figure 1: Sketch of the BPA applied to a 2D manifold [3].

2.2. Poisson Surface Reconstruction Algorithm

The Poisson surface reconstruction algorithm [6] employs an implicit function framework to address the surface reconstruction problem. In summary, the Poisson surface reconstruction algorithm leverages the indicator function and its

gradient to estimate the surface of the model from oriented point samples. By reconstructing the surface using this implicit function framework, the algorithm can handle noise and irregular point cloud data more effectively than BPA, resulting in a smoother and more accurate surface representation (see A.2 for more details).
The main approach involves computing a 3D indicator function, denoted as χ, which equals 1 at points inside the model and 0 at points outside. The reconstructed surface is then extracted based on an appropriate isosurface.
The core idea is that there exists an integral relationship between oriented points sampled from the surface of a model (Figure 2, the first) and the indicator function of the model (Figure 2, the third). Specifically, the gradient of the indicator function forms a vector field that is nearly zero almost everywhere (since the indicator function is constant almost everywhere), except at points near the surface. In those regions, the gradient of the indicator function aligns with the inward surface normal (Figure 2, the second). Consequently, the oriented point samples can be regarded as samples of the gradient of the model's indicator function.
The problem of computing the indicator function thus reduces to inverting the gradient operator, i.e. finding the scalar function χ whose gradient best approximates a vector field $\vec{V}$ defined by the samples, i.e.

$$\min_{\chi} ||\nabla\chi - \vec{V}||. \tag{1}$$

The variational problem (1) can be transformed into a standard Poisson problem, as derived by Kazhdan et al. in [6]; applying the divergence operator, the goal becomes to compute a scalar function χ such that:

$$\Delta\chi = \nabla \cdot \vec{V}. \tag{2}$$

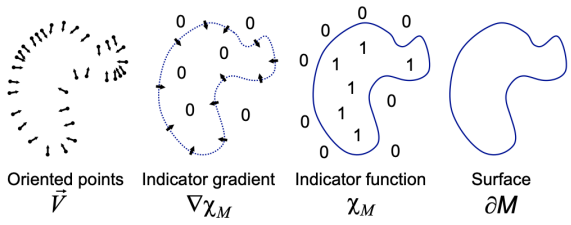


Figure 2: Poisson surface reconstruction in 2D [6].

2.3. Numerical results

Each algorithm is tested on two different point clouds: the first, very smooth and regular, is sampled from a sphere; the second is sampled from the so-called Stanford bunny [1], which represent a challenging benchmark case in computer graphics. The library we refer to is `Open3d` [7] and two different algorithms are tested on the point clouds at our disposal: Ball Pivoting Algorithm (BPA) and Poisson Surface Reconstruction.

Both the BPA and the Poisson algorithm can be easily tested on Python, since the library `Open3d` offers already implemented methods that reproduce the algorithms introduced above. Both the methods require the normals in each point of the cloud as input: in a first attempt, we tested the methods giving as input the exact normals generated with the open source software `Paraview` (applying the filters `ExtractSurface` and `GenerateSurfaceNormals`); then, the normals were estimated taking advantage of the already implemented methods `estimate_normals` and `orient_normals_consistent_tangent_plane`.
Both the methods are tested on the sphere point cloud and on the bunny point cloud, first with the exact normals and then with the estimated ones.
When using the exact normals all the `Open3d` algorithms perform well, in both point clouds, with the Poisson method (with `depth` = 8) that is a bit more accurate with respect to the BPA (with `radius` = 0.75 * `avg_dist`). While for the sphere case this hold true also when the normals are estimated, in the case of the bunny some normals on the left ear are not pointing outwards as it should be. This behavior reflects in an inaccurate mesh reconstruction in that area, both with the BPA and the Poisson algorithm.

(a) Bunny estimated normals. (b) Bunny mesh with BPA. (c) Bunny mesh with Poisson.

Figure 3: Third case: the bunny with estimated normals.

Given these results, the focus of our work is on the Poisson method for mesh reconstruction. This choice is driven by its significant benefits in generating smooth meshes, especially in complex geometries. While we acknowledge that the estimation of normals can be challenging in this method, our project is specifically focused on addressing this aspect. Our aim is to improve the accuracy and reliability of normal estimation within the Poisson method, thereby enhancing the overall quality of the reconstructed meshes.

3. Numerical reconstruction of the surface normals

This section introduces two important algorithms for processing point cloud data. The first one estimates surface normals for each point while the second ensures a consistent orientation of these normals across the surface. These techniques are fundamental to several applications, including 3D reconstruction and object recognition. This section explores their implementation and demonstrates their effectiveness through visual representations.

3.1. Estimating Normal direction

The method of normal estimation is based on the idea of identifying local neighborhoods and finding a basis to describe them, following a PCA-like approach.

Specifically, for each point i in the point cloud:

1. Identify the set $\Omega_K^{(i)}$ consisting of the point itself and its K neighbors (the default value of the algorithm is $K = 30$).
2. Compute the covariance matrix Σ_i relative to $\Omega_K^{(i)}$.
3. Decompose $\Sigma_i \in \mathbb{R}^{3\times3}$ into eigenvalues-eigenvectors in order to obtain eigenvectors associated with the principal directions of the "local" point cloud.
4. Select the eigenvector associated with the smallest eigenvalue and assign it as the normal of point i.

In Figure 4 a 2D sketch of the explained procedure is reported, with the point i displayed in red and its corresponding normal represented as the orange arrow $\mathbf{v}$.

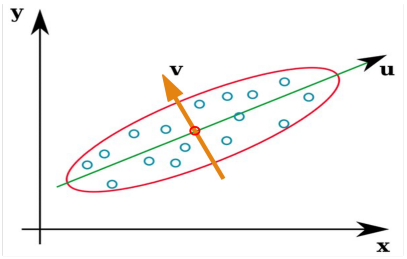


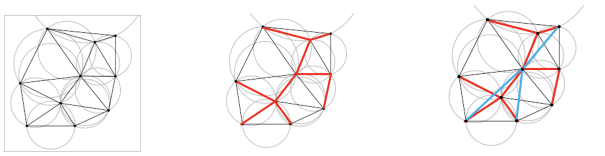
Figure 4: Schematic representation (in 2D) of the `Open3d` algorithm for the estimation of the normals.

3.2. Orienting Normal orientation

Once the normals in the point cloud have been estimated, another method is required to consistently orient the normals to the point cloud (as reported in Section 3.3 of [4]). This operation requires an input parameter, an integer K, which represents the number of neighbors each point should consider. The algorithm executes the following steps:

1. create a Delaunay graph from the point cloud where edges weights are the Euclidean distance between points;
2. find the minimum spanning tree with Kruskal algorithm (see Appendix A.1);
3. for every point, find its K nearest (in terms of Euclidean distance) neighbors and add them to the resulting graph if they were not present in the initial Delaunay graph;
4. for every pair of connected points i, j substitute the weights with: $normal\_weights = 1 - |n_i \cdot n_j|$;
5. extract the final minimum spanning tree with Kruskal algorithm;
6. set an initial point - the one with the highest z coordinate - and orient its normal n_0 according to z-versor $\hat{e_z} = (0, 0, 1)$ i.e such that $n_0 \cdot \hat{e_z} > 0$
7. orient all the normals in the point cloud following the tree, i.e by ensuring that the dot product between the father node and the child nodes is greater than 0.

In Figure 5 the first part of the above-mentioned algorithm is applied to a simple case, as an example. The final graph is given by the union of the red one (from Kruskal algorithm) and the blue one (from KNN algorithm).



(a) Delaunay graph.

(b) MST obtained with Kruskal algorithm.

(c) Final graph with KNN algorithm (K = 4).

Figure 5: From the point cloud to the graph used to orient the normals.

4. Our contributions to Open3d

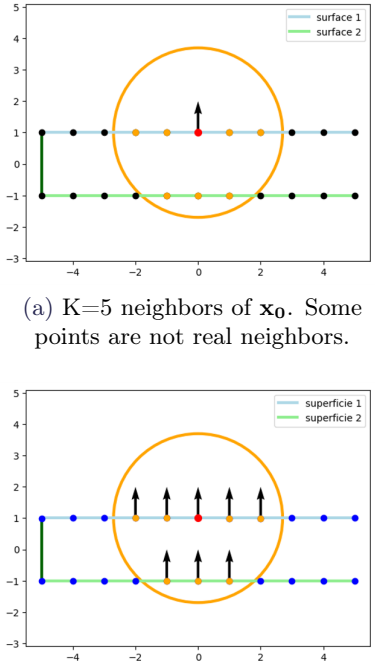
As showed by the results reported in the Section 2, when the point cloud becomes more complex the algorithm for the orientation of the normals fails, and consequently the reconstructed mesh is inaccurate. After a deep analysis of the function `orient_normals_consistent_tangent_plane`, whose key points have been reported in subsection 3.2, we identified two main criticalities:

1. considering simply the Euclidean distance may be inaccurate in some situations;
2. taking as initial point the one with the highest z coordinate may cause problems in complex meshes.

4.0.1 Selection of the metric

As it is widely known in statistics, the Euclidean distance is not always the optimal metric for expressing the similarity between two points. This holds true in the context of our numerical test cases.

Let's consider a simple 2D geometry consisting of two parallel surfaces, denoted as S_1 and S_2, as illustrated in Figure 6a. Taking the reference point $\mathbf{x_0}$ (depicted in red) on surface S_1, we observe that when considering the first K nearest neighbors based on the Euclidean distance, points on surface S_2 are misclassified and incorrectly identified as neighbors of $\mathbf{x_0}$. This misclassification leads to an inaccurate orientation of the normals associated with the lower surface, as depicted in Figure 6b. This issue extends to 3D metrics, even in the case of more complex geometries or critical points such as cusps or stenosis.



(a) K=5 neighbors of $\mathbf{x_0}$. Some points are not real neighbors.

(b) Misorientation of the normals.

Figure 6: Illustration of the misclassification issue in the Euclidean distance metric.

To address this issue, an alternative metric can be considered to accurately identify the true neighbors of a point $\mathbf{x_0}$ in such geometries. Assuming the normal direction is correct, we can utilize the tangent plane defined by $\mathbf{x_0}$ and its normal $\mathbf{n_0}$. We define the distance between any point $\mathbf{x}$ and $\mathbf{x_0}$ as follows:

$$dist(\mathbf{x}, \mathbf{x_0}) = ||\mathbf{x} - \mathbf{x_0}||_2 + \lambda \, |(\mathbf{x} - \mathbf{x_0}) \cdot \mathbf{n_0}|$$

where $\lambda \in \mathbb{R}_{\geq 0}$. In this metric, the parameter λ penalizes the distance between $\mathbf{x}$ and the tangent plane. Specifically, as λ increases, the distance between $\mathbf{x_0}$ and a point $\mathbf{x}$ not lying on the plane becomes greater. Graphically, the "pseudosphere" - which, by definition, has points on its surface equidistant from the center - tends to flatten along the plane's axis, as depicted in Figure 7.

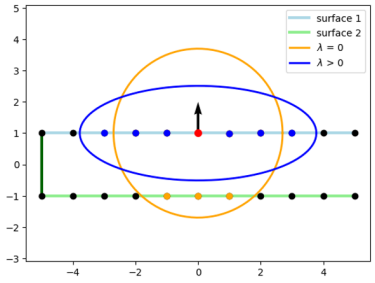


Figure 7: New concept of distance. Blue points are now the K nearest neighbours

An alternative solution, which can also be combined with the previous one if necessary, is to exclude neighbors that do not satisfy certain constraints. These constraints can be specific to the problem at hand and depend on the characteristics of the data or the desired properties of the neighbors.
For example, in the context of a 2D/3D geometry, we can exclude neighbors that violate a certain angle threshold with respect to the normal direction. This means that only points within a certain range of angles from the normal vector will be considered as valid neighbors. By imposing such constraints, we can refine the set of neighbors and ensure that only relevant points are included.
In particular, let $\mathbf{x_0}$ be the reference point, $\mathbf{n_0}$ its associated normal, and $\mathbf{x} \in \Omega$ be any point in the point cloud. The set of potential neighbors of point $\mathbf{x_0}$, is defined as:

$$\Omega_{\alpha_0}^{(0)} = \{\mathbf{x} \in \Omega : \frac{|(\mathbf{x} - \mathbf{x_0}) \cdot \mathbf{n_0}|}{||\mathbf{x} - \mathbf{x_0}||_2} \leq cos(\alpha_0)\}.$$

where α_0 is the threshold angle[1]. As shown in Figure 8, for each point in the domain, the space is partitioned into two subsets: an acceptance region where neighbors can be selected, and a rejection region where, if neighbors exist, they will not be considered.

[1]Given the two vectors $\mathbf{p}$ and $\mathbf{q}$, it holds $\mathbf{p} \cdot \mathbf{q} = |\mathbf{p}||\mathbf{q}| \cos(\alpha)$.

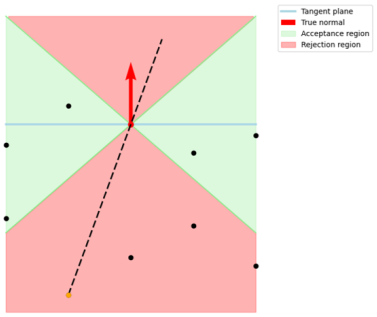


Figure 8: Only points within the acceptance region (green) are candidate neighbors.

This approach can be further extended to include other constraints based on specific properties or conditions that need to be satisfied by the neighbors. By carefully defining and applying these constraints, we can enhance the accuracy and reliability of the neighbor selection process. Combining multiple approaches, such as incorporating a modified distance metric along with constraint-based filtering, can provide a comprehensive solution to accurately identify the true neighbors in complex geometries or critical point scenarios. The specific combination and adjustment of these techniques would depend on the characteristics of the data and the requirements of the problem at hand.

4.0.2 Selection of the starting point

Once the `orient_normals_consistent_tangent_plane` function defines the minimum spanning tree (MST), it initiates the correct normal orientation process. Starting from the point with the highest z-coordinate, the algorithm aligns its normal vector to have a positive dot product with the z-axis $(0, 0, 1)$. Subsequently, it follows a path determined by the connections of the starting point within the MST. Along this path, the algorithm sequentially orients the normals of the nodes to ensure a positive dot product with the normal of the calling node. However, the first stage of this method can encounter issues, particularly in the upper regions of the mesh, where irregularities and uneven point distribution are more prevalent. Specifically, when starting from these regions, there is a risk of incorrectly orienting the first normal, leading to a subsequent flip of all the normals by 180 degrees. As a result, this propagation of inaccurately oriented normals along

the connected path significantly undermines the overall quality of mesh reconstruction.
To reduce the likelihood of encountering this issue, we decided to initiate the orientation of the normals from the base of the point cloud with respect to the z-axis. Similar to the original algorithm, which consistently oriented the first point towards the vector (0,0,1), we now consistently orient the first point towards (0,0,-1).

5. Our implementation

In order to apply the methods discussed in 4.0.1 and 4.0.2, we had to make changes to the source code of the `Open3D` library [7]. Specifically, we modified the `OrientNormalsConsistentTangentPlane` function, whose original algorithm is described in subsection 3.2. We extended the list of inputs by adding `lambda` and `cos_alpha_tol` to the parameter list, assigning default values of 0 and 1, respectively, to preserve the original functionality of the library. We refined the construction of the minimum spanning tree (MST) used in the normal orientation process.

Firstly, we modified the edge weights of the Delaunay Graph. Consistently with Section 4.0.1, we introduced a penalty term to the original weight, dependent on `lambda`, and a constraint on the relative positioning of two vertices on an edge, given by `cos_alpha_tol`.
Subsequently, during the expansion phase of the MST generated by the Kruskal's algorithm (see A.1), we introduced two constraints. After finding the K-nearest neighbors (KNN) of a point p in the point cloud, we check if, for each found neighbor p_k, where k=1,...,K:

- the edge defined by the two points (p, p_k) is not already present in the set of edges of the Delaunay Graph;
- the point p_k resides within the acceptance region, i.e.,

$$\frac{|(\mathbf{p}-\mathbf{p_k})\cdot\mathbf{n_0}|}{||\mathbf{p}-\mathbf{p_k}||_2} \leq \texttt{cos\_alpha\_tol}; \quad (3)$$

- the point p_k is not an outlier in terms of distance from the tangent plane defined by p and n_0 within the neighbor set $\Omega_K^{(p)} = \{p_k\}_{k=1}^K$, i.e.,

$$|(\mathbf{p}-\mathbf{p_k})\cdot\mathbf{n_0}| \leq Q_3 + 1.5 * IQR,$$

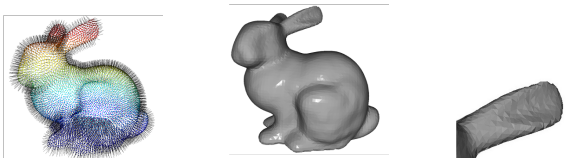
where Q_3 and IQR are the third quartile and interquartile range, respectively, of the set $\Omega_K^{(p)}$.

In such cases, the edge (p, p_k) will be added to the resulting graph.

Secondly, we modified the starting point of the `traversal_queue`, a queue used to orient the normals following the final MST; as described in Section 4.0.2, indeed, we start from the point with the lowest z-coordinate, and then orient all the other normals accordingly.
An overview of the modified algorithm is presented in A.3. For more detailed information, the modified code can be found in our GitHub fork at: `https://github.com/eugeniovaretti/Open3D`.

6. Numerical results

As confirmed by the mesh obtained as output, our changes to the original algorithm perform well if tested on the point cloud considered in this work. Thanks to the correct orientation of the outward normals in all the regions, indeed, the Poisson surface reconstruction algorithm returns as output an accurate mesh, also in the area that presented criticalities before. The results reported in Figure 9 have been obtained running our new code; the output mesh is comparable both following a pure penalization approach (with $\lambda = 10$) and imposing an angle threshold (with $\alpha = 60° \rightarrow \cos\alpha = 0.5$).



(a) Point cloud with estimated normals.

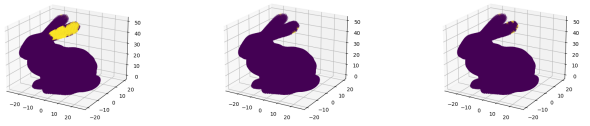
(b) Mesh given as output by Poisson algorithm.

(c) Zoom on the critical region of the ear.

Figure 9: Poisson reconstruction using the normals estimated with our improvements.

In order to have full awareness of the obtained results and to appreciate more the improvements to the existing library, some comparisons on the number of misoriented normals in the different cases is performed. We consider a normal as *misoriented* when the real normal

and the estimated one form an angle greater than 90°.
We depict in yellow all the points in the cloud whose normal has been misoriented by the algorithm, while in purple all the remaining points, for which we consider the orientation of the normal as acceptable. From Figure 10 we can conclude that both our approaches perform well, with the number of misoriented normals that has considerably decreased with respect to the original situation.



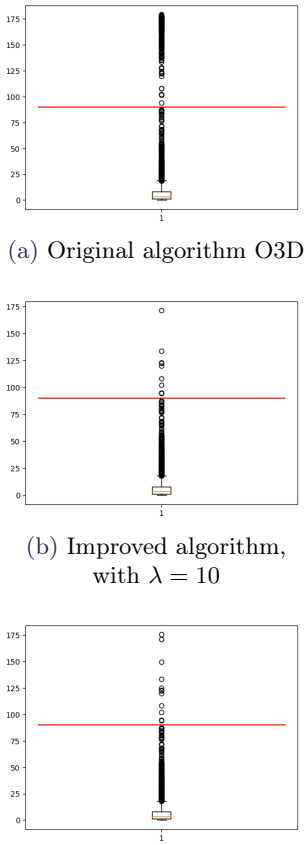
(a) Original algorithm O3D

(b) Improved algorithm, with $\lambda = 10$

(c) Improved algorithm, with $cos(\alpha_0) = 0.5$

Figure 10: Visualization of misoriented normals.

From a more quantitative point of view, instead, the error angle for each point of the cloud is reported in a boxplot, for all the three cases (Figure 11). It is clear how the number of outliers have significantly reduced from the original Open3d implementation to our modifications, both in the "λ-approach" and in the "cosα-approach". If we fix also here 90° as threshold for considering a normal as misoriented, the number of mistaken normals passes from 169 in the original algorithm to 9 and 12 in our implementations with λ and cosα, respectively.



(a) Original algorithm O3D

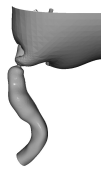
(b) Improved algorithm, with $\lambda = 10$

(c) Improved algorithm, with $cos(\alpha_0) = 0.5$

Figure 11: Boxplots of the error angles in the three cases, with the threshold of 90°.

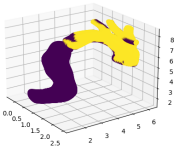
Once the analysis of the pathological bunny point cloud was completed, we conducted additional tests on the algorithms using a different test case that, according to our preliminary analysis, could present challenges for the original algorithm: the reconstruction of an aortic coarctation, a geometry available in the *Vascular Model Repository* [2] .

Since the original dataset was very dense (approximately 62,000 points), we imposed a high number K of neighbors to stress-test the algorithm.
With the same value of K, the results were consistent with our expectations, demonstrating the robustness of the modified algorithm. As shown in Figure 12, the stenosis point caused orientation problems that propagated throughout the entire domain.

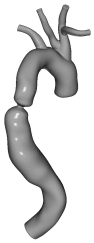


(a) Original algorithm O3D.

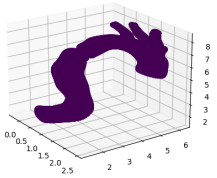
(b) Original algorithm O3D.



(c) Misoriented normals.



(d) Modified algorithm with $\lambda = 10$, $cos(\alpha_0) = 0.5$



(e) Misoriented normals.

Figure 12: Results for the original and improved algorithm.

7. Conclusions and future works

The main improvement brought about by our new algorithm is illustrated in Figure 11: when we compare the number of nearest neighbors, K, our extension of the algorithm outperforms the original. Specifically, by defining as "misoriented" those normals misaligned by an angle greater than 90° with respect to the exact normal, in the original algorithm, there are 169 misoriented normals, whereas in our implementations with λ and $\cos\alpha$, they are only 9 and 12, respectively.

This is the primary advantage of the new algorithm: it exhibits more consistency and independence from the chosen parameter K, which would typically require a trial and error procedure.

Possible future work might target the limitation represented by the fact that the library heavily relies on the class `KDTree` (K-dim tree); the choice of the parameters that this class takes as input, first of all the number of neighbours K, considerably influences the mesh that is given as output. However, the correct choice of such parameters is not trivial at all and moreover its modification would require a deep knowledge of the library that only expert users have; thus, usually the default value of `knn = 30` is kept irrespective of the characteristics of the specific point cloud, even if it could give inaccurate outputs. A possible improvement can thus be to allow the user to give the number of `knn`s as input in an user-friendly way; in this manner, the choice of the best parameter for the `KDTree` class is eased a lot, with improvements in the final result.

Moreover, concerning the choice of the starting point for the orientation of all the normals, we changed it from the one with the maximum z coordinate to the one with the minimum z coordinate; we are aware that this may not be the ideal solution as it still heavily relies on the orientation of the geometry in space. Introducing a parameter to reverse the orientation of all normals could address this problem, but it would remain a manual solution requiring users to recognize the algorithm's weakness and adjust it interactively. Therefore, we have identified two potential automated solutions.

The first solution involves implementing an automatic check for orientation from within the volume to be reconstructed. In this case, the main challenge would be to accurately identify the interior of the volume and subsequently perform consistent checks on important neighboring points. One way to envision the resolution is to imagine a balloon inflating from the inside, touching the innermost points of the point cloud.

The second solution is to identify a sufficiently regular point on the mesh and consistently orient its normal. The challenge here lies in determining how to measure local regularity in a point cloud; some ideas are for example to look at the local curvature in each point or at local density of the points in the cloud.

Overall, these alternatives aim to mitigate the problem without relying solely on user intervention, offering more automated approaches to handle the orientation of normals.

References

[1] The Stanford 3D Scanning Repository.

[2] Vascular model. `https://www.vascularmodel.com/`. Accessed: June 16, 2023.

[3] Fausto Bernardini, J. Mittleman, Holly Rushmeier, Cláudio Silva, and Gabriel Taubin. The ball-pivoting algorithm for surface reconstruction. *Visualization and Computer Graphics, IEEE Transactions on*, 5:349 – 359, 11 1999.

[4] H. Hoppe, T. Derose, T. Duchamp, J. Mcdonald, and W. Stuet-zle. Surface reconstruction from unorganized point clouds. 1992.

[5] Zhangjin Huang, Yuxin Wen, Zihao Wang, Jinjuan Ren, and Kui Jia. Surface reconstruction from point clouds: A survey and a benchmark. *Eurographics Symposium on Geometry Processing*, 2022.

[6] M. Kazhdan, M. Bolitho, and H. Hoppe. Poisson surface reconstruction. *Eurographics Symposium on Geometry Processing*, 2006.

[7] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

A. Appendix

A.1. Kruskal Algorithm

Kruskal's algorithm is used to reduce the initial graph to the connected one that minimizes the total weight. The algorithm is reported in Algorithm 1.

Algorithm 1 Kruskal's Algorithm

1: Sort all the edges of the graph in non-decreasing order of their weights.
2: Create a separate set for each vertex of the graph. Initially, each vertex is its own set.
3: **while** MST has fewer than $|V|-1$ edges **do**
4: Consider the next edge, e, in the sorted order.
5: **if** Adding edge e does not create a cycle **then**
6: Add edge e to the Minimum Spanning Tree (MST).
7: Merge the sets of the two vertices of the edge.
8: **end if**
9: **end while**

A.2. From normals to mesh

The Poisson surface reconstruction algorithm, as explained, begins by solving the Poisson problem, taking into account the surface normals, to compute a 3D indicator function χ. This indicator function distinguishes points inside the model (assigned a value of 1) from points outside (assigned a value of 0). The reconstructed surface is obtained by extracting the isosurface of the indicator function corresponding to a specific threshold, capturing the shape and geometry of the underlying surface.
Once the indicator function is computed, the algorithm applies the marching cubes technique. This technique divides the volumetric data into small cubes and examines the scalar values at the vertices of each cube, considering the information from the indicator function and the surface normals. Based on these values, the algorithm determines the configuration of each cube and uses a lookup table to determine the appropriate triangulation for its surface.
By repeating this process for all cubes in the volumetric data, the algorithm generates a mesh representation that closely approximates the reconstructed surface.

A.3. Final Algorithm

Algorithm 2 OrientNormalsConsistentTangentPlane(int K, double λ, double $cos(\alpha_0)$) - Extended version

1: **Input:** Point cloud with estimated normals, K, λ, $cos(\alpha_0)$
2: Create a Delaunay graph where edge weights are the Penalized Euclidean distance between points.
3: Find the minimum spanning tree with Kruskal's algorithm.
4: **for** every point p in the point cloud **do**
5: Find its K nearest neighbors (in terms of Euclidean distance).
6: **for** every edge (p, p_k) **do**
7: Add the edge to the resulting graph if:
- the edge (p, p_k) is not (already) present in the initial Delaunay graph;
- the point p_k respects Property (3)
- the point p_k is not an outlier in terms of distance from the plane wrt the set of K neighbours

8: **end for**
9: **end for**
10: **for** every pair of connected points i and j **do**
11: Substitute the edge weights with $1 - |n_i \cdot n_j|$, where n_i and n_j are the normals at points i and j, respectively.
12: **end for**
13: Extract the final minimum spanning tree with Kruskal's algorithm.
14: Set an initial point x_0 as the one with the lowest z coordinate.
15: Orient its normal n_0 such that $n_0 \cdot \hat{e}_z > 0$, where $\hat{e}_z = (0, 0, -1)$.
16:
17: **for** every node in the minimum spanning tree (except the initial point) **do**
18: Orient the normal of the current node by ensuring that the dot product between the parent node's normal n_p and the current node's normal n_i is greater than 0:

$$n_p \cdot n_i > 0$$

19: **end for**
20: **Output:** Point cloud with consistently oriented normals.