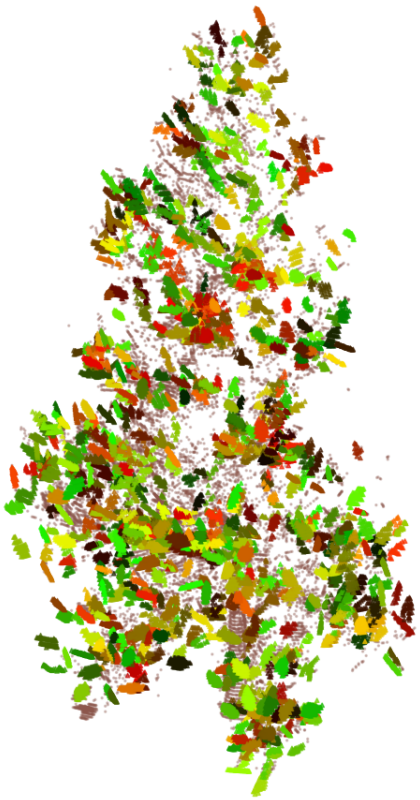
Leaf Counting and Area Estimation using 3D Point Cloud Data

December 9, 2025

TAU02E - Tomi Maijala (LUT), **Vinh Van** (TAU)



1 Introduction

The quantification of foliar elements within vegetation canopies—specifically the counting of leaves and the estimation of leaf area—stands as a central problem in contemporary remote sensing, forestry, and precision agriculture. Leaves function as the primary biological interface for gas exchange, driving the essential processes of photosynthesis, transpiration, and carbon sequestration. Consequently, the ability to derive precise, spatially explicit metrics of leaf distribution is not

merely an academic exercise but a fundamental requirement for robust ecological modeling, crop yield forecasting, and the monitoring of forest health in the face of climate change.

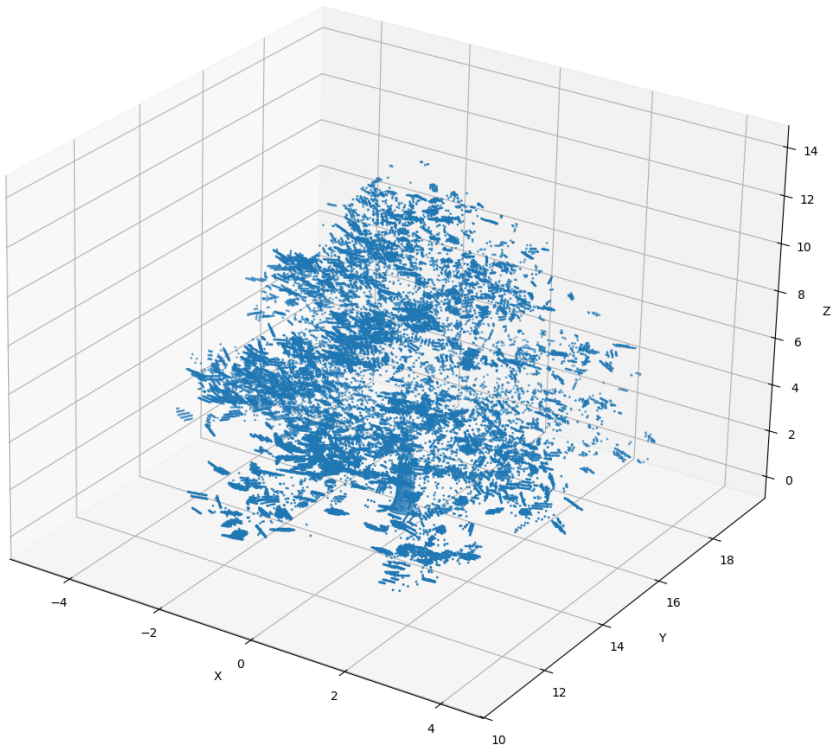
Historically, the assessment of canopy structure relied heavily on direct, destructive sampling or passive optical methods. Destructive techniques, while accurate, are labor-intensive, site-specific, and impossible to scale; optical methods such as hemispherical photography or localized light sensors provide valuable estimates of Leaf Area Index (LAI) but are inherently limited by their two-dimensional nature and saturation issues in dense canopies. [1] The emergence of Light Detection and Ranging (LiDAR) technology has fundamentally altered this landscape. As an active remote sensing modality capable of penetrating canopy gaps and recording multiple returns, LiDAR offers a unique capacity to digitize the three-dimensional (3D) architecture of vegetation with millimeter-level precision.

However, the transition from raw LiDAR point clouds to meaningful biological integers—such as a "leaf count"—is fraught with complexity. It requires navigating the "semantic gap" between geometric coordinates and biological organs. This challenge necessitates a sophisticated interplay of data acquisition strategies, noise filtration, semantic segmentation (differentiating wood from leaf), and instance segmentation (distinguishing individual leaves).

1.1 Dataset

This report demonstrates a workflow for leaf counting and area estimation using 3D point cloud data acquired via terrestrial LiDAR scanning. For the Basic Course on Mathematical Modelling final project, we were given LiDAR data of a graphically simulated tree. The point cloud has been measured by LiDAR from [0 0 1.5]. The LiDAR is able to measure from the same direction only the object closest to it. The measured point can be a leaf, tree branch, tree trunk, or ground surface. There are no obstacles to the line connecting the measured point and the laser, the point [0 0 1.5]. The data is obtained from HELIOS++.

This dataset only contains the 3D coordinates (X, Y, Z) of the points in the point cloud. So, there is no ground truth information for semantic segmentation (differentiating wood from leaf), or instance segmentation (distinguishing individual leaves), or total leaf area or leaf count. Therefore, we will rely mainly on qualitative analysis and comparison with simulated data from HELIOS++ to assess the reliability and accuracy of our methods.



Here you can see that the point cloud is denser on the side facing the LiDAR scanner, while the back side (approximately $Y > 15$) is much sparser due to occlusion.

1.2 Challenges

Several challenges arise when attempting to accurately count leaves and estimate leaf area from 3D point cloud data:

1. **Data Quality**: The distances between points in the point cloud can vary significantly, leading to uneven point density. This is due to factors such as the diffrent sampling density between horizontal and vertical directions, distance from the LiDAR to the object (the further the object is, the sparser the points are). In general, the sparser the points are, the harder it is to accurately identify and segment individual leaves.
2. **Occlusion**: Leaves can be occluded by other leaves or branches, making it difficult to accurately count them. This is especially true in dense canopies where leaves overlap significantly.

Additionally, **this dataset is obtained from a single scan position, which can lead to significant occlusion issues**.

3. **Leaf Size and Shape Variability**: Leaves can vary greatly in size and shape, making it challenging to develop a one-size-fits-all algorithm for leaf detection and area estimation.
4. **Leaf Overlap**: In dense canopies, leaves often overlap each other, leading to hard-to-seperate clusters of points that may represent multiple leaves. This can lead to undercounting of leaves and inaccurate leaf area estimates.

1.3 Assumptions

To address these challenges, we make several assumptions in our analysis. These assumptions also based on what we observe visually from the point cloud data:

1. **Point Density**: The point cloud density is adequate to capture the essential structure of the leaves. Its resolution is finer than the most leaves' size.
2. **Leaf Shape**: Leaves are mostly planar and can be approximated as flat surfaces for area estimation.
3. **Environmental Conditions**: The data was obtained in still air conditions, minimizing motion blur or distortion in the point cloud.

1.4 Approach Summary

Our method goes through several steps:

1. Keep only the tree half which faces the LiDAR scanner. (to mitigate occlusion issues)
2. Cluster the tree into smaller clusters.
3. Use different clustering proposing methods **(DBSCAN, RANSAC plane fitting)** to find the best clustering which gives the most leaf-like clusters for each smaller cluster. The leaf-like clusters are identified by **shape analysis**.
4. Compute leaf area for each identified leaf cluster and sum them up to get the total leaf area. The leaves areas are aproximated by **half the surface area of the convex hull of the leaf clusters**, because we assume the leaves are flat surfaces.
5. Only keep the leaves which have area below 0.15 m^2 to avoid suspiciously large clusters.
6. Visualize the final result and **show CT scans along different axes, for qualitative evaluation.** Clusters which look like leaves should be detected as leaves, and different leaves should be separated in the CT scans.
7. Using **simulated data from HELIOS++** for estimate the effect of occlusion.

2 Model and Solutions

2.1 Leaf cluster identification

The leaves in this dataset can be modeled by a set of points, sampled from a 2D plane in 3D space. The size of the leaves can vary, but they are generally small compared to the size of the tree. The

clusters of points that represent leaves should therefore have the following characteristics:

- **Planarity**: The points in a leaf cluster should lie approximately on a plane. Ofcause, a line will also be able to lie on a plane, so the cluster should forms a flat surface, not a line. There can be some error tolerance due to the leaf curvature but overall, the points should be close to a flat surface.
- **Separable**: The cluster should be have certain density inside, and separated from other clusters by sparse regions. There is different in density on horizontal and vertical directions due to how the LiDAR scaner works, but on each leaf cluster, the horizontal density is uniform and the vertical density is uniform.
- **Shape and Size**: The area and dimensions of the cluster should be within reasonable limits for a leaf. Clusters that are too large or too small can be discarded.

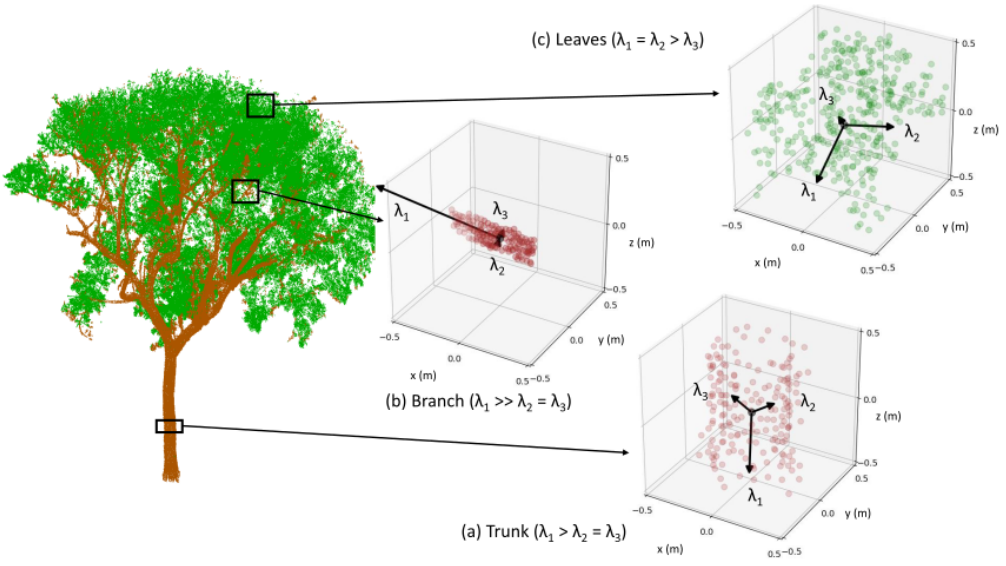
And that is how we identify leaf clusters in our approach. **Branches** are usually have sticky shapes, so they should fail planarity test. **Trunks** are usually big, unsperable surfaces, although a very large trunk piece can be planar, it should be unseparable large surface.

We use these algorithms to identify leaf clusters:

```
from sklearn.linear_model import RANSACRegressor, LinearRegression
import numpy as np

def analyze_cluster_shape(points):
    """Analyze the shape of a cluster of points using PCA."""
    if points.shape[0] < 3:
        return 0.0, 0.0, 0.0  # not enough points to analyze shape

    # get linearality, planarity, sphericity
    pca = PCA(n_components=3)
    pca.fit(points)
    eigenvalues = pca.explained_variance_
    total_variance = np.sum(eigenvalues)
    linearity = (eigenvalues[0] - eigenvalues[1]) / total_variance
    planarity = (eigenvalues[1] - eigenvalues[2]) / total_variance
    sphericity = eigenvalues[2] / total_variance
    return linearity, planarity, sphericity

def points_lie_in_a_plane_criteria(points, threshold=0.02):
    """
        Check if the points lie approximately in a plane using linear␣
 ↪regression.
    """
    # fit a 3D line to the points using linear regression
    lm = LinearRegression()
    lm.fit(points[:, :2], points[:, 2])
    # all point should be within a certain distance to the line, threshold = 0.
 ↪02
    predicted_z = lm.predict(points[:, :2])
```

```
    distances = np.abs(points[:, 2] - predicted_z)
    if np.all(distances < threshold):
        return True
    return False

def cluster_is_leaf_like(cluster_points, min_points=10, max_points=80,
                         min_ratio=20, plane_threshold=0.02):
    """
        Determine if a cluster of points is leaf-like based on size and shape␣
 ↪criteria.
    """
    linearity, planarity, sphericity = analyze_cluster_shape(cluster_points)
    ratio = planarity / (sphericity + 1e-7)
    is_on_plane = points_lie_in_a_plane_criteria(cluster_points,␣
 ↪threshold=plane_threshold)
    if cluster_points.shape[0] >= min_points and\
        cluster_points.shape[0] <= max_points and\
        ratio >= min_ratio and is_on_plane:
        return True, linearity, planarity, sphericity
    return False, linearity, planarity, sphericity
```

`analyze_cluster_shape` function: compute the shape characteristics of a cluster using PCA. This figure [Moorthy et al, 2020] can explain the meaning of linearity (λ_1), planarity (λ_2), and sphericity (λ_3).



`points_lie_in_a_plane_criteria` function: even though PCA can give us planarity value, it does not guarantee that the points really lie on a plane. Many shapes can have high planarity value but the points do not lie on a plane, for example, a half-cylinder shape, with large diameter. So we use this function to check if the points really lie on a plane, with some error tolerance.

Finally, we combine these two functions in `cluster_is_leaf_like` function to identify leaf clusters. Given a set of points, the function returns True if the cluster is leaf-like, otherwise False. The criteria for a leaf-like cluster are:

1. The number of points in the cluster is between `min_points` and `max_points`. (based on observed leaf sizes in the point cloud)
2. The ratio of planarity to sphericity is greater than `min_ratio`. (to ensure the cluster forms a flat surface, not a line or a blob)
3. The points lie approximately on a plane, as determined by the `points_lie_in_a_plane_criteria` function.

2.2 Objective Function

Now that we have a method to identify leaf clusters, we can define our objective function which by optimizing it, we can find the best clustering of our point cloud, which also includes semantic segmentation (differentiating wood from leaf) and instance segmentation (distinguishing individual leaves).

```
def score_function(points, labels, min_points=10, max_points=80,
                   min_ratio=20, plane_threshold=0.02):
    """
        Receive some clustered points and their labels
        and return a score based on quantity and quality of leaf-like clusters␣
 ↪found
    """
    leaf_cluster_id_2_stats = {}
    for cluster_label in set(labels):
        if cluster_label == -1:
            continue  # skip noise
        cluster_points = points[labels == cluster_label]

        is_leaf,\
        linearity,\
        planarity,\
        sphericity = cluster_is_leaf_like(cluster_points,
                                          min_points=min_points,
                                          max_points=max_points,
                                          min_ratio=min_ratio,
                                  plane_threshold=plane_threshold)
        if is_leaf:
            leaf_cluster_id_2_stats[cluster_label] = (linearity, planarity,␣
 ↪sphericity)

    score = 0
    for cluster_label, stats in leaf_cluster_id_2_stats.items():
        planarity = stats[1]
        num_points = np.sum(labels == cluster_label)
```

```
        # score += num_points**2 * planarity
        score += num_points**2
    return score, leaf_cluster_id_2_stats
```

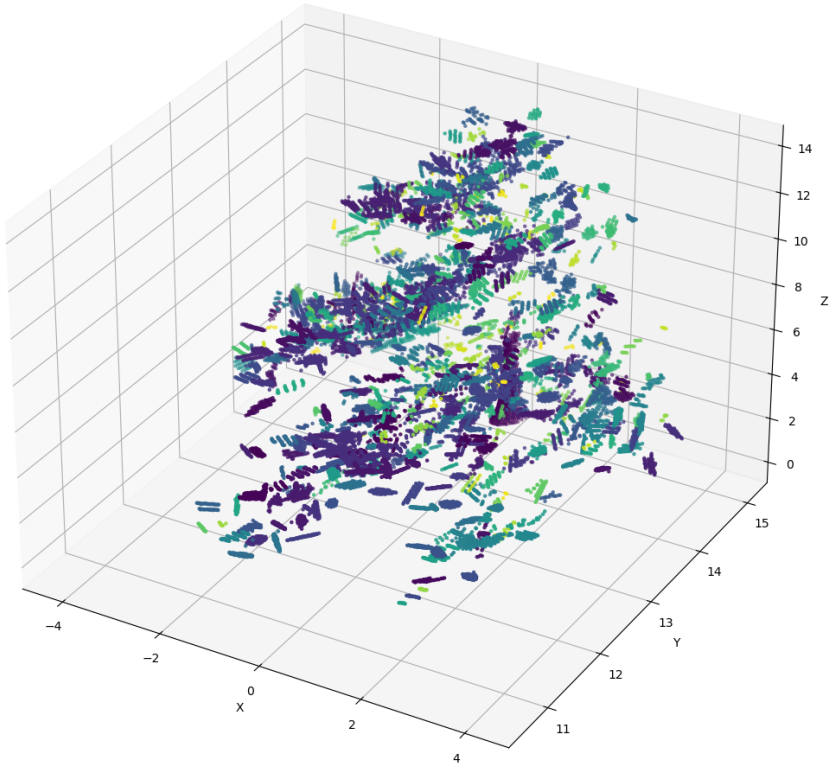
As we can see in `score_function`, the **score is equal the sum of square of number of points in each leaf-like cluster**. This objective function encourages the clustering algorithm to produce clusters that are leaf-like and as large as possible (in terms of number of points), because we don't want out algorithm to segment leaves into many small clusters, that would cause over estimate the number of leaves and underestimate their sizes. That is how we **incoporated the leaf-like constraint into the objective function**. By maximizing this score, we can find a clustering that effectively segments the point cloud into individual leaves while minimizing the inclusion of non-leaf points.

2.3 Optimization Method

To optimize the objective function, we experimented with different clustering algorithms to segment the point cloud into clusters. We used **DBSCAN** and **RANSAC plane fitting** as our main clustering methods. There is no sophisticated optimization algorithm used here, we simply apply **exhaustive search over different hyperparameters and clustering methods**, and select the one which gives the highest score according to our objective function.

2.3.1 Initial Clustering

After removing all the ground points by Z thresholding, removing the back half of the tree which is mostly occluded from the LiDAR scanner, we first cluster the tree into smaller clusters using DBSCAN with fixed parameters (eps=0.07, min_samples=5). We then optimize the clustering within each smaller cluster using different clustering methods and hyperparameters.



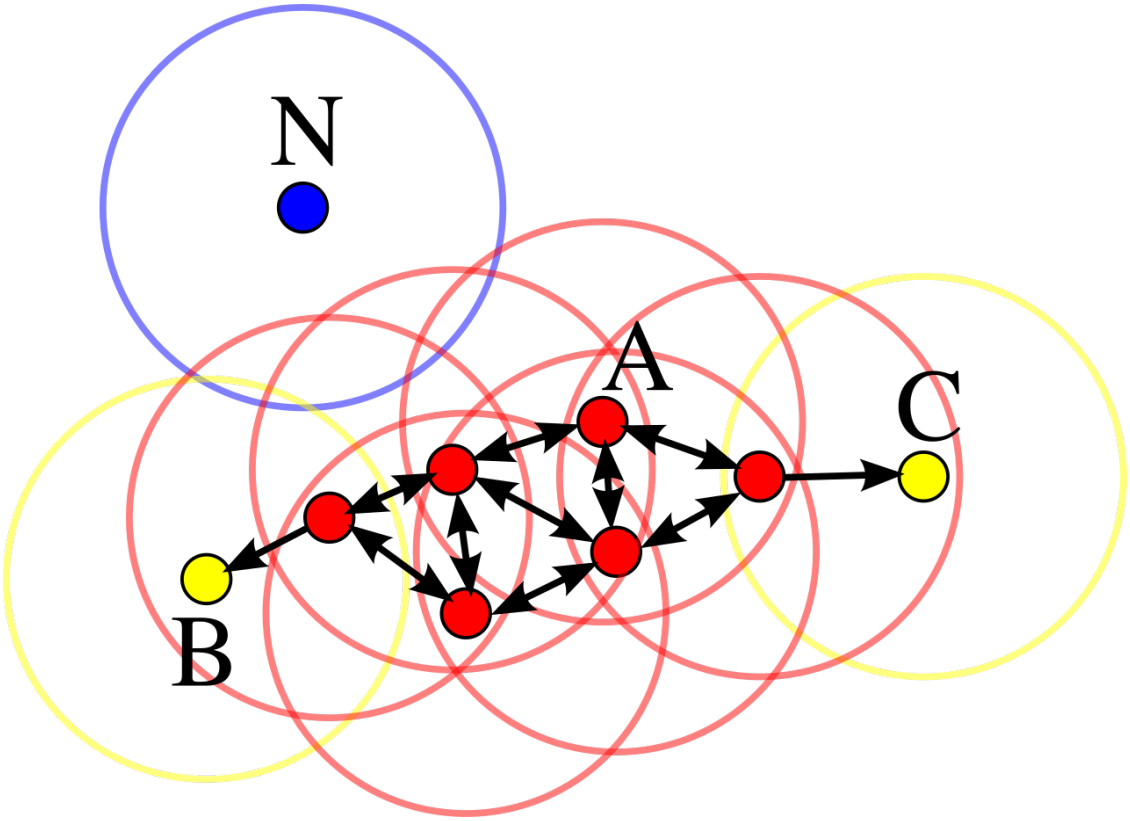
2.3.2 DBSCAN

```
from sklearn.cluster import DBSCAN
from sklearn.linear_model import RANSACRegressor

def DBSCAN_cluster_optimization(cluster_points, min_points=15, max_points=80,
                                min_ratio=10, plane_threshold=0.02):
    """
        Receive some clustered points and try different DBSCAN parameters to␣
 ↪find the best clustering
        that maximizes the score_function.

        Returns: best_score, best_num_leaf_clusters, clustering_assignment,␣
 ↪leaf_label_set
```

```
        clustering_assignment: array of cluster labels for each point in␣
↪cluster_points
        leaf_label_set: set of cluster labels that are considered leaf-like
        best_score: highest score achieved
        best_num_leaf_clusters: number of leaf-like clusters found at best score
    """
    best_score = -1
    best_num_leaf_clusters = 0
    clustering_assignment = -1 * np.ones(cluster_points.shape[0], dtype=int)
    leaf_label_set = set()

    list_eps = np.arange(0.1, 0.01, -0.005)
    list_min_samples = [5, 10, 15, 20]

    for min_samples in list_min_samples:
        for eps in list_eps:

            scaled_points = cluster_points.copy()
            dbscan = DBSCAN(eps=eps, min_samples=min_samples, n_jobs=16)
            labels = dbscan.fit_predict(scaled_points)

            # # the number of points in -1 cluster should be less than 20% of␣
↪total points
            num_noise_points = np.sum(labels == -1)
            if num_noise_points > 0.2 * scaled_points.shape[0]:
                continue  # skip this scale

            # use ransac to remove points that are not on planes?
            # for each cluster, fit a plane using ransac, keep only inlier␣
↪points
            for label in set(labels):
                if label == -1: # don't process noise
                    continue
                cluster_points_ransac = scaled_points[labels == label]
                if cluster_points_ransac.shape[0] < 3:
                    continue
                ransac = RANSACRegressor(residual_threshold=plane_threshold,
                                         min_samples=3, max_trials=300)
                X = cluster_points_ransac[:, :2]  # use x and y as input
                y = cluster_points_ransac[:, 2]    # use z as output
                ransac.fit(X, y)

                inlier_mask = ransac.inlier_mask_
                cluster_points_ransac = cluster_points_ransac[inlier_mask]
                # update labels
                labels_indices = np.where(labels == label)[0]
                for i, idx in enumerate(labels_indices):
```

```
                    if not inlier_mask[i]:
                        # remove outlier point
                        labels[idx] = -1  # mark as noise

            score, leaf_cluster_id_2_stats = score_function(scaled_points,
                                                    labels,
                                                    min_points=min_points,
                                                    max_points=max_points,
                                                    min_ratio=min_ratio,
                                                    plane_threshold=plane_threshold)

            if score > best_score or (np.abs(score - best_score) <= 5 and␣
↪len(leaf_cluster_id_2_stats) > best_num_leaf_clusters):
                best_score = score
                best_num_leaf_clusters = len(leaf_cluster_id_2_stats)
                leaf_label_set = set(leaf_cluster_id_2_stats.keys())
                # outlier points are labeled as -1
                clustering_assignment = labels.copy()

    return best_score, best_num_leaf_clusters, clustering_assignment,␣
↪leaf_label_set
```

Here, DBSCAN is choosen because it is able to find clusters of arbitrary shapes and sizes, and these clusters are sperated by sparse regions, which aligns well with our leaf cluster identification criteria. Here is a short explanation of DBSCAN:



In this diagram, minPts = 4. Point A and the other red points are core points, because the area surrounding these points in an radius contain at least 4 points (including the point itself). Because they are all reachable from one another, they form a single cluster. Points B and C are not core points, but are reachable from A (via other core points) and thus belong to the cluster as well. Point N is a noise point that is neither a core point nor directly-reachable. [wikipedia]

Also, in this algorithm, we use **RANSAC plane fitting** to refine the clusters found by DBSCAN. **RANSAC (RANdom SAmple Consensus)** can be used to find a planar fit for a set of 3D points, while being robust to outliers. In our case, we treat the x, y coordinates as inputs and z as the output to find a planar fit. Points within each cluster that are identified as outliers to this plane (based on a predefined threshold) are re-labeled as noise (-1). This step helps to further refine the clusters by removing points that do not conform to the planar structure expected of leaf surfaces. A short introduction to RANSAC can be found in Fouhey, 2011.

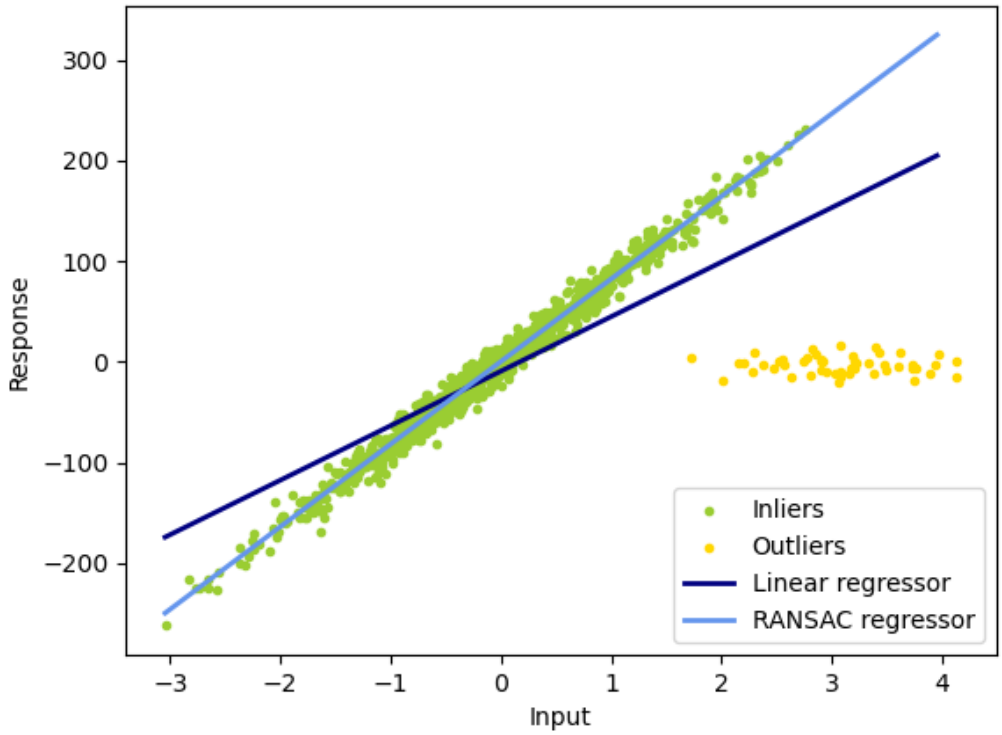


image source: sklearn RANSAC example

The `DBSCAN_cluster_optimization` function clustering the set of points following these steps:

1. **Choose a set of parameter combinations** for `eps` and `min_samples` to try.
2. **Apply DBSCAN**: For each parameter combination, run the DBSCAN algorithm on the cluster_points to generate initial cluster labels.
3. **Initial Noise Filter**: Check the percentage of points classified as noise (label -1). If more than 20% of the total points are noise, discard this parameter set and move to the next iteration.
4. **Planar Refinement (RANSAC)**: Iterate through each identified cluster and attempt to fit a plane to the points using `RANSACRegressor`. It treats the x, y coordinates as inputs and z as the output to find a planar fit.Points within the cluster that are identified as outliers to this plane (based on the `plane_threshold`) are re-labeled as noise (-1).
5. **Scoring**: Calculate a fitness score for the current clustering using the `score_function`.
6. **Optimization**: Compare the current score against the `best_score` recorded so far. **Update the best result if: The current score is strictly higher than the best score. OR the current score is within 5 points of the best score, but the current configuration yields a higher number of leaf clusters.**
7. **Return**: Finally, return the configuration that produced the best score, including the cluster assignments, the count of leaf clusters, and the set of valid leaf label

2.3.3 RANSAC

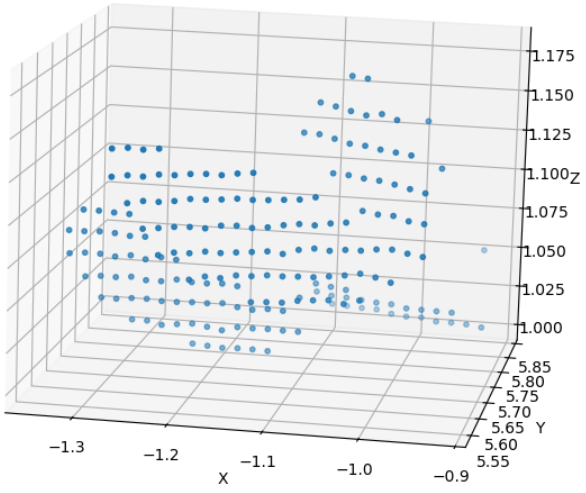
Another way to cluster the points is to first use RANSAC to find planar clusters in the point cloud. But the planes found by RANSAC are not bounded by point density, these planes will cut through multiple objects in the point cloud. Therefore, after finding planes using RANSAC, we further cluster the points within each plane using DBSCAN to separate different objects that lie on the same plane.

```
def RANSAC_cluster_optimization(cluster_points, distance_threshold=0.01,
                                min_inliers=20, dbscan_min_points=15,
                                dbscan_max_points=80, dbscan_min_ratio=10,
                                dbscan_plane_threshold=0.02):
    """
        Sequential RANSAC to find leaf planes in 3D point clouds.
        After finding a plane using RANSAC, use DBSCAN to separate different␣
↪clusters within the plane.
        If the clusters found by DBSCAN are leaf-like, keep them,
            and remove the clusters which are not leaf-like (for the next␣
↪iterations).
        Repeat until no more planes can be found with enough inliers,
            or no new inliers are found even after several iterations.
    """
    remaining_points = cluster_points.copy()
    leaf_planes = []

    list_len_remaining = []
    while remaining_points.shape[0] >= min_inliers:
        # fit RANSAC plane model
        ransac = RANSACRegressor(residual_threshold=distance_threshold,
                                 min_samples=3, max_trials=300)
        X = remaining_points[:, :2]  # use x and y as input
        y = remaining_points[:, 2]    # use z as output
        ransac.fit(X, y)

        inlier_mask = ransac.inlier_mask_
        inlier_points = remaining_points[inlier_mask]

        # there can be multiple cluster in inlier points so we want to separate␣
↪them
        # use DBSCAN to separate them, find the best DBSCAN clustering and␣
↪remove good clusters.
        best_score, best_num_leaf_clusters,\
            clustering_assignment,\
                leaf_label_set = DBSCAN_cluster_optimization(inlier_points,
                                                min_points=dbscan_min_points,
                                                max_points=dbscan_max_points,
                                                min_ratio=dbscan_min_ratio,
```

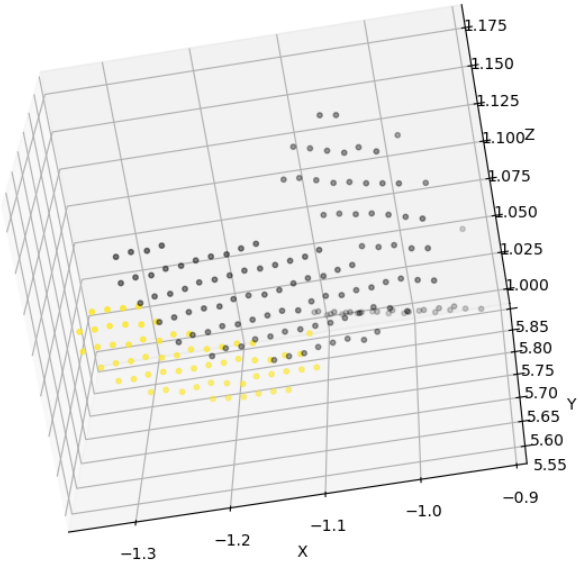
```
                                        plane_threshold=dbscan_plane_threshold)

        # # set inlier_points to only those points that are labeled as␣
↪leaf-like clusters
        for cluster_id in leaf_label_set:
            cluster_points = inlier_points[clustering_assignment == cluster_id]
            # if cluster_points.shape[0] >= min_inliers:
            leaf_planes.append(cluster_points)

        new_inlier_mask = list()
        inlier_points_pos = 0
        for i in range(inlier_mask.shape[0]):
            if inlier_mask[i]:
                if clustering_assignment[inlier_points_pos] in leaf_label_set:
                    new_inlier_mask.append(True)
                else:
                    new_inlier_mask.append(False)
                inlier_points_pos += 1
            else:
                new_inlier_mask.append(False)

        # remove inliers from remaining points
        outlier_mask = np.logical_not(new_inlier_mask)
        remaining_points = remaining_points[outlier_mask]

        len_remaining = remaining_points.shape[0]
        list_len_remaining.append(len_remaining)
        # if 3 time and still no change, break
        if len(list_len_remaining) >= 5:
            if all(x == list_len_remaining[-1] for x in list_len_remaining[-3:
↪]):
                break

    return leaf_planes
```

For comparision, these are the resulting clusters from DBSCAN and RANSAC methods on the same cluster of points:

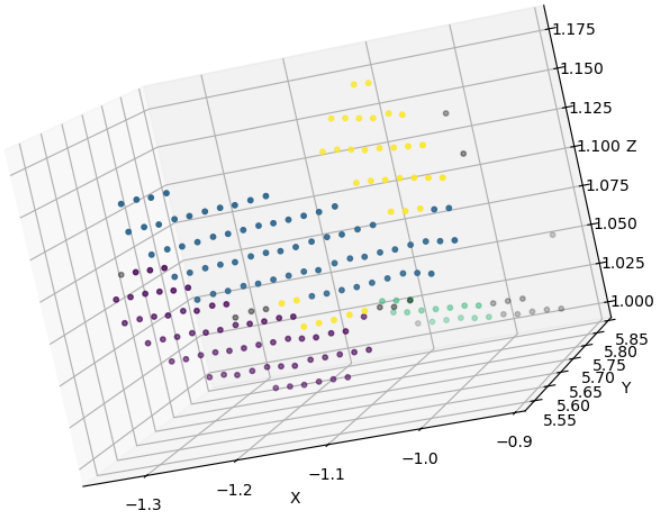
Original Point Cloud:



DBSCAN Clustering Result:



RANSAC Clustering Result:



We can see that when the point cloud is dense with no clear separation between objects, DBSCAN tends to group multiple leaves into a single cluster. On the other hand, RANSAC is able to separate these leaves into different clusters by fitting planes and then using DBSCAN within each plane. But cutting a plane through multiple objects can also lead to over-segmentation, where a single leaf is split into multiple clusters if it is not perfectly planar, so we still keep both methods and choose the best one based on our objective function.

2.3.4 Final cluster assignment optimization function

The final optimization function then compare clutering results from both DBSCAN and RANSAC methods, and select the one which gives the highest score according to our objective function.

```
def optimize_for_cluster(cluster_points, ransac_only=False):
    """
        This function return the best subclustering labels for the given␣
 ↪cluster points
        by trying both DBSCAN and RANSAC methods, and selecting the one with␣
 ↪the best score.
        1. Try DBSCAN to find subclusters
        2. Try sequential RANSAC to find leaf planes
        3. Return the best clustering assignment.
```

```
    """
    # try DBSCAN first
    best_score = -1
    clustering_assignment = -1 * np.ones(cluster_points.shape[0], dtype=int)
    leaf_label_set = set()
    best_num_leaf_clusters = 0
    if not ransac_only:
        best_score, best_num_leaf_clusters, \
            clustering_assignment,\
            leaf_label_set = DBSCAN_cluster_optimization(cluster_points,
                        min_points=param['dbscan']['min_points'],
                        max_points=param['dbscan']['max_points'],
                        min_ratio=param['dbscan']['min_ratio'],
                        plane_threshold=param['dbscan']['plane_threshold'])

    # # Try RANSAC
    best_leaf_planes = []
    scaled_points = cluster_points.copy()
    leaf_planes = RANSAC_cluster_optimization(scaled_points,
                distance_threshold=param['ransac']['distance_threshold'],
                min_inliers=param['ransac']['min_inliers'],
                dbscan_min_points=param['ransac']['dbscan_min_points'],
                dbscan_max_points=param['ransac']['dbscan_max_points'],
                dbscan_min_ratio=param['ransac']['dbscan_min_ratio'],
               ␣
↪dbscan_plane_threshold=param['ransac']['dbscan_plane_threshold'])
    # score is total number of points in leaf planes
    score = sum(plane.shape[0]**2 for plane in leaf_planes)

    if score > best_score or (np.abs(score - best_score) <= 5 and␣
↪len(leaf_planes) > best_num_leaf_clusters):
        best_score = score
        best_num_leaf_clusters = len(leaf_planes)
        best_leaf_planes = leaf_planes

        # translate best_leaf_planes to clustering_assignment
        if len(best_leaf_planes) > 0:
            clustering_assignment = -1 * np.ones(cluster_points.shape[0],␣
↪dtype=int)
            cluster_id = 0
            leaf_label_set = set()
            for plane in best_leaf_planes:
                for point in plane:
                    # find index of point in cluster_points
                    indices = np.where((np.round(cluster_points, decimals=4) ==␣
↪np.round(point, decimals=4)).all(axis=1))[0]
```

```
                    assert len(indices) > 0, "Point from leaf plane not found␣
↪in original cluster points"
                    for index in indices:
                        clustering_assignment[index] = cluster_id
                leaf_label_set.add(cluster_id)
                cluster_id += 1
        print(f"After RANSAC: Best Score: {best_score}, Best Number of␣
↪leaf-like clusters: {best_num_leaf_clusters}")
    else:
         print(f"After DBSCAN: Best Score: {best_score}, Best Number of␣
↪leaf-like clusters: {best_num_leaf_clusters}")

    return best_score, best_num_leaf_clusters, clustering_assignment,␣
↪leaf_label_set
```

Each cluster from initial clustering is processed independently, so the optimization can be parallelized for faster computation. The whole optimization process's duration is around 1.5 minutes on our personal computer with 16 CPU cores.

```
from joblib import Parallel, delayed
def process_cluster(cluster_label, scaled_points, labels):
    if cluster_label == -1:
        return cluster_label, None  # skip noise
    cluster_points = scaled_points[labels == cluster_label]

    best_score, best_num_leaf_clusters, clustering_assignment, leaf_label_set =␣
↪optimize_for_cluster(cluster_points)
    # print(f"Cluster {cluster_label}: Best Score (leaf-like clusters):␣
↪{best_score}, Number of leaf-like clusters: {best_num_leaf_clusters}")
    return cluster_label, (best_score, best_num_leaf_clusters,␣
↪clustering_assignment, leaf_label_set)
results = Parallel(n_jobs=16)(delayed(process_cluster)(cluster_label,␣
↪scaled_points, labels) for cluster_label in set(labels))
cluster_id_2_best_clustering = {cluster_label: result for cluster_label, result␣
↪in results if result is not None}
```

You can also see that somethings very important is missing here. In `optimize_for_cluster` function, we use `param` but nowhere to define it. The `param` variable is a dictionary that contains the hyperparameters for the RANSAC and DBSCAN algorithms used in the clustering process. The hyperparameters in `param` are tuned by manually, based on visual inspection of the clustering results on a few sample clusters from the point cloud. The chosen hyperparameters are:

```
param = {
    'dbscan':{
        'min_points':12,
        'max_points':70,
        'min_ratio':20,
```

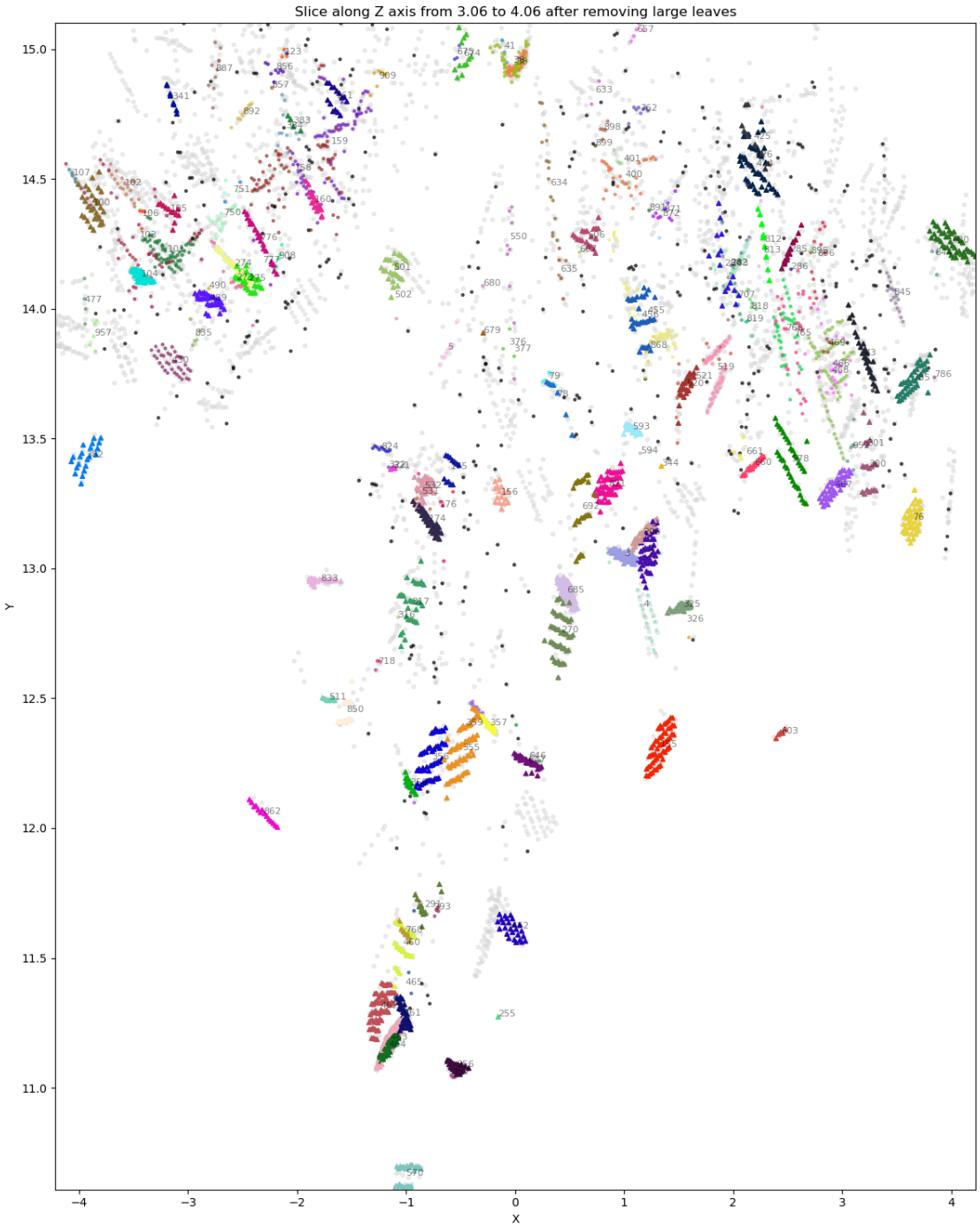
```
        'plane_threshold':0.04,
    },
    'ransac':{
        'distance_threshold':0.04,
        'min_inliers':12,
        'dbscan_min_points':12,
        'dbscan_max_points':70,
        'dbscan_min_ratio':20,
        'dbscan_plane_threshold':0.04,
    }
}
```

It is very hard to inspect the clustering results in 3D visually, so we use CT scans along different axes to evaluate the clustering quality. Clusters which look like leaves should be detected as leaves, and different leaves should be separated in the CT scans.

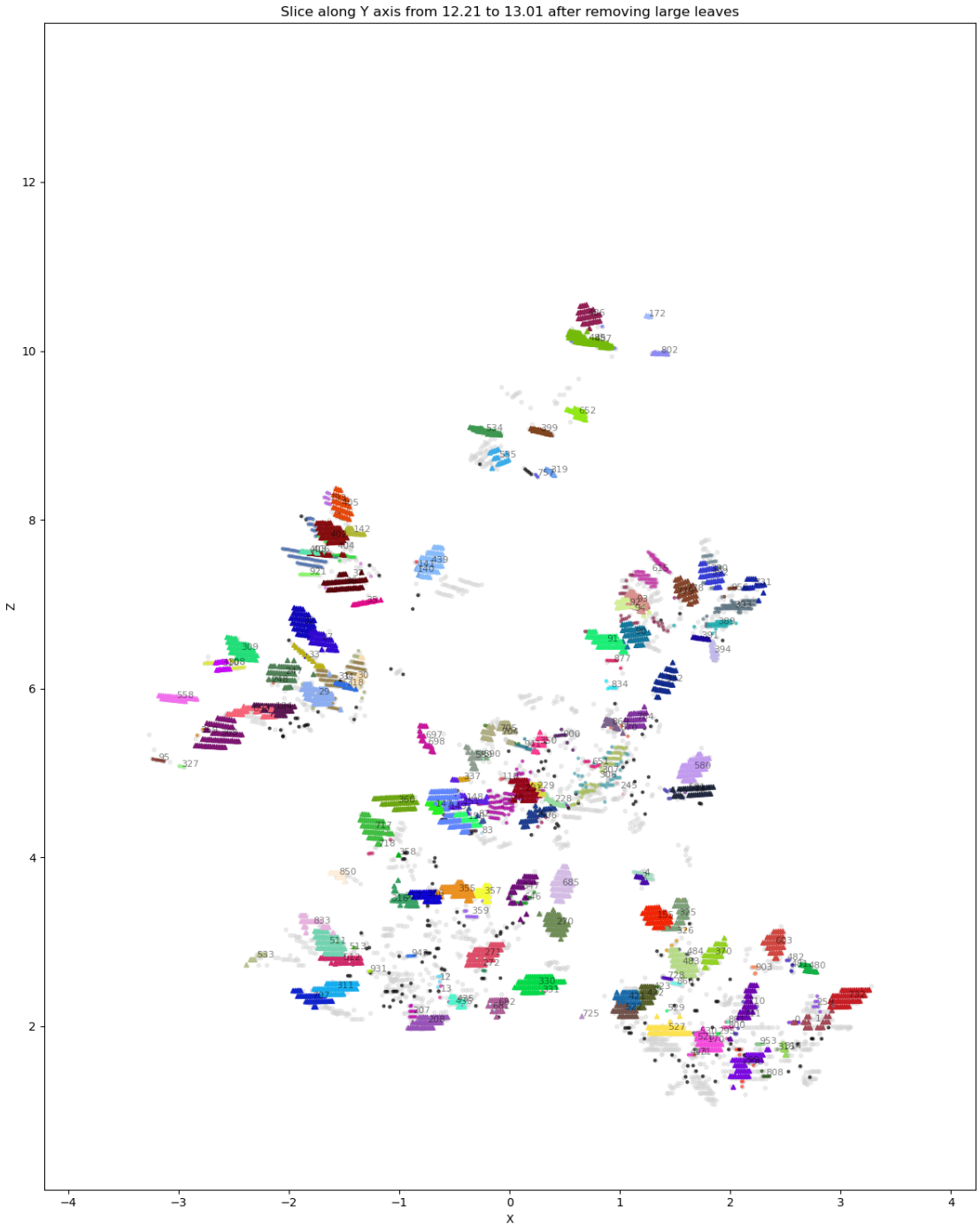
In the below CT scans, you can see that most leaves are correctly identified and separated.

- **Black dots** are noise points, seperated by the initial clustering
- **Colored dots** are clusters which failed leaf-like criteria
- **Colored triangles** are identified leaf clusters
- **Gray faded dots** are occluded data, which we was able to uncovered by using HELIOS++ simulation.

CT scan along Z axis:



CT scan along Y axis:

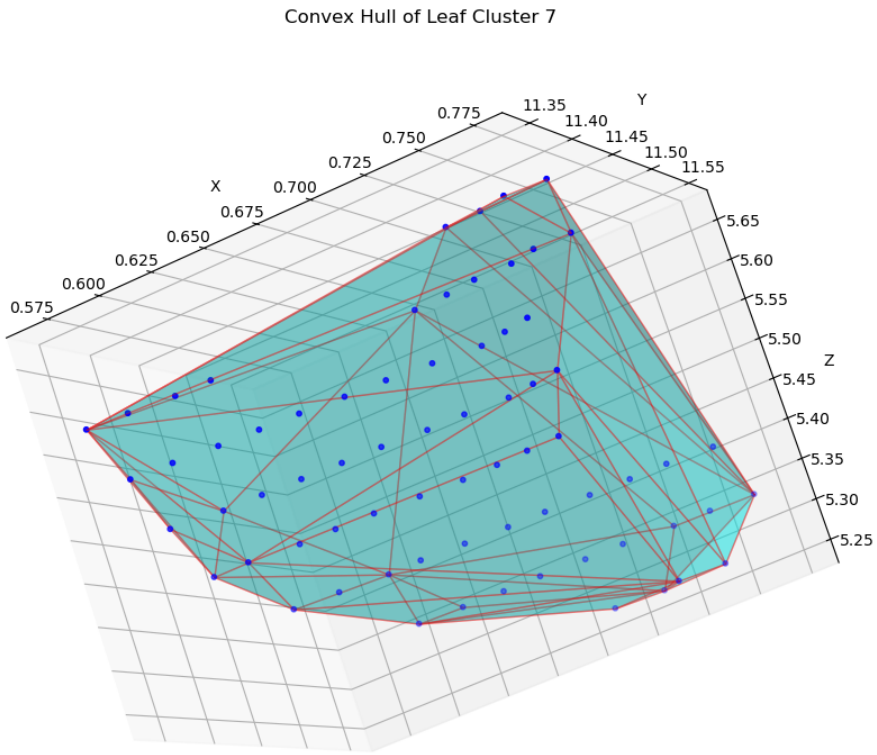


2.4 Area Estimation and Leaf Counting

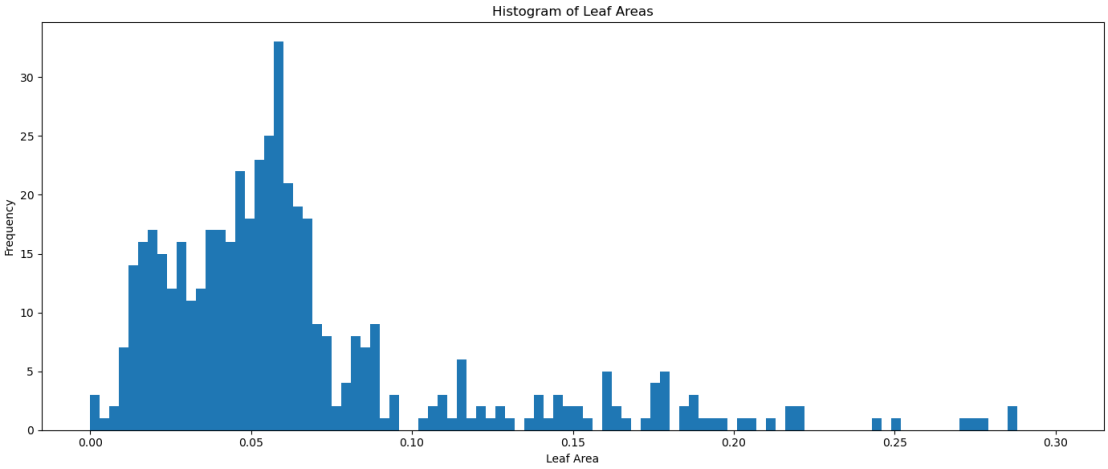
After identifying the leaf clusters, we estimate the area of each leaf cluster using the convex hull method. The convex hull of a set of points is the smallest convex shape that encloses all the points. For a leaf cluster, we compute the convex hull of its points and calculate its surface area. Since leaves are generally flat surfaces, we **approximate the leaf area as half the surface area of the convex hull.**

```
from scipy.spatial import ConvexHull
leaf_areas = []
leaf_largest_distances = []
cluster_label_2_area = {}
cluster_label_2_largest_distance = {}
count = 0
for cluster_label in set(new_labels):
    if cluster_label == -1:
        continue  # skip noise

    points = orig_points[new_labels == cluster_label]
    is_leaf = is_leaf_label[new_labels == cluster_label]

    if not np.all(is_leaf):
        continue  # not a leaf cluster

    # compute convex hull
    try:
        hull = ConvexHull(points)
    except:
        print(f"Cluster {cluster_label} cannot find convex hull.")
        continue  # skip clusters that cannot be triangulated

    # compute largest distance between points in the cluster
    from scipy.spatial.distance import pdist
    largest_distance = np.max(pdist(points))
    cluster_label_2_largest_distance[cluster_label] = largest_distance
    leaf_largest_distances.append(largest_distance)

    # compute area of each triangle
    area = hull.area / 2.0  # Leaft area is half of the convex hull area
    leaf_areas.append(area)
    cluster_label_2_area[cluster_label] = area
```

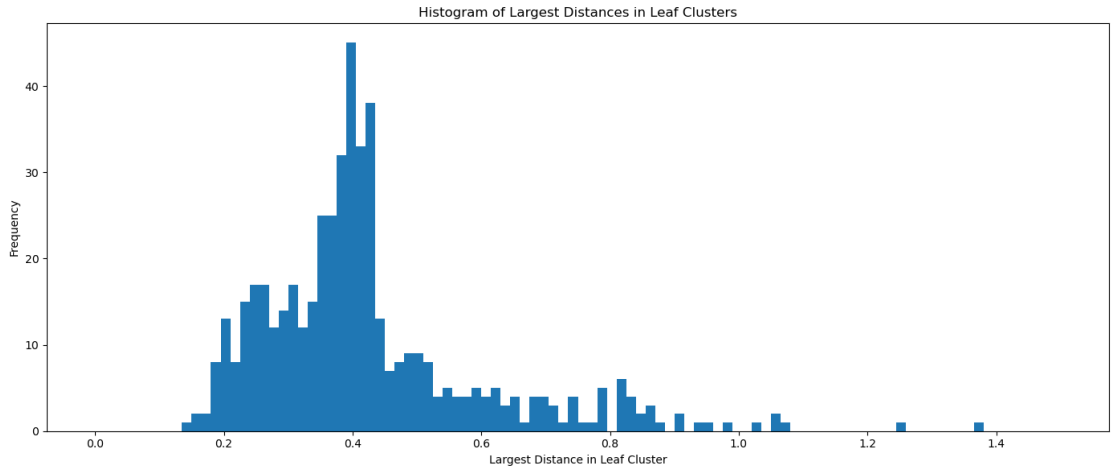
A typical convex hull of a leaf cluster



Histogram of estimated leaf areas



Histogram of largest distance between points in leaf clusters



For the final leaf count and total leaf area estimation, we apply a threshold to filter out suspiciously large clusters that are likely to be spurious leaves. We set a maximum area threshold of 0.2 m^2; any leaf cluster with an estimated area above this threshold is discarded from the final count and area sum. We discard leaves with largest distance between points above 1 m for the same reason. The reason for these large thresholds is we alow cases where mutiple leaves are clustered together in dense regions, leading to larger clusters that still represent valid leaf structures.

And this is how we **incorporated shape and size constraints** into our leaf counting and area estimation process.

3 Results

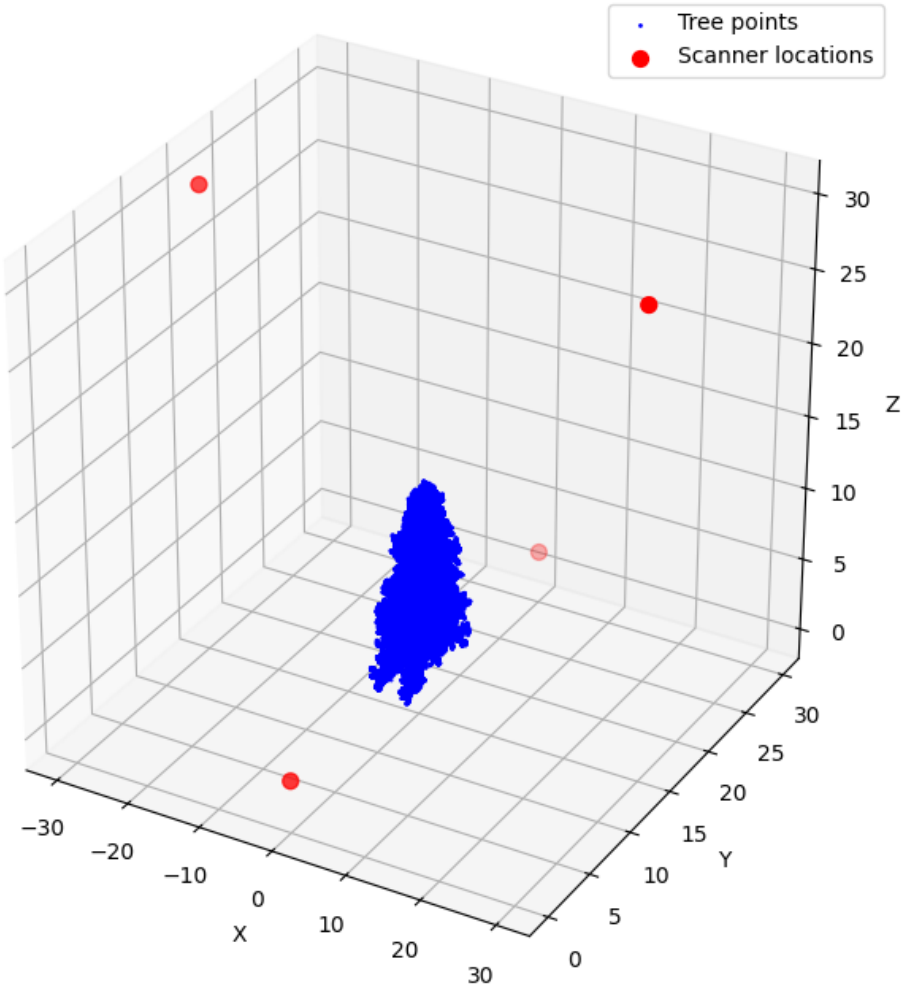
For the final results, after filtering our spurious clusters, and multiply everything by 2, as our input data was only half of the tree, we estimated a **total of 916 leaves with a combined leaf area of approximately 55.59 m^2.**

3.1 Objective Evaluation

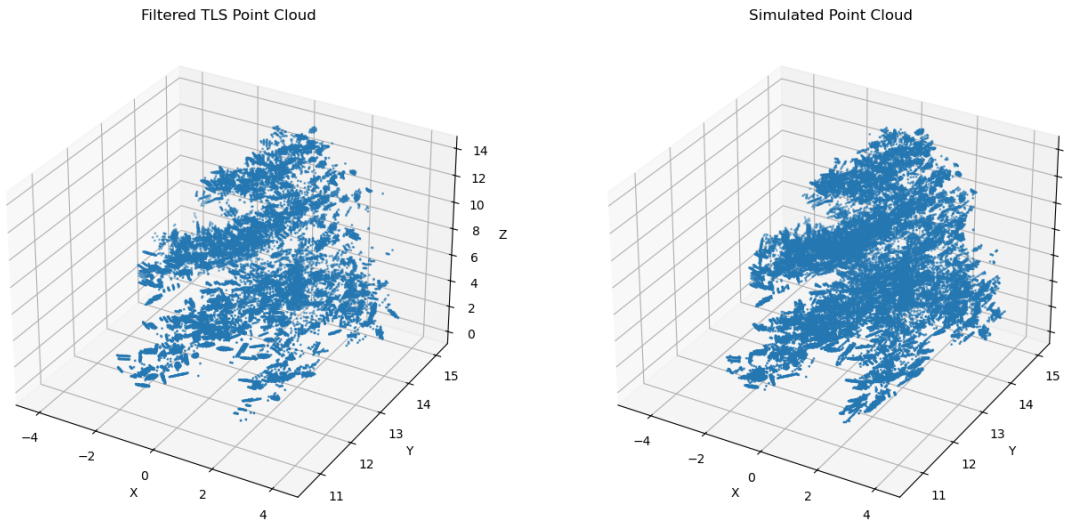
As stated earlier, there is no ground truth information in this dataset, so we only rely on qualitative evaluation using CT scans and visual inspection of clustering results.

However, to estimate the effect of occlusion, luckily, we have founded the same exact tree model in HELIOS++ dataset, we then scan it from multiple angles to obtain a more complete point cloud of the tree. In this way, we can compare the leaf count and area estimation results from our single-scan data against the more complete multi-scan data to assess the impact of occlusion on our estimates.

Position of scanner in HELIOS++ multi-scan data:

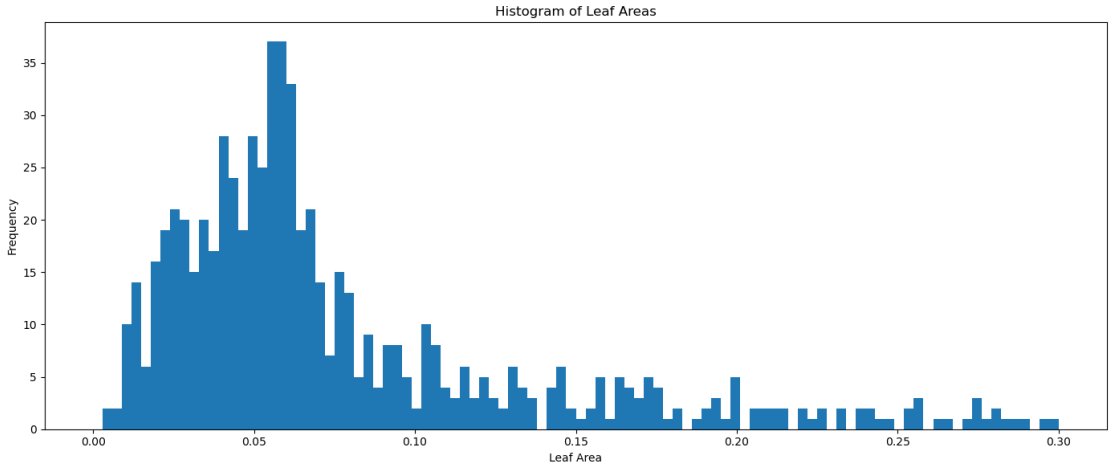


Compared to our single-scan data, the multi-scan data is denser and have duble the number of points:

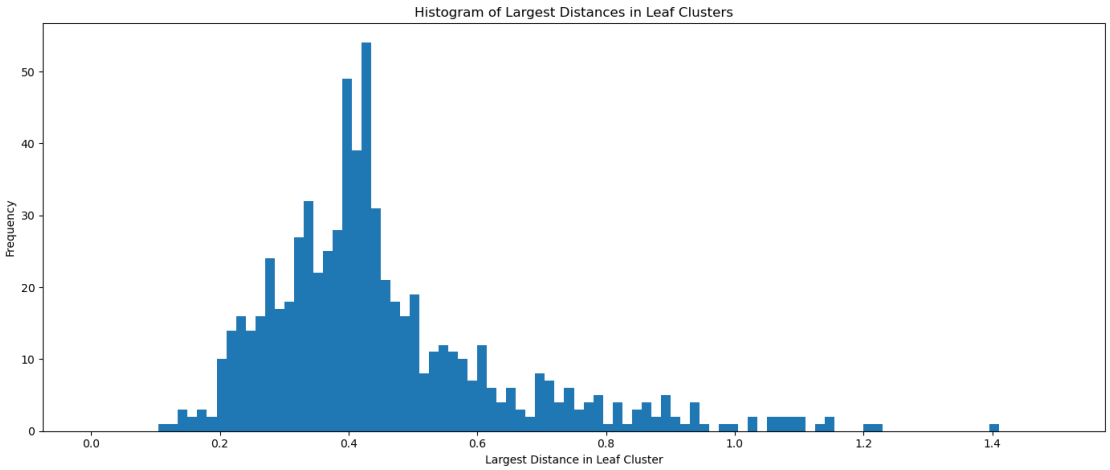


Rerun our leaf counting and area estimation pipeline on the HELIOS++ multi-scan data, we obtained a total of **1256 leaves with a combined leaf area of approximately 83.52 m^2.**

Historgram of estimated leaf areas from HELIOS++ multi-scan data



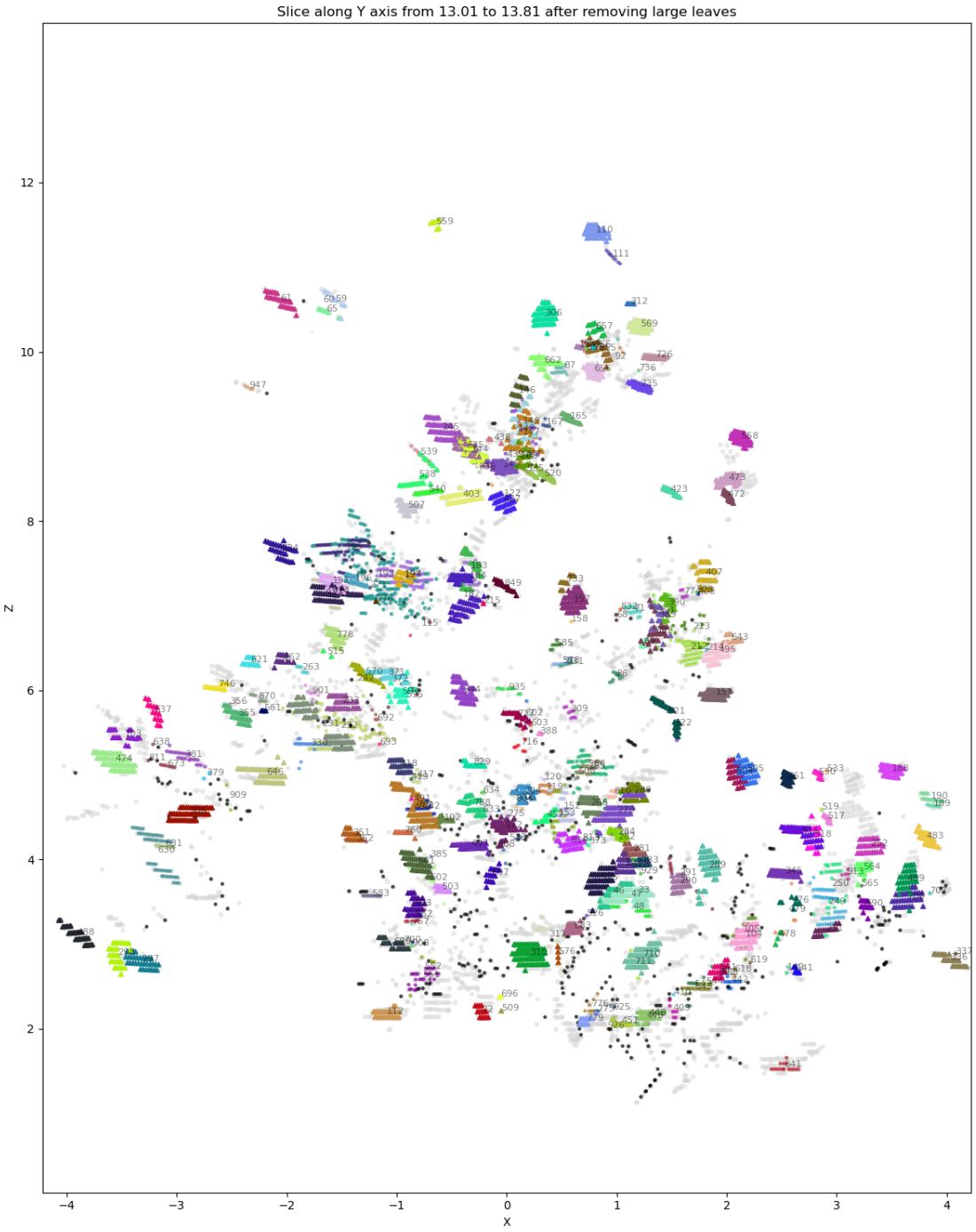
Histogram of largest distance between points in leaf clusters from HELIOS++ multi-scan data



We can see in the histograms that the average leaf area is about 0.06 m^2 and the average largest distance between points in leaf clusters is about 0.4 m, which are similar to the results from our single-scan data.

So, **underestimation due to occlusion in our single-scan data is approximately: 27% for leaf count and 33% for leaf area.** The percentage of leaf count is lower than leaf area, because in casses where even half of a leaf is visible, our algorithm can still detect it as a leaf cluster, contributing to the leaf count, but the estimated area will be lower due to occlusion.

This magnitude of underestimation seems reasonable given the significant occlusion observed in the single-scan point cloud. It highlights the importance of multi-angle scanning for more accurate leaf quantification in complex canopies. We can see there are a lot of occluded leaves (gray faded dots) in this CT scan:



3.2 Robustness

There are many hand tuned hyperparameters in our method, from the initial clustering to the leaf-like cluster identification criteria. These hyperparameters are tuned based on visual inspection of clustering results on a few sample clusters from the point cloud. Therefore, the robustness of our method depends on how well these hyperparameters generalize to different tree structures and point cloud densities.

4 Summary

We proposed a pipeline to identify leaves from a tree point cloud obtained by a terrestrial LiDAR scanner.

The method go through several steps:

1. Keep only the tree half which faces the LiDAR scanner. (to mitigate occlusion issues)
2. Cluster the tree into smaller clusters using DBSCAN to reduce the search space for the next step. DBSCAN is choosen because it is able to find clusters of arbitrary shapes and sizes, and these clusters are sperated by sparse regions, which aligns well with our leaf cluster identification criteria.
3. For each smaller cluster, find the best clustering which gives the most leaf-like clusters. The clustering proposing methods are: **DBSCAN**; **RANSAC** plane fitting. The leaf-like clusters are identified by **shape analysis**. We use **exhaustive search over different hyperparameters and clustering methods**, and select the one which gives the highest score according to our **objective function**.
4. We compute leaf area for each identified leaf cluster and sum them up to get the total leaf area. The leaves areas are aproximated by half the **area of the convex hull** of the leaf clusters, because we assume the leaves are flat surfaces.
5. We only keep the leaves which have area below 0.2 m^2 and largest dimension below 1.0 m to **avoid suspiciously large clusters.**
6. We visualize the final result and show **CT scans along different axes, for qualitative evaluation.**
7. We report the total leaf area as our estimation of leaf area of the tree.
8. We use **simulated data from HELIOS++** for estimate the effect of occlusion.

5 Appendix

5.1 Github Repository

The code for this project can be found in this GitHub repository