Face Segmentation with Topological Data Analysis

Andre Lim, Keaton Hawkins, Matías Pinto

Duke University, Fall 2024

1 Introduction and Background

To correctly interpret 3D point cloud data, it is often necessary to partition the data set into regions corresponding to objects or parts of an object. This process, known as segmentation, divides the data set into perceptually meaningful clusters. Within a 3D point cloud, a region is defined as a group of connected points with similar properties. Understanding regions is fundamental for interpreting point cloud data, as these regions often represent objects in a scene. Furthermore, segmentation of regions is a crucial preprocessing step towards pattern recognition and scene understanding.

For a topological space X, a discrete representation of X consists of simplices. A d-simplex σ is the convex hull of $d+1$ affinely independent vertices $v_0, \ldots, v_d \in \mathbb{R}^n$. For any vertex v_i, the d vectors $v_j - v_i$ (where $j \neq i$) are linearly independent. This implies that given a set of $d+1$ vertices, a simplex comprises points that are linear combinations of these vertices with non-negative coefficients summing to 1. A face τ of σ is the convex hull of any subset of the $d+1$ vertices. For instance, the faces of a triangle correspond to its three edges.

Simplicial complexes are formed by combining simplices. A simplicial complex $\mathcal{K}$ is a collection of simplices satisfying two conditions: if $\sigma \in \mathcal{K}$ and τ is a face of σ, then $\tau \in \mathcal{K}$; and if $\sigma, \sigma' \in \mathcal{K}$, then $\sigma \cap \sigma'$ is either empty or a face of both σ and σ'. Given a set of vertices $V = \{v_0, \ldots, v_d\}$, an abstract simplicial complex is a collection $\mathcal{K}$ of simplices closed under the operation of taking subsets. In other words, if $\sigma \subseteq V$ is a simplex ($\sigma \in \mathcal{K}$) and τ is a face of σ ($\tau \subseteq \sigma \subseteq V$), then $\tau \in \mathcal{K}$.

For a set of vertices $V \subseteq \mathbb{R}^n$ and a fixed radius ϵ, the Vietoris-Rips complex of the space X is an abstract simplicial complex whose d-simplices correspond to unordered $(d+1)$-tuples of vertices in X that are pairwise within ϵ distance from each other.

A simplicial complex is constructed with respect to a scale parameter ϵ. However, when computing homology, the topological features of this simplicial complex may arise due to noise or an incorrectly chosen scale parameter. Persistent homology addresses this issue by analyzing the growth of complexes over the entire range of scale parameter values. Concretely:

$$\emptyset \subseteq \mathcal{K}_0 \subseteq \mathcal{K}_1 \subseteq \cdots \mathcal{K}_L = \mathcal{K},$$

where L represents the maximum value for constructing the complex. This approach provides control over how long a feature must persist in the filtration before being considered significant, thus excluding short-lived, noise-induced features.

In this project, we use the concept of persistent homology to explore the segmentation of a human head to identify and separate facial features. The ultimate goal is to identify and isolate meaningful structures—like a nose, eye or mouth—that can be further analyzed or utilized.

2 Dataset

The `.obj` file format is a widely used plain text format for representing 3D geometry in graphics applications. Its simplicity and ease of parsing have made it a standard in several 3D modeling and rendering tools. The format supports the representation of key geometric elements such as vertices, vertex normals, texture coordinates, and faces. Below we break down the key components of the `.obj` file format:

a. Vertex Data

Vertices represent points in 3D space that define the shape of the object. Each vertex is specified on a line starting with the letter `v`, followed by the coordinates x, y, and z. Optionally, a w coordinate can be included, with the default value being $w = 1.0$ for 3D points.

```
v x y z [w]
```

Example:

```
v 1.0 2.0 3.0
v -1.0 0.5 2.5
```

b. Texture Coordinates

Texture coordinates are used to map 2D textures onto the 3D model. These are specified with lines starting with `vt`, followed by the u and v coordinates, and optionally a w coordinate.

```
vt u v [w]
```

Example:

```
vt 0.5 0.75
vt 0.1 0.2
```

c. Vertex Normals

Vertex normals are used to define the direction of a vertex for lighting and shading calculations. They are specified with lines starting with `vn`, followed by the x, y, and z components of the normal vector.

```
vn x y z
```

Example:

```
vn 0.0 1.0 0.0
vn 1.0 0.0 0.0
```

d. Faces

Faces define the polygons of the model, typically triangles or quadrilaterals. They also map vertex normals to their corresponding vertices. Each face is specified on a line starting with `f`, followed by a list of vertex indices. Each index may optionally include texture coordinate indices and normal indices, formatted as follows:

```
f v1[/vt1[/vn1]] v2[/vt2[/vn2]] v3[/vt3[/vn3]]
```

Example:

```
f 1/1/1 2/2/2 3/3/3
f 4//1 5//2 6//3
```

e. Comments

Comments can be included in the file by starting a line with the `#` character. These lines are ignored by parsers.

Example:

```
# This is a comment
```

Example .obj File

```
# Example OBJ file
v 1.0 0.0 0.0
v 0.0 1.0 0.0
v 0.0 0.0 1.0
vn 0.0 0.0 1.0
f 1//1 2//1 3//1
```

In this project, we test our algorithm on the CoMA Dataset provided by the Max Planck Institute for Intelligent Systems. This dataset contains scanned data of 12 different subjects' faces in .obj format, each performing 12 extreme, asymmetric facial expressions. [1]

[1] *Generating 3D Faces using Convolutional Mesh Autoencoders*

3 Methodology and Algorithm

In this project, we develop an algorithm to perform region growing similar to a Vietoris Rips filtration on a 3D point cloud based on geometric proximity and normal similarity. The process begins by extracting vertices, normals, and faces from an `.obj` file. The algorithm then computes connected components based on a proximity threshold (ϵ) and a normal angle threshold ($\theta_{\max}$)).

a. Input Data

The algorithm requires the following input data:

- **Vertices:** A list of 3D coordinates representing points in the object.
- **Normals:** A list of normal vectors, each corresponding to a vertex, used to measure similarity in orientation.
- **Faces:** A list of triangular face indices, used to compute the edges between vertices.
- **Thresholds:**
  - $\theta_{\max}$: Angle threshold for similarity in normal vectors.
  - $\epsilon_{\min}, \epsilon_{\max}$: The minimum and maximum proximity thresholds for growing regions.

b. Algorithm

The algorithm proceeds in several steps:

1: KDTree Construction

A KDtree is built from the vertex positions to facilitate efficient neighbor searches within a given radius.

2: Region Growing

The region-growing process is performed iteratively over a range of ϵ values:

1. For each vertex i, query the KDTree to find neighbors within the current ϵ radius.
2. For each neighbor j, compute the angle θ between the normal vectors of i and j. The angle is computed as:

$$\theta = \arccos\left(\frac{\mathbf{n}_i \cdot \mathbf{n}_j}{\|\mathbf{n}_i\|\|\mathbf{n}_j\|}\right)$$

   where $\mathbf{n}_i$ and $\mathbf{n}_j$ are the associated normal vectors of vertices i and j, respectively.

3. If $\theta \leq \theta_{\max}$, the neighbor is marked as similar and recorded.

3: Union-Find for Region Merging

A union-find data structure is used to manage connected components:

1. The neighbor with the smallest index is selected as the representative.
2. If the neighbors belong to different components, merge them using the union operation, and set the death time of the merged component to the current ϵ value.

c. Output Data

- **Labels:** An array where each entry represents the region label of a vertex.
- **Death Times:** An array where each entry represents the ϵ value at which a vertex stopped contributing to its region.

d. Thresholds and Parameters

- $\theta_{\max}$: Defines the angular similarity for normal vectors.
- $\epsilon_{\min}$ and $\epsilon_{\max}$: Define the proximity range for growing regions.
- $\Delta\epsilon$: Step size for incrementing ϵ during the iteration.

e. Complexity Analysis

The algorithm's complexity depends on:

- KDTree queries: $O(N \log N)$ for construction and $O(N \log N)$ for each radius query, where N is the number of vertices.

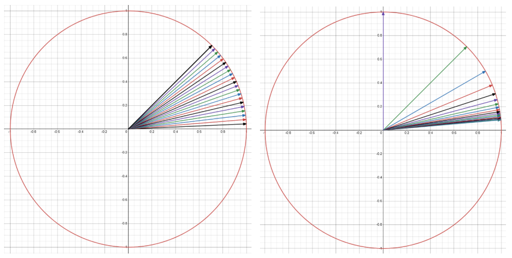
- Union-Find operations: Nearly $O(1)$ for each union or find operation, due to path compression.

- Total: Approximately $O(N^2 \log N)$ for dense point clouds.

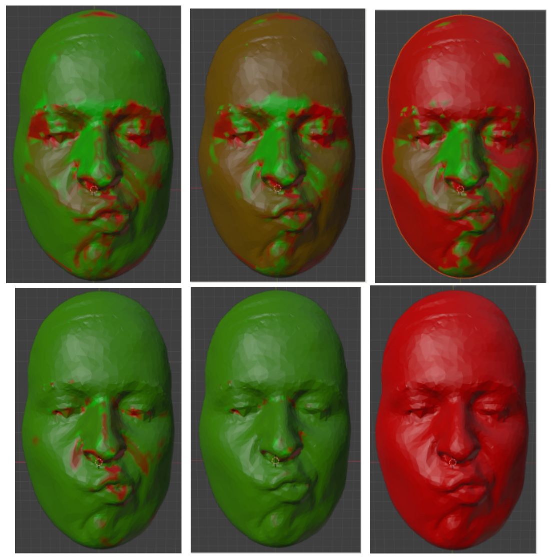
4 Results

We tested our algorithm on three different versions of the 3D face data, starting with a densely sampled point cloud. Due to the number of pairwise computations needed, it was computationally inefficient to run our algorithm on this high resolution face. We then attempted to decimate the mesh granularity, obtaining a very course triangulation with many fewer points. However, this reduction was significant enough to interfere the angles between normal vectors, since new faces had to be created in order to fill in gaps where previous points had existed. Thus, the algorithm produced inaccurate results. We settled on analyzing a reduced resolution point cloud with the sampling sparsity in between that of the high resolution face and the decimated version. This was both computationally feasible and accurate.

We also tested our algorithm with twenty one different evenly spaced angles ranging from $\frac{\pi}{72}$ radians to $\frac{\pi}{4}$ radians, and also with 18 angles ranging from $\frac{\pi}{36}$ to $\frac{\pi}{2}$, where each angle was obtained from the previous by subtracting 2 from the denominator (i.e., $\theta_{max} = \frac{\pi}{36}, \frac{\pi}{34}, \frac{\pi}{32}, \frac{\pi}{30}, ...$). To the right is a visualization of the angles tested:

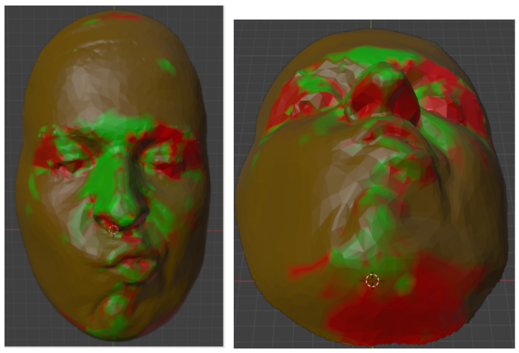


One would expect that for large values of $\theta_{\max}$, the face will consist of one large segment after the algorithm has completed. This is because almost all of the pairwise normal vectors will be at angles less than $\theta_{\max}$ for large $\theta_{\max}$, and so many region mergings will happen. Further, one would expect that as $\theta_{\max}$ decreases, the number of different segments will increase. Eventually, for very small $\theta_{\max}$, it becomes difficult to distinguish noise in angle changes from angle changes that result from boundaries around different segments of the face. Thus, for sufficiently small $\theta_{\max}$, we expect the segmentation to become unpredictable. Below are the segmentation results for various selected values of $\theta_{\max}$:



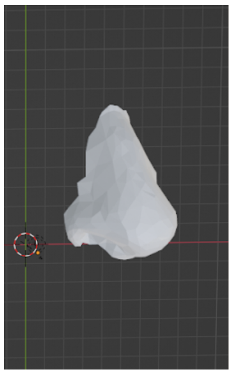
Starting from the top and moving left to right in each row, these images represent the segmentation obtained by running our algorithm for $\theta_{max} = \frac{\pi}{36}, \frac{\pi}{34}, \frac{\pi}{32}, \frac{\pi}{26}, \frac{\pi}{18}$, and $\frac{\pi}{4}$ respectively. As expected, the $\frac{\pi}{4}$ segmentation simply merges the entire face into a single component, indicated by the uniform redness. Then, as the angle decreases, new segments (red for $\frac{\pi}{18}$ and $\frac{\pi}{26}$) slowly begin to grow. In the $\frac{\pi}{32}$ case, a clear region has been identified in green surrounding the nose, and likewise, a clear region in red has identified the rest of the face. In the $\frac{\pi}{34}$ case, more segmentation is present. Finally, the accuracy of the segmentation is reduced in the $\frac{\pi}{36}$ case, which contains more noise in its segmentation coloring. These results correspond precisely with our mathematical inference.

Upon visual inspection, we selected $\theta_{\max} = \frac{\pi}{34}$ as the angle which most effectively segmented the face. The results of this segmentation are shown below:

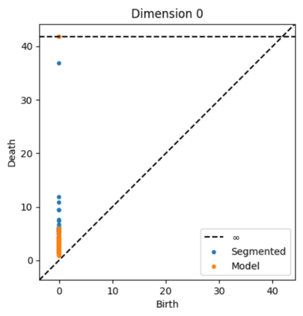


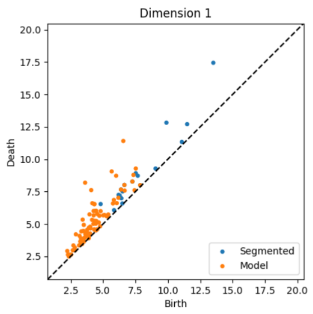
As shown above, our algorithm was able to segment out the nose and its surrounding region (bright green), including some of the lower forehead. It also created one segment around each of the eye sockets (red) and one large segment that connects the smoothly varying surface of the rest of the face (brown). In the figure on the right, we see that separate components were also identified for each nostril (red) and the section of the face underneath the chin (red). The peak of the chin, the top of the head, and the lips were also weakly identified. Therefore, our algorithm was successful at identifying the various segments of the face, as separated by varying normal vector angles. However, the results are imperfect. For example, the segment containing the nose also extends to erroneously include several other portions of the face, including parts of the cheeks, lips, and the bottom section of the forehead. This implies that our algorithm could be refined or enhanced.

In order to quantitatively evaluate the performance of our algorithm, we analyze the persistent homology of the segments it generated. To begin, we manually segmented the nose out of the face, shown below:



We then used the Gudhi python library to compute the persistence diagram generated by the formation of a Vietoris Rips Complex on this point cloud. We also computed the persistence diagram for the corresponding region that was segmented out by our algorithm. These persistence diagrams are shown below:

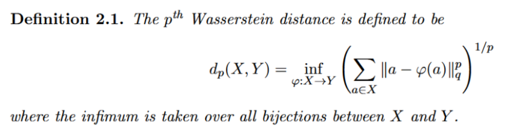




The top diagram shows the 0 dimensional persistent homology of both point clouds, with the features of the actual nose (model) displayed in orange, and the features of the algorithm's output (segmented) in blue. Since every vertex begins as its own component, everything is born at time 0 in this diagram. As ϵ increases, features die at different times distributed along the vertical axis. The (orange) features of the model are more concentrated around lower death times than the (blue) segmented features. In particular, there is one very persistent feature of the segmented data, which does not die until close to 40. Finally, both point clouds have one component that lives forever, as expected. This distribution aligns with intuition. The actual nose model is compact, whereas the segmented output of our algorithm is spread out and includes many erroneous segments. Therefore, it is not surprising that there might be two different components in the segmented output which do not join until relatively late.

The bottom diagram shows the 1 dimensional persistent homology of both clouds, with the coloring the same. Most of the 1 dimensional features are close to the diagonal, indicating topological noise, although several points drift farther away from the diagonal, indicating more persistent 1 cycles. In particular, several (blue) points from the output of the segmentation algorithm are born much later than the latest birth time of any (orange) model point, and are also somewhat persistent. This indicates that the segmentation creates new significant topological features that are not present in the true nose model, highlighting the imperfection of our segmentation algorithm.

Using Gudhi, we also computed the 1-Wasserstein distance between the Persistence diagrams of the nose model and the segmentation output. The Wasserstein distance between the 0 dimensional persistence diagrams was 363.08, while the Wasserstein distance between the 1 dimensional diagrams was 49.75. In both cases, the distance was relatively large, indicating that there is significant room for improvements in our algorithm in order to produce segments whose topology more closely aligns with the correct segments of the face. Recall that the Wasserstein distance between two persistence diagrams is, informally, the minimally distant mapping between the points of the two diagrams, formally defined as follows:



Definition 2.1. *The p^{th} Wasserstein distance is defined to be*

$$d_p(X,Y) = \inf_{\varphi : X \to Y} \left(\sum_{a \in X} \|a - \varphi(a)\|_q^p \right)^{1/p}$$

where the infimum is taken over all bijections between X and Y.

5 Conclusions, Limitations, and Future Work

Our algorithm was ultimately successful at identifying different features of the face, including the nose, eyes, chin, and the rest of the face. However, as described above, there is room for improvement. First of all, a more quantitative method could have been chosen for optimizing the choice of the mesh density and the optimal value of $\theta_{\max}$. For example, we could have run the algorithm with many different values of $\theta_{\max}$, or on many different meshes, and then selected the parameter values that minimize the Wasserstein distance between the resulting persistence diagrams and the persistence diagrams of the correct segmentations. However, the algorithm is not especially efficient due to the many pairwise comparisons, so it could take a long time to compute the segments for many different parameter values. Therefore, future work could include efficiency optimizations that reduce the number of comparisons. The object data chosen also impacts the effectiveness of the algorithm. For example, a human face contains many smooth curves, meaning that, especially for densely sampled point clouds, the pairwise normal angle differences may not be significant. By contrast, some objects, such as a table, have sharp edges with orthogonal faces, so our algorithm would likely run more effectively on such data. Finally, we intend to implement a version of the algorithm which, rather than using angles between normal vectors, uses differences in discrete mean and Gaussian curvature via the discrete Laplacian operator and 2nd fundamental form for each mesh triangular face.

References

[1] S. Rusinkiewicz, "Estimating curvatures and their derivatives on triangle meshes," in *Proceedings. 2nd International Symposium on 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004.* IEEE, 2004, pp. 486–493.

[2] A. Ranjan, T. Bolkart, S. Sanyal, and M. J. Black, "Generating 3d faces using convolutional mesh autoencoders," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 704–720.

[3] C. Tralie, N. Saul, and R. Bar-On, "Ripser. py: A lean persistent homology library for python," *Journal of Open Source Software*, vol. 3, no. 29, p. 925, 2018.

[4] A. Ranjan, T. Bolkart, S. Sanyal, and M. J. Black, "Generating 3D faces using convolutional mesh autoencoders," in *European Conference on Computer Vision (ECCV)*, 2018, pp. 725–741.

6 Appendix

We provide the code for our main algorithm as follows. The remainder of our code, including our entire data processing pipeline and our `.ply` segmented output files, is present in Gitlab at `https://gitlab.oit.duke.edu/khh36/tda-final-project`. Some `.ply` output files are present in the "test-branch" branch:

main.py

```python
import numpy as np
import matplotlib.pyplot as plt
from quickunion import WeightedQuickUnionWithPathCompressionUF
from scipy.spatial import KDTree

def compute_normal_angle(normal1, normal2):
    dot_product = np.dot(normal1, normal2) / (np.linalg.norm(normal1) * np.linalg.norm(normal2))
    dot_product = np.clip(dot_product, -1.0, 1.0)
    angle = np.arccos(dot_product)
    return np.degrees(angle)


def compute_curvature_criterion(vertex1, vertex2, eps):
    # TODO: Implement the curvature computation between vertex1 and vertex2
    # Placeholder: Assuming all curvature is acceptable for now
    return True


def inRange(val, min, max):
    if min <= val <= max:
        return True


def growRegions(points, normals, edges, theta_max, eps_min, eps_max):
    tree = KDTree(points)
    unionArray = WeightedQuickUnionWithPathCompressionUF(len(points))
    death_times = np.full(len(points), np.inf)

    edge_set = set(tuple(sorted(edge)) for edge in edges)

    eps = eps_min
    while eps <= eps_max:
        for i, point in enumerate(points):
            neighbors = tree.query_ball_point(point, r=eps)
            similar_nn = []

            for nn in neighbors:
                if nn == i:
                    continue

                # Check angle
                theta = compute_normal_angle(normals[i], normals[nn])
                if not (theta <= theta_max):
                    continue

                # Check connectivity
                if tuple(sorted([i, nn])) not in edge_set:
                    continue

                similar_nn.append(nn)

            if similar_nn:
                min_nn = min(similar_nn)

            for nn in similar_nn:
                if unionArray.find(nn) != unionArray.find(min_nn):
                    death_times[nn] = eps # Set death time for the point, record for persistence
                    unionArray.union(nn, min_nn) # Merge regions

        eps += 0.1

    return unionArray, death_times

#-----------------------------------------------------------------------------
# DATA READING

if __name__ == "__main__":

    vertex_file = 'vertices2.txt'
    normal_file = 'normals2.txt'
    face_file = 'faces2.txt'

    # Parse vertices
    vertices = []
    with open(vertex_file, 'r') as v_file:
        for line in v_file:
            parts = line.strip().split()
            if parts[0] == 'v':
                x, y, z = map(float, parts[1:])
                vertices.append([x, y, z])
    vertices = np.array(vertices)
    unionArray = (len(vertices))

    # Parse vertex normals, assume each vertex has its own normal
    normals = []
    with open(normal_file, 'r') as vn_file:
        for line in vn_file:
            parts = line.strip().split()
            if parts[0] == 'vn':
                nx, ny, nz = map(float, parts[1:])
                normals.append([nx, ny, nz])
    normals = np.array(normals)

    # Parse faces
    faces = []
    with open(face_file, 'r') as f_file:
        for line in f_file:
            parts = line.strip().split()
            if parts[0] == 'f':
                face = []
                for v in parts[1:]:
                    vertex_index = int(v.split('/')[0]) - 1 # OBJ indices are 1-based
                    face.append(vertex_index)
                faces.append(face)
    faces = np.array(faces)

    edges_set = set()
    for face in faces:
        v0, v1, v2 = face
        edges_set.update({
            tuple(sorted([v0, v1])),
            tuple(sorted([v1, v2])),
            tuple(sorted([v2, v0])),
        })
    edges = list(edges_set)

    # Thresholds and parameters
    theta_max = np.pi / 18
    eps_min = 0.001
    eps_max = 100

    # Grow regions and compute death times
    unionArray, death_times = growRegions(vertices, normals, edges, theta_max, eps_min, eps_max)

    labels = np.array([unionArray.find(i) for i in range(len(vertices))])
    print("Labels:", labels)
    print("Death Times:", death_times)
```

dataprep.py

```
if __name__ == "__main__":
    # Path to .obj file
    input_file = 'testobj.obj'

    v_file = 'vertices2.txt'
    vn_file = 'normals2.txt'
    f_file = 'faces2.txt'

    # Open the .obj file and the output files
    with open(input_file, 'r') as obj_file, \
        open(v_file, 'w') as v_out, \
        open(vn_file, 'w') as vn_out, \
        open(f_file, 'w') as f_out:

        for line in obj_file:
            if line.startswith('v '):
                v_out.write(line)
            elif line.startswith('vn '):
                vn_out.write(line)
            elif line.startswith('f '):
                f_out.write(line)

    print(f"Vertices saved in: {v_file}")
    print(f"Normals saved in: {vn_file}")
    print(f"Faces saved in: {f_file}")
```