POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Final project report

Mesh reconstruction from point cloud

Scientific computing tools for advanced mathematical modelling

Authors: Simone Piazza, Federica Valentini and Eugenio Varetti

Academic year: 2022-2023

---

1. Research of already existent open source algorithms

The first part of the project is devoted to the research of already existent open source algorithms for the reconstruction of a mesh from a 3D point cloud. Each algorithm is then tested on two different point clouds: the first one, very smooth and regular, represents a sphere; the second one represents a more involved geometry, in particular a bunny. Two main libraries were used and tested on the point clouds at our disposal: `Pyvista` and `Open3d`.

1.1. Libraries introduction - Mathematical Framework

1.1.1 `Pyvista` library

The `Pyvista` library utilizes a method that progresses through three main phases: initial surface estimation, mesh optimization, and piecewise smooth surface optimization.
The algorithm begins by constructing a local neighborhood graph based on the proximity of points, capturing their connectivity information. From this, an initial surface approximation is created by connecting neighboring points with triangular faces.
Through an iterative refinement process, the algorithm aims to preserve the local surface characteristics while interpolating the input points and maintaining smoothness. This is achieved through techniques such as vertex relocation and smoothing operations, which adjust the surface geometry to better fit the input points.
The refinement process continues until a satisfactory surface representation is obtained, closely approximating the underlying geometry of the unorganized point cloud.
For more comprehensive information, refer to the PhD thesis by Hugues Hoppe [2].

1.1.2 `Open3d` library: Ball pivoting algorithm and Poisson surface reconstruction

In the `Open3d` library, two different algorithms were analyzed: Ball Pivoting Algorithm (BPA) and Poisson surface reconstruction.

The idea behind **BPA** is the following: a 3D ball with a user-specified radius ρ is dropped on the point cloud, starting from a seed triangle. If it hits any 3 points (and it does not fall through those 3 points) it creates a triangle. Then, the algorithm starts pivoting from the edges of the existing triangles and every time it hits 3 points where the ball does not fall through, another triangle is created. The process continues until all reachable edges have been tried, and then starts from another seed triangle, until all points have been considered.
A 2D-sketch of how the algorithm works is reported in Figure 1(a), where a circle of radius ρ pivots from sample point to sample point, connecting them with edges. In Figure 1(b)-(c), instead, two possible related issues are presented: in (b) it is shown that when the sampling density is too low, some of the edges will not be created,

leaving holes; in (c) it is instead represented the situation when the curvature of the manifold is larger than $1/\rho$, thus some of the sample points will not be reached by the pivoting ball, and features will be missed.

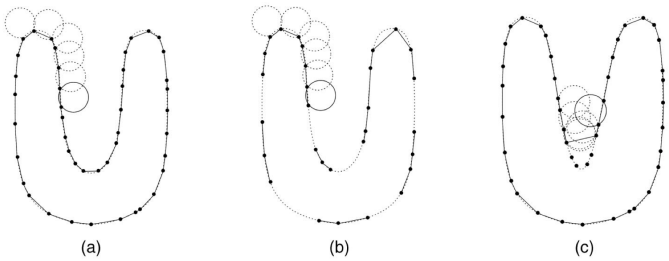


Figure 1: BPA in 2D.

In the **Poisson surface reconstruction** algorithm, instead, an implicit function framework is used to address the problem of surface reconstruction. Specifically, a 3D indicator function χ is computed (defined as 1 at points inside the model, and 0 at points outside), and then the reconstructed surface is obtained by extracting an appropriate isosurface.
The key idea is that there is 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 is a vector field that is zero almost everywhere (since the indicator function is constant almost everywhere), except at points near the surface, where it is equal to the inward surface normal (Figure 2, the second). Thus, the oriented point samples can be viewed 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.

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

If we apply the divergence operator, this variational problem (1) transforms into a standard Poisson problem: compute the scalar function χ such that:

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

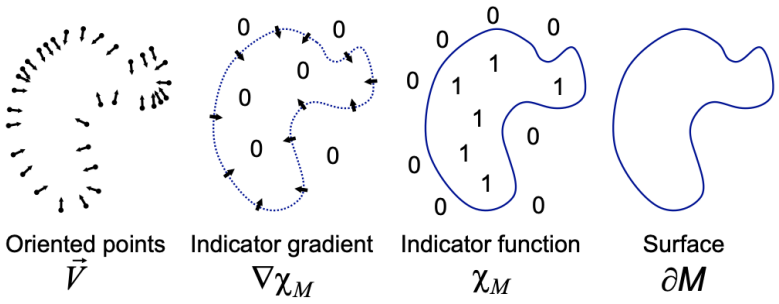


Figure 2: Poisson surface reconstruction in 2D.

1.2. Numerical results

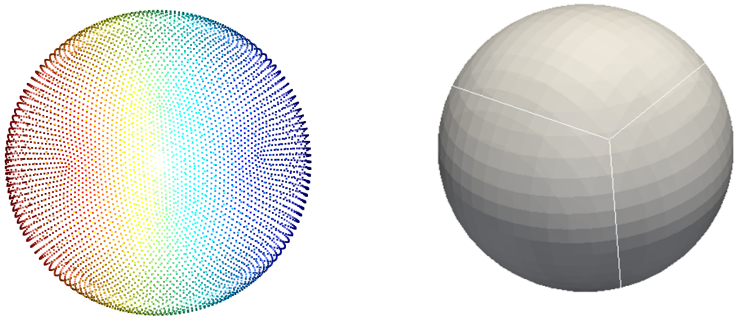
In the following, we present the numerical results obtained from testing the algorithm on two different point cloud datasets. The purpose of these tests was to evaluate the algorithms' performance in reconstructing meshes from point clouds of varying complexity.
The first dataset used for testing was a simple and regular sphere, serving as a straightforward test case to assess the algorithm's ability to handle basic geometries.
The second dataset presented a more challenging scenario, consisting of a bunny-shaped point cloud, which introduced irregularities and complexities. This test aimed to evaluate the algorithm's performance in more intricate environments.

1.2.1 `Pyvista` library

The `Pyvista` library provides a straightforward procedure for building a mesh from a point cloud. From the user's perspective, the Python code required is minimal:

```
points = Pyvista.PolyData(points)
mesh = cloud.reconstruct_surface()
```

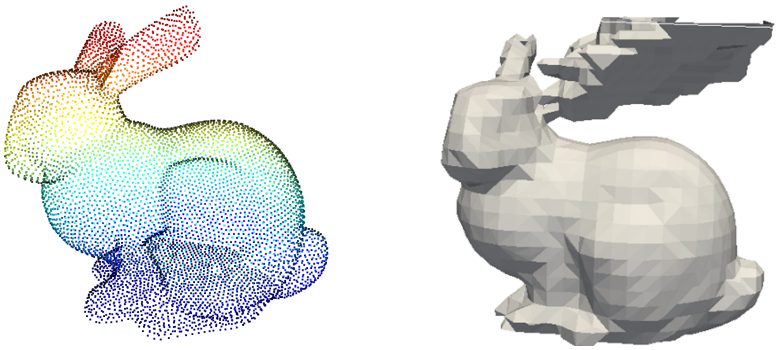
This code snippet converts the input point cloud (points) into a `PolyData` object using the `PolyData()` function. Then, the `reconstruct_surface()` function is called to perform the mesh reconstruction process.
In the case of a simple geometry like a sphere, the `Pyvista` algorithm performs well and accurately reconstructs the surface.



(a) Sphere point cloud. (b) Sphere mesh

Figure 3: `Pyvista`, First case: the sphere.

In contrast, when dealing with complex geometries such as a bunny, the performance of the Pyvista algorithm is noticeably inadequate. It fails to accurately reconstruct the surface, particularly in the challenging region around the bunny's ear. This specific area poses a significant obstacle for the algorithm, resulting in a considerable deviation from the expected reconstruction.



(a) Bunny point cloud. (b) Bunny mesh

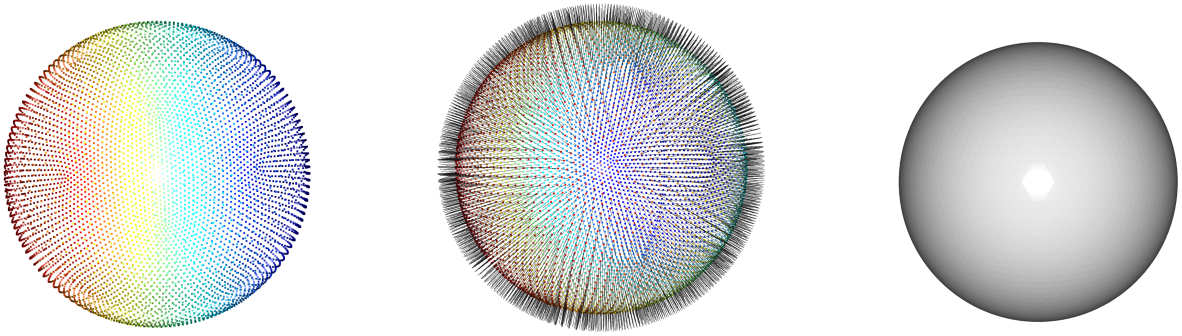
Figure 4: `Pyvista`, Second case: the bunny. Bad reconstruction in the ear region.

1.2.2 `Open3d` library: Ball pivoting algorithm 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.

The simplest point cloud, and thus the first on which the algorithms are tested, is the one of the sphere (Figure 5). In this case, all the `Open3d` algorithms perform well, both with the real normals and with the estimated ones; this behavior can be related to the very regular point cloud given in input.



(a) Sphere point cloud. (b) Sphere estimated normals. (c) Sphere mesh, accurate in all the cases.

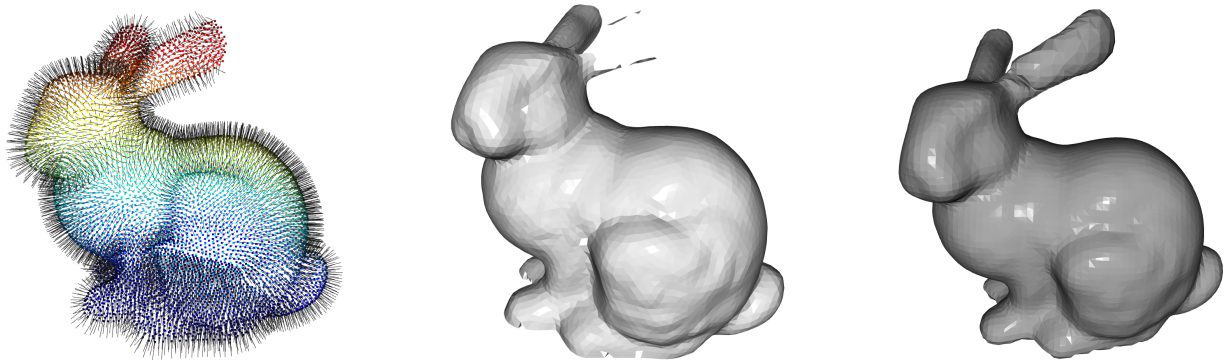
Figure 5: First case: the sphere.

The second point cloud on which the already presented algorithms are tested is the one of the bunny (Figure 6). In this case, the geometry is more complex, thus the various cases must be studied separately. At first, we start from the point cloud with given normals and we test both the BPA (with `radius` = 0.75 * `avg_dist`) and the Poisson algorithm (with `depth` = 8); both methods perform well, with the Poisson giving a more precise result with respect to the BPA, where some holes are present in sharp-cornered areas.

(a) Bunny point cloud. (b) Bunny mesh with BPA. (c) Bunny mesh with Poisson.

Figure 6: Second case: the bunny with given normals.

The next step is thus to apply the same methods to a point cloud where the normals are not given as input, but they are estimated with the functions `estimate_normals` and `orient_normals_consistent_tangent_plane`. As it can be seen from Figure 7a, these two functions do not estimate perfectly the normals for all the points in the cloud: specifically, on the left ear of the bunny some normals 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 7: Third case: the bunny with estimated normals.

1.3. Summary

Library (+ method)	Pros	Cons
Pyvista	No normals required	Bad reconstruction on complex geometry Complex method, code changes impactful
Open3D - Ball Pivoting	Better reconstruction than Pyvista	Estimations of the normals is required Some features are missed
Open3D - Poisson	Smooth reconstruction	Estimations of the normals is required

Given the summarized considerations presented in the table, we have decided to employ 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.

2. Open3d - Detailed description of the methods

2.1. Estimating Normals

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 $\Omega_K^{(i)}$ using the function `EstimatePerPointCovariance`.
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.

To summarize, the pseudo-algorithm is reported in 1.
In Figure 8 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 $\mathbf{v}$.

Algorithm 1 Estimating Normals

procedure EstimateNormals(PointCloud)
for i in PointCloud (in parallel) **do**
  $\Sigma_i \leftarrow$ EstimatePerPointCovariance(i, neighbours=30)
  $\text{eigenvectors}_i, \text{eigenvalues}_i \leftarrow$ Eigendecomposition(Σ_i)
  $\text{lowest\_eigenvalue\_index}_i \leftarrow$ FindLowestEigenvalueIndex(eigenvalues_i)
  $\text{normals}_i \leftarrow \text{eigenvectors}_i[\text{lowest\_eigenvalue\_index}_i]$
end for
end procedure=0

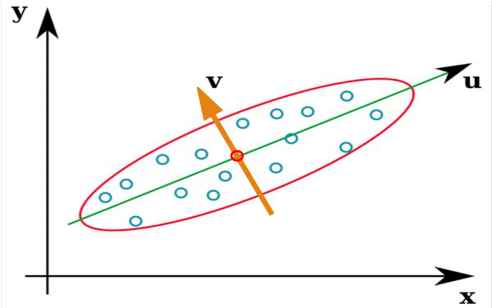


Figure 8: Schematic representation (in 2D) of the `Open3d` algorithm for the estimation of the normals.

2.2. Orienting Normals

Once the normals in the point cloud have been estimated, a method called `OrientNormalsConsistentTangentPlane` is invoked to consistently orient the normals to the point cloud. The function takes as input the integer K that expresses the number of neighbours that must be visited for each point. The algorithm performs 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.

The pseudo algorithm can be seen as follow:

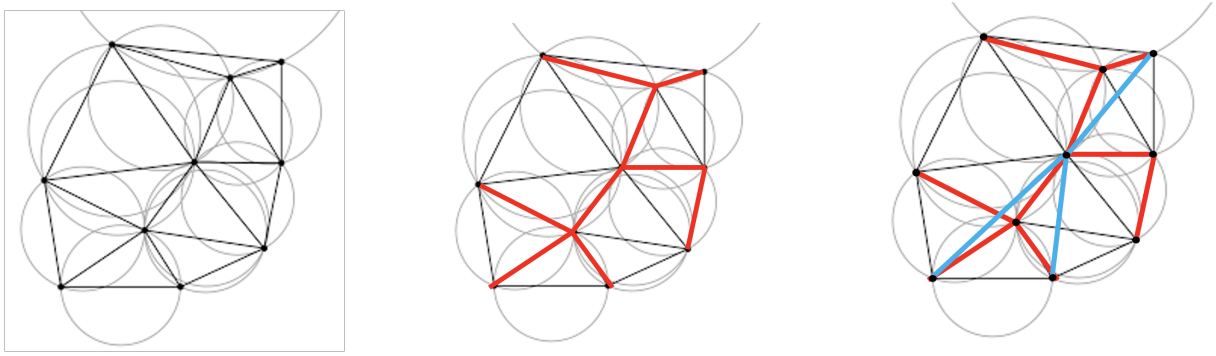
Algorithm 2 OrientNormalsConsistentTangentPlane(int K)

Input: Point cloud with estimated normals, K (number of neighbors)
Create a Delaunay graph where edge weights are the Euclidean distance between points.
Find the minimum spanning tree with Kruskal's algorithm.
for every point p in the point cloud **do**
    Find its K nearest neighbors (in terms of Euclidean distance).
    for every edge (p, p_k) **do**
        Add the edge to the resulting graph if it is not present in the initial Delaunay graph.
    end for
end for
for every pair of connected points i and j **do**
    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.
end for
Extract the final minimum spanning tree with Kruskal's algorithm.
Set an initial point x_0 as the one with the highest z coordinate.
Orient its normal n_0 such that $n_0 \cdot \hat{e}_z > 0$, where $\hat{e}_z = (0, 0, 1)$.
for every node in the minimum spanning tree (except the initial point) **do**
    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:

$$n_p \cdot n_i > 0$$

end for
Output: Point cloud with consistently oriented normals. =0

In Figure 9 the first part of the above-mentioned algorithm is applied to a simple case, as an example. The final graph is the one in the most right figure and it 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 9: From the point cloud to the graph used to orient the normals.

3. Our improvements for Open3d

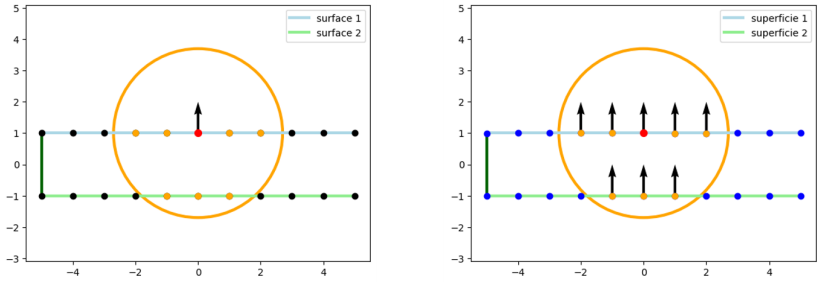
As showed by the results reported in the previous section, when the point cloud becomes more complex the algorithm for the orientation of the normals fails, and consequently the reconstructed mesh is inaccurate. After a deep analysis of the function `orient_normals_consistent_tangent_plane`, whose key points have been reported in Algorithm 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.

3.1. Mathematical formulation of the problem

3.1.1 Euclidean distance, not always the best choice

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 this paper.
Let's consider a simple 2D geometry consisting of two parallel surfaces, denoted as S_1 and S_2, as illustrated in Figure 10a. Taking the reference point 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 x_0. This misclassification leads to an inaccurate orientation of the normals associated with the lower surface, as depicted in Figure 10b. 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 x_0. Some points are not real neighbors.

(b) Misorientation of the normals.

Figure 10: 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 x_0 in such geometries. Assuming the normal direction is correct, we can utilize the tangent plane defined by x_0 and its normal n_0. We define the distance between any point x and x_0 as follows:

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

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 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 11.

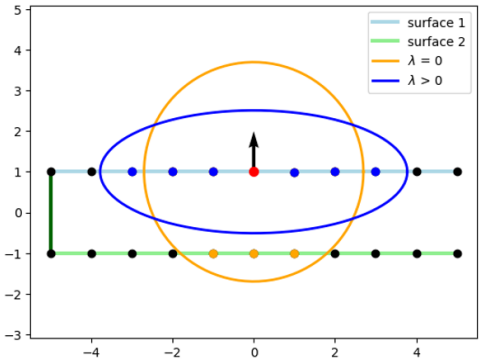


Figure 11: New concept of distance. Blue points are now the K nearest neighbours

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 x_0 be the reference point, n_0 its associated normal, and $x \in \Omega$ be any point in the point cloud. The set of potential neighbors of point x_0, is defined as:

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

where α_0 is the threshold angle[a]. As shown in Figure 12, 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.

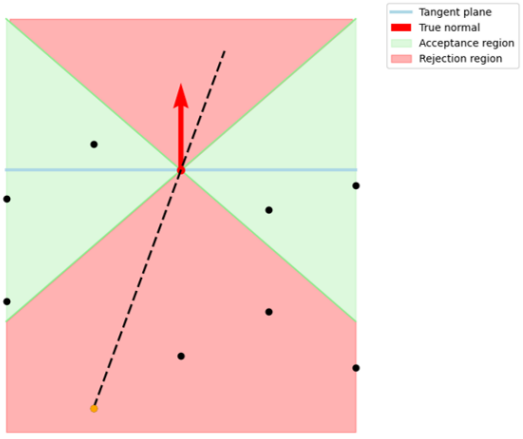


Figure 12: Only points within the acceptance region (green) are candidate neighbors.

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.

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

3.1.2 Changing 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 the mesh.
To reduce the likelihood of encountering this issue, we decided to initiate the orientation of the normals from the base of the point cloud. This choice was based on our heuristic observation that this particular region of the domain is more likely to exhibit a regular shape. 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).

3.2. Methods

In order to apply the results of our theoretical discussion, we had to make changes to the source code of the `Open3D` library. Specifically, we modified the `OrientNormalsConsistentTangentPlane` function, whose original algorithm is described in Algorithm 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. Consistent with Section 3.1.1, 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, 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.,

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

- 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.,

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

  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 3.1.2, 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 Algorithm 3. For more detailed information, the modified code can be found in our GitHub fork at: `https://github.com/eugeniovaretti/Open3D`.

Algorithm 3 OrientNormalsConsistentTangentPlane(int K, double λ, double $cos(\alpha_0)$) - Extended version

Input: Point cloud with estimated normals, K, λ, $cos(\alpha_0)$
Create a Delaunay graph where edge weights are the Penalized Euclidean distance between points.
Find the minimum spanning tree with Kruskal's algorithm.
for every point p in the point cloud **do**
    Find its K nearest neighbors (in terms of Euclidean distance).
    for every edge (p, p_k) **do**
        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) (threshold angle)
        - the point p_k is not an outlier in terms of distance from the plane wrt the whole subset of K neighbours
    end for
end for
for every pair of connected points i and j **do**
    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.
end for
Extract the final minimum spanning tree with Kruskal's algorithm.
Set an initial point x_0 as the one with the lowest z coordinate.
Orient its normal n_0 such that $n_0 \cdot \hat{e}_z > 0$, where $\hat{e}_z = (0, 0, -1)$.

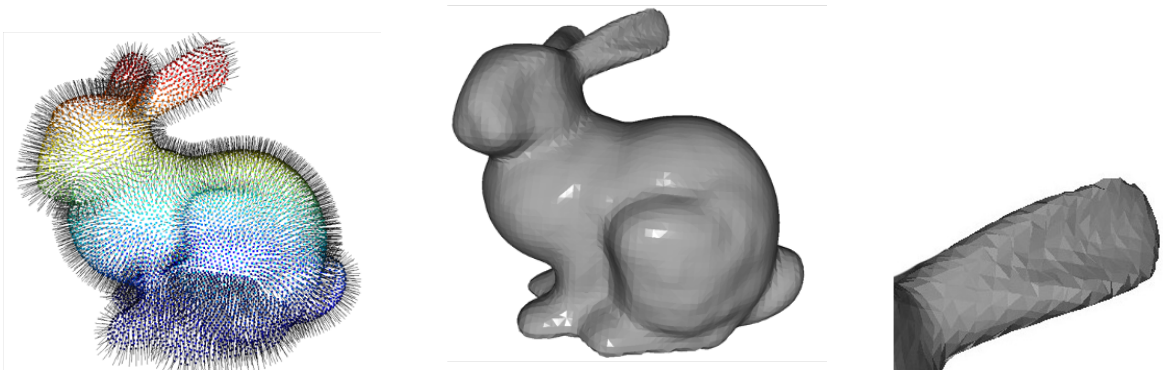
for every node in the minimum spanning tree (except the initial point) **do**
    Orient the normal of the current node by ensuring that the dot product between the parent node's normal n_p and the current node's normal n_i is greater than 0:

$$n_p \cdot n_i > 0$$

end for
Output: Point cloud with consistently oriented normals. =0

3.3. 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 13 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 $\alpha = 60° \rightarrow \cos\alpha = 0.5$).



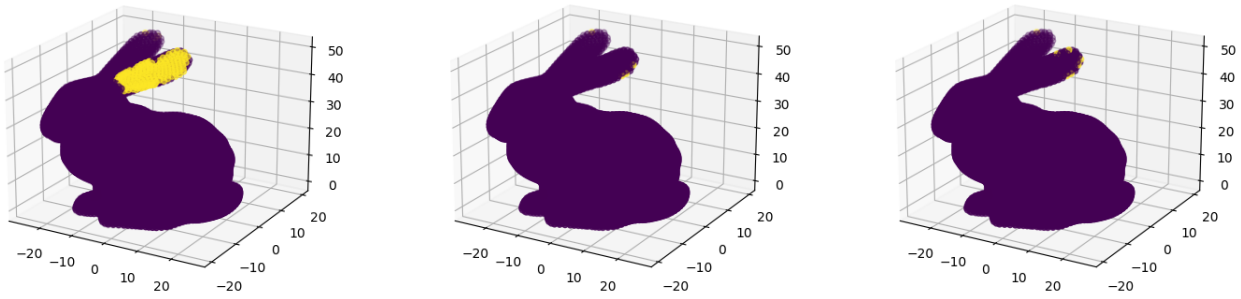
(a) Point cloud with estimated normals.

(b) Mesh given as output by Poisson algorithm.

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

Figure 13: Poisson reconstruction using the normals estimated with our improvements.

In order to have full awareness of the obtained results and to appreciate more our 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°.
In a qualitative framework, 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 14 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.



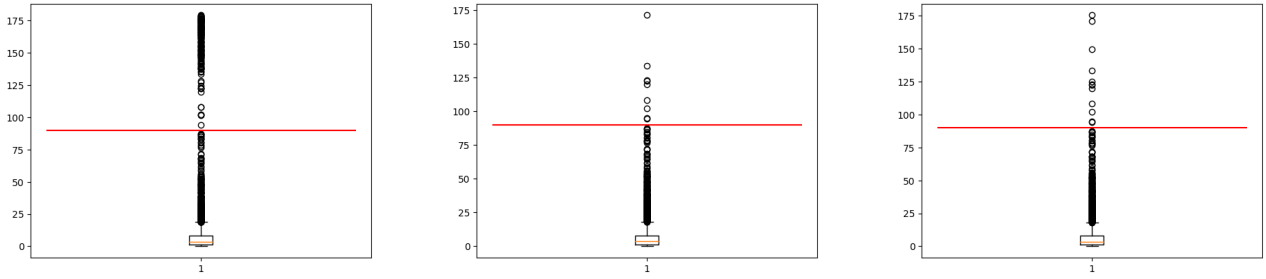
(a) Original algorithm O3D

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

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

Figure 14: 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 15). 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 15: Boxplots of the error angles in the three cases, with the threshold of 90°.

4. Conclusions

In conclusion, we are quite satisfied with the obtained results; indeed, we were able to understand how the functions implemented in the library worked and to identify where to operate in order to resolve the issues that we detected. As confirmed by the results reported in the previous section, our modifications were effective, but surely improvable.

A first weakness is 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 `knns` 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.

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 4.

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

Algorithm 4 Kruskal's Algorithm

```
Sort all the edges of the graph in non-decreasing order of their weights.
Create a separate set for each vertex of the graph. Initially, each vertex is its own set.
while MST has fewer than |V| − 1 edges do
    Consider the next edge, e, in the sorted order.
    if Adding edge e does not create a cycle then
        Add edge e to the Minimum Spanning Tree (MST).
        Merge the sets of the two vertices of the edge.
    end if
end while=0
```

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.

References

[1] 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.

[2] H. Hoppe. *Surface Reconstruction from Unorganized Points*. PhD thesis, University of Washington, 1994.

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

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