Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Building Models of Protected Trees of Exceptional Age

Jan Svoboda

Supervisor: RNDr. Petr Štěpán, Ph.D.
Study program: Open Informatics
Specialisation: Artificial Intelligence and Computer Science
May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Svoboda Jan**	Personal ID number:	**508501**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Building Models of Protected Trees of Exceptional Age

Bachelor’s thesis title in Czech:

Vytváření modelů chráněných stromů mimořádného stáří

Guidelines:

1) Learn the principle of Lidar operation and the algorithms for processing Lidar data.
2) Design an algorithm that will detect the trunks of individual trees in a multi-tree environment.
3) Extend the algorithm to detect main branches and test it on the provided dataset.
4) Propose a method for fusing data from multiple sensor positions and test the proposed approach to create a trunk and main branch model for protected trees of exceptional age.

Bibliography / sources:

[1] T. Zmeškalová: “Detekce stromů pro lokalizaci UAV“, bakalářská práce, katedra kybernetiky, FEL, ČVU v Praze, 2023
[2] B. Douillard et al., "On the segmentation of 3D LIDAR point clouds," 2011 IEEE International Conference on Robotics and Automation, Shanghai, China, 2011, pp. 2798-2805, doi: 10.1109/ICRA.2011.5979818.
[3] X. Chen, A. Milioto, E. Palazzolo, P. Giguère, J. Behley and C. Stachniss, "SuMa++: Efficient LiDAR-based Semantic SLAM," 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Macau, China, 2019, pp. 4530-4537, doi: 10.1109/IROS40897.2019.8967704
[4] Kusenbach, M., Himmelsbach, M., & Wuensche, H. J. (2016, June). A new geometric 3D LiDAR feature for model creation and classification of moving objects. In 2016 IEEE Intelligent Vehicles Symposium (IV) (pp. 272-278). IEEE.

Name and workplace of bachelor’s thesis supervisor:

RNDr. Petr Štěpán, Ph.D. Multi-robot Systems FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **31.01.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

RNDr. Petr Štěpán, Ph.D.	prof. Dr. Ing. Jan Kybic	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Acknowledgements

I would like to express my thanks to RNDr. Petr Štěpán, Ph.D., for all his help and provided expertise during the course of writing this thesis.

All LIDAR scans were captured and provided by RNDr. Petr Štěpán, Ph.D.

ChatGPT 3.5 was utilized to perform tasks such as customizing this LaTeX template, troubleshooting LaTeX errors in general, setting up Biblatex, and generating several random colors for coloring individual clouds. No AI has been used to directly create the content of this thesis.

All point clouds were processed using the Point Cloud Library. The program was written in C++. The visualizations of point clouds were created by either the *PCLVisualizer*[Lib], or VS Code extensions - *vscode-3d-preview*[Tat] and *pcd-viewer*[Azu].

Declaration

I declare that I have prepared the submitted work independently and that I have presented all information sources used in accordance with the Methodological Instructions on compliance with ethical principles in the preparation of university theses.

In Prague, 24. May 2024

Abstract

This thesis deals with developing algorithms for creating tree models from LIDAR data. The thesis proposes a method to remove the ground and detect the tree trunk and the branches attached to it. The algorithm works with real data from an experiment with 400-year-old pedunculate oaks standing on the dam of the Homolka pond in Prague. From the acquired data, a graph modeling the structure of the trunk and branches was created. The algorithm was also used to reconstruct the trunk and branch structure from data taken from multiple LIDAR sensor positions.

Keywords: LIDAR, tree model, Point Cloud Library, ICP

Supervisor: RNDr. Petr Štěpán, Ph.D.
Praha, Resslova 307/9

Abstrakt

Tato práce se zabývá vývojem algoritmů pro vytváření modelů stromů z dat LIDARu. V práci je navržen postup pro odstranění země a pro detekci kmene stromu a na něj navazujících větví. Algoritmus pracuje s reálnými daty z experimentu se 400 let starými duby letními stojícími na hrázi rybníka Homolka v Praze. Z pořízených dat byl vytvořen graf modelující strukturu kmene a větví. Pomocí algoritmu se podařilo zrekonstruovat také strukturu kmene a větví z dat pořízených z více pozic LIDAR senzoru.

Klíčová slova: LIDAR, model stromu, Point Cloud Library, ICP

Contents

1 Introduction **1**

2 LIDAR **3**
2.1 Point Cloud Library 4
2.1.1 Introduction 4
2.1.2 Point Cloud 4
2.1.3 PCL Modules 4

3 Tree Trunk Detection **7**
3.1 Algorithm Outline 7
3.2 Loading the Input Point Cloud . . 7
3.3 Surface Normal Estimation 8
3.4 Ground Removal 9
3.4.1 Planar Segmentation 9
3.4.2 Pass-Through Filter 11
3.5 Region Growing Segmentation . . 14
3.6 Cylinder Segmentation 15

4 Trunk and Main Branches Detection **17**
4.1 Whole Tree Region Growing Segmentation 17
4.2 Tree Cluster Cylinder Segmentation 18
4.2.1 Euclidean Cluster Extraction 19
4.2.2 Cylinder Radius Optimization 20
4.2.3 Adding Points to Optimized Cylinders 22
4.3 Adjacent Branches Identification 22
4.3.1 Oriented Bounding Box 22
4.3.2 Nearest Neighbor Search 26
4.3.3 Endpoint Nearest Neighbor . 27

5 Tree Graph **29**
5.1 Cylinder Axes as Skew Lines . . . 29

6 Fusing Multiple Scans Together **35**
6.1 Input Dataset Transformation . . 35
6.2 PCL ICP 36
6.3 Voxel Grid Filtering 37

7 Conclusion **41**

Bibliography **43**

Figures

2.1 LIDAR Scan Example[Wik23a] . . 3

3.1 Loaded Input Cloud 8
3.2 Ground Removal Result 11
3.3 Planar Segmentation, Incorrect Result 12
3.4 Ground Removal, Correct Result 13
3.5 Colored Clusters 15
3.6 All Cylinders 16

4.1 Example of a Selected Cluster . . 18
4.2 Isolated Sets of Points Belonging to One Cylinder 20
4.3 Cylinder Radius Optimization (left) vs No Optimization (right) . . 21
4.4 A Comparison of AABB and OBB of a Point Cloud[Kak21] 23
4.5 Checking for Disjoint Intervals[GML00] 24
4.6 Example of Overlapping OBBs . 25
4.7 An Incorrect Evaluation of Overlaps 26

5.1 Skew Lines: Case 1 30
5.2 Skew Lines: Case 2a and 2b 31
5.3 Skew Lines: Case 3 31
5.4 Tree Graph 32
5.5 Tree Graph with Points 33

6.1 Fitness Score Evaluation 36
6.2 An Odd Branch 37
6.3 ICP Tree Graph w/o Voxel Filter 38
6.4 ICP Tree Graph with Voxel Filter 38
6.5 A Photo of the Tree 39

Tables

4.1 Possible Separating Axes 24

6.1 Scan Sizes 35

Chapter 1
Introduction

Unmanned Aerial Vehicles (UAVs), such as drones, are emerging as increasingly prominent in society as they offer a large number of previously unimagined applications. Gone are the days of highly expensive UAVs that were only used by armies or specific organizations. As drones have become commercially available to the general population due to a drop in price, the range of their possible uses has expanded considerably.

These applications include many practical tasks such as plant disease detection for precision agriculture, plant and tree health surveys, and imaging and mapping of various objects in nature. Due to the compact size and weight of drones, they are well-suited for flying in places that would be difficult for humans to reach while retaining the ability to carry various accessories and sensors.

A LIDAR laser scanner is an important sensor for current UAVs. It provides information about the distances of objects around the drone. Using these data, the UAV is able to create a model of the environment it is moving through and plan its flight accordingly.

Detection of the condition of protected trees of exceptional age fits the description of applications suitable for UAVS. Since little research has been done on this topic, there is an opportunity to find further use for UAVs in this area.

This thesis first focuses on learning the principles of LIDAR operation and then considers algorithms for processing LIDAR data. Secondly, this knowledge is used to design an algorithm capable of detecting trunks of individual trees in a multi-tree environment. This algorithm is further extended to detect the main branches of selected trees.

Finally, a method for fusing data from multiple sensor positions is proposed. The combination of all these steps leads to the creation of trunk and main branch models of protected trees of exceptional ages. These models make it possible to track tree growth and possibly losses of the main branches of the trees.

Chapter 2
LIDAR

LIDAR (Light Detection And Ranging) is a method of determining distances to objects using a laser. Simply put, it employs the principle of reflecting a laser from an object and measuring the time it takes for the light to return to the sensor. More precisely:

$$d = \frac{c \times t}{2}$$

where d is the distance to the object, c is the speed of light, and t is the time it takes the laser to reach the object and reflect back to the sensor.

Old LIDAR devices were only capable of scanning in one plane. Modern sensors emit up to 128 beams at the same time and are able to scan the full 360° surroundings. The output of these LIDAR scanners are clusters of points (point clouds). Each of these points has coordinates in 3D space (relative to the position of the scanner). It is possible to add additional information, such as color, to the data recorded this way if an RGB camera is used in addition to LIDAR [MN20].

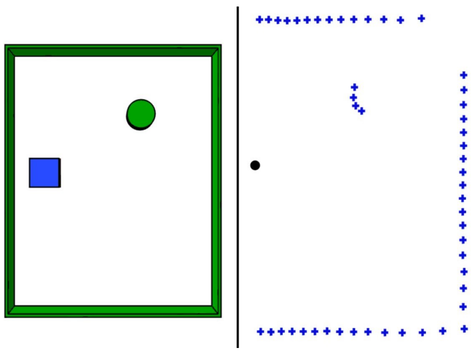


Figure 2.1: LIDAR Scan Example[Wik23a]

One of the shortcomings of LIDAR is apparent in Figure 2.1. The blue square represents a device equipped with a LIDAR scanner, and the green circle represents an obstacle. Just like a classic RGB camera, LIDAR cannot see through (nontransparent) objects, which creates spots in the scan that are "in a shadow," i.e., behind some other object in the LIDAR's scanning axis. Therefore, to capture a comprehensive scan of a complex object in a given space, we must take images from multiple positions and angles and then run software to combine these scans into one complete model.

LIDAR is used in a large number of scientific fields such as archaeology, geography, geology, seismology, forestry, laser guidance, and control and navigation of autonomous cars (or unmanned robots).

2.1 Point Cloud Library

2.1.1 Introduction

Point Cloud Library (PCL) is an open-source project enabling the processing of 2D and 3D images and point clouds. Several modern algorithms for filtering, segmentation, registration, and more are included in the PCL framework. These algorithms serve as a comprehensive set of tools for efficient work with point clouds. The library can work with data taken by a stereo camera, 3D scanner, and time-of-flight camera, as well as data artificially created by a computer[RC11][Lib][Rus09]. For proper functionality, Point Cloud Library requires the installation of additional libraries: Boost, Eigen, FLANN, and VTK, and is available on Linux, MacOS, Windows and Android. CMake is required to build the library[Wik23b].

2.1.2 Point Cloud

A point cloud is a data structure that represents a set (cloud) of multi-dimensional points. Most often, these are points in 3D that have X, Y, and Z coordinates. Depending on the technology used to capture the points, they may also have a color parameter. Point Cloud Library contains its own data format for point clouds - PCD (Point Cloud Data).

2.1.3 PCL Modules

The library consists of several smaller sub-libraries (modules), each covering a certain area of problem solutions

- **filters** - mechanisms for removing outliers, noise reduction, and filtering 3D point cloud data.
- **features** - data structures and mechanisms for estimating 3D features from point cloud data. 3D features are representations of points that describe certain geometric patterns depending on information from the surroundings of the given point. This selected space around the point is referred to as the K-neighborhood.
- **keypoints** - algorithms for detection of keypoints in the point cloud. Keypoints are points that are distinctive and stable in the dataset and can thus be easily identified using a precisely defined criterion. Keypoints function as a compact but reliable representation of a given point cloud.

- **registration** - a range of point cloud registration algorithms for both organized and unorganized datasets. Registration is the joining of several datasets into one consistent model. For this, it is necessary to determine the corresponding points in individual datasets and find a transformation that minimizes the distance between these points.
- **kdtree** - kd-tree data structure using FLANN (Fast Library for Approximate Nearest Neighbors[ML09]), enables fast nearest neighbor searches.
- **octree** - methods for creating a tree structure where each internal node has exactly eight children[Wik24b], routines for nearest neighbor search.
- **segmentation** - algorithms for point cloud segmentation into individual clusters for subsequent processing.
- **sample_consensus** - Sample Consensus (SAC) methods (e.g., RANSAC, model definitions for planes, cylinders, lines, etc., determination of individual models and their parameters.
- **surface** - reconstruction of original surfaces from 3D scans such as meshes, hulls, or surfaces with normal vectors(normals).
- **recognition** - algorithms for object recognition applications.
- **io** - classes and functions for reading/writing files and capturing point clouds using scanning devices.
- **visualization** - allows quick visualization of the results of various algorithms when working with point clouds.

Chapter 3

Tree Trunk Detection

The aim of this thesis is to design algorithms that allow UAVs to detect trees, their trunks, and their branches. This chapter describes the detection of tree trunks in a multi-tree environment. The detection of individual branches of a given tree is analyzed in Chapter 4.

3.1 Algorithm Outline

To successfully segment individual tree trunks in a point cloud, it is crucial to pre-process the data. That is done in several steps. After loading the input data, it is necessary to remove points that make up the ground and then calculate the normals for use in other algorithms. Subsequently, Region Growing Segmentation and Cylinder Segmentation are performed. Here is an outline of the necessary steps.

1. Load the input point cloud.
2. Estimate normal vectors of the points.
3. Using Planar Segmentation, remove all points that make up the ground.
4. Recalculate normal vectors of the remaining points.
5. Start Region Growing Segmentation to divide the point cloud into multiple clusters.
6. Start Cylinder Segmentation on individual clusters to find tree trunks.
7. Save all identified cylinders (tree trunks) into one PCD file.

3.2 Loading the Input Point Cloud

The function *pcl::io::loadPCDFile<pcl::PointXYZ>* from the *io* module of PCL loads and prepares the point cloud for further handling in the program.

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new
    pcl::PointCloud<pcl::PointXYZ>);

pcl::io::loadPCDFile<pcl::PointXYZ>("smalles_2a_16517381
    87092102_downsampled2.pcd", *cloud);
```

Listing 3.1: Load PCD

This loaded the relevant PCD file into *pcl::PointCloud<pcl::PointXYZ>::Ptr cloud*. The point cloud has already been downsampled in advance to ensure faster data manipulation.

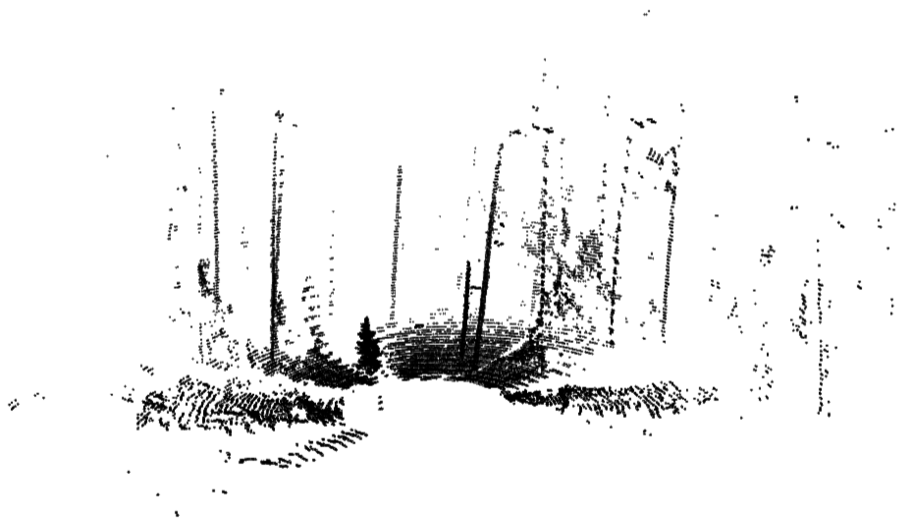


Figure 3.1: Loaded Input Cloud

3.3 Surface Normal Estimation

Surface normal estimation provides important information regarding the orientation of individual surfaces in the scan. The output normal vectors[Wei] are required for the proper functionality of subsequent segmentation algorithms.

```
pcl::NormalEstimation<PointT, pcl::Normal> ne;
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals(new
    pcl::PointCloud<pcl::Normal>);
pcl::search::KdTree<PointT>::Ptr tree(new
    pcl::search::KdTree<PointT>());

ne.setSearchMethod(tree);
ne.setInputCloud(cloud);
ne.setKSearch(10);
ne.compute(*cloud_normals);
```

Listing 3.2: Surface Normal Estimation

Estimation of the normals is performed by the *pcl::NormalEstimation* class. It requires the *pcl::search::KdTree* class to find the k-nearest neighbors of a given point. It is, therefore, set as the search method for *pcl::NormalEstimation*. The input cloud is set as the point cloud loaded in the previous step. Ten nearest neighbors of a given point are selected for the calculation. The calculated normal vectors are then stored in *pcl::PointCloud<pcl::Normal>::Ptr cloud_normals*, meaning each vector in *cloud_normals* corresponds to a point in the original input cloud.

3.4 Ground Removal

The newly calculated normals help remove points that represent the ground in the forest. Removing the ground prior to any segmentation has shown beneficial for reducing the input data size and improving segmentation performance[KHW16][Dou+11].

3.4.1 Planar Segmentation

Using enough points, a model of a plane is created. Points lying directly on the plane and points at a certain distance from it are then filtered and removed from the original point cloud, only leaving points that could theoretically represent tree trunks in the point cloud. This greatly improves the efficiency of algorithms used later, as fewer points generally mean a shorter computation time.

```
pcl::SACSegmentationFromNormals<PointT, pcl::Normal> seg;
pcl::PointIndices::Ptr inliers_plane(new
    pcl::PointIndices);
pcl::ModelCoefficients::Ptr coefficients_plane(new
    pcl::ModelCoefficients);

seg.setOptimizeCoefficients(true);
seg.setModelType(pcl::SACMODEL_NORMAL_PLANE);
seg.setMethodType(pcl::SAC_RANSAC);
seg.setMaxIterations(100);
seg.setDistanceThreshold(0.6);
seg.setInputCloud(cloud);
seg.setInputNormals(cloud_normals);
seg.segment(*inliers_plane, *coefficients_plane);
```

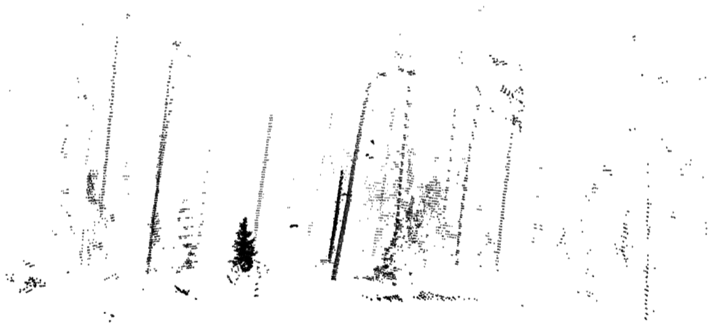
Listing 3.3: Planar Segmentation

The *pcl::SACSegmentationFromNormals* class implements Sample Consensus methods that use normals to estimate the parameters of various geometrical models and also holds templates for models like planes, cylinders, or lines. The class *pcl::PointIndices* stores the indices of points that make up a given plane. The class *pcl::ModelCoefficients* represents a geometrical model using relevant coefficients. In our case, the model being estimated is *pcl::SACMODEL_NORMAL_PLANE*, and the estimation is done by the RANSAC (Random Sample Consensus) method. The number of iterations of the algorithm is limited to 100 to not slow the program down unnecessarily. The command *seg.setDistanceThreshold(0.6)* specifies the distance from the model of the plane to which points are still marked as part of the model; these points are called inliers. Finally, *seg.segment(*inliers_plane, *coefficients_plane)* performs the actual segmentation of the plane. It stores the indices of the points that belong to the model in *inliers_plane* and the planar coefficients in *coeffcients_plane*.

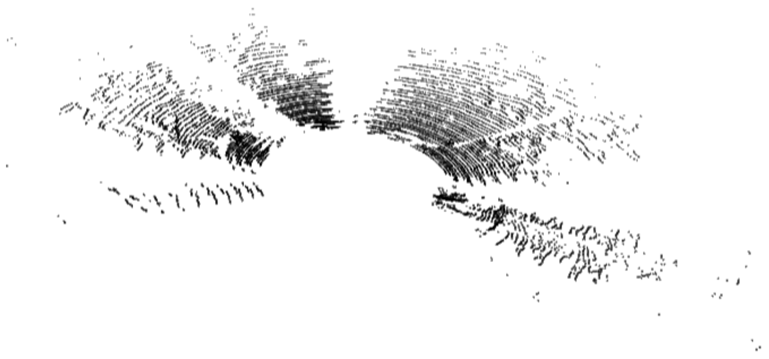
```
pcl::ExtractIndices<PointT> extract;
pcl::PointCloud<PointT>::Ptr cloud_filtered(new
    pcl::PointCloud<PointT>);

extract.setInputCloud(cloud);
extract.setIndices(inliers_plane);
extract.setNegative(true);
extract.filter(*cloud_filtered);
```

Listing 3.4: Plane Extraction

Using the indices in *inliers_plane*, points that form the ground are selected in the original cloud and removed. This functionality is ensured by the *pcl::ExtractIndices* class. By setting *extract.setIndices(inliers_plane)* and *extract.setNegative(true)* at the same time, all points, except those in *inliers_plane* (i.e., the points we want to keep), are selected. The *extract.filter(*cloud_filtered)* method then writes these points to a new point cloud on which Region Growing Segmentation will be used.



(a) : Points Left After Planar Segmentation



(b) : Points Classified as Ground

Figure 3.2: Ground Removal Result

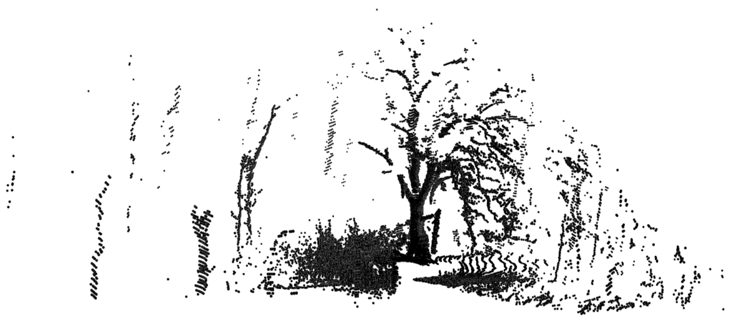
Figure 3.2a is made up of points representing tree trunks and bushes. Points that were removed using Planar Segmentation are in Figure 3.2b.

Testing has shown that it is more convenient to recalculate the estimated normal vectors before continuing onto Region Growing Segmentation to achieve improved results. The process looks analogous to the one shown in Listing 3.2, with the difference that it is done on the already filtered cloud. New normals are stored in *pcl::PointCloud<pcl::Normal>::Ptr filtered_cloud_normals.*

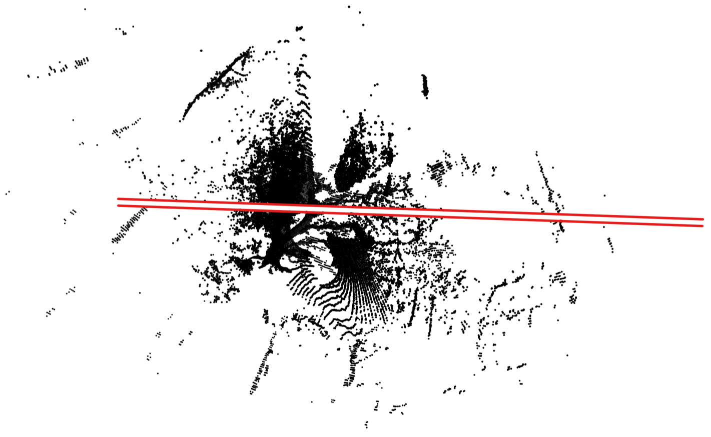
3.4.2 Pass-Through Filter

Correct data pre-processing has shown to be challenging. While Planar Segmentation produced satisfactory results on the first dataset of multiple trees, it failed terribly when implemented on the dataset with one sizable tree. The problem is caused by the fact that the profile of the ground near the tree is uneven and irregular. Naturally, a plane cannot be fitted through such points. Therefore, the plane that the Planar Segmentation (Listing 3.3) algorithm finds is, in a majority of cases, a random plane that enough points fall onto by chance. This means that Planar Segmentation yields wrong and, thus, unsatisfactory results.

Figure 3.3a shows a view of the tree after unsuccessful Planar Segmentation. The mistake becomes apparent when looking at the scan from below; see Figure 3.3b. A strip of missing points (marked by the red lines for illustration) is evident in the scan. That is the result of removing the fitted plane along with a set margin around it.



(a) : Dataset After Incorrect Planar Segmentation



(b) : Red Lines Showing the Incorrectly Selected Plane

Figure 3.3: Planar Segmentation, Incorrect Result

The simple yet effective solution to this problem is to remove all points whose Z-coordinate is less than some threshold depending on the loaded point cloud. That creates a point cloud where the tree is separated from the rest of the foliage near it, enabling the Region Growing Algorithm to effortlessly segment the whole tree with all its branches. Thanks to the fact that the origin of the coordinate system is the drone itself and the fact that the drone remained mostly stable (there is some tilt present, however, so the Z-axis does not exactly represent a direction perpendicular to the ground, as one might think) throughout the scan around the tree, this threshold is similar for all partial scans. The filtering itself is executed by a pass-through filter. This only leaves points that satisfy a given constraint - a value of the Z-coordinate in this case.

```
pcl::PassThrough<pcl::PointXYZ> passThrough;

passThrough.setInputCloud(cloud);
passThrough.setFilterFieldName("z");
passThrough.setFilterLimits(-2.8, 100.0);
passThrough.filter(*cloud_filtered);
```

Listing 3.5: Pass Through Filter

For this specific dataset, the cutoff threshold hovered around $z = -3$.

Figure 3.4 displays the new, simplified approach. While not all ground points are removed, a large enough gap forms between the tree trunk and the adjacent ground allowing for a successful subsequent Region Growing Segmentation. As previously stated, this method generates sufficient results, unlike the Planar Segmentation method.

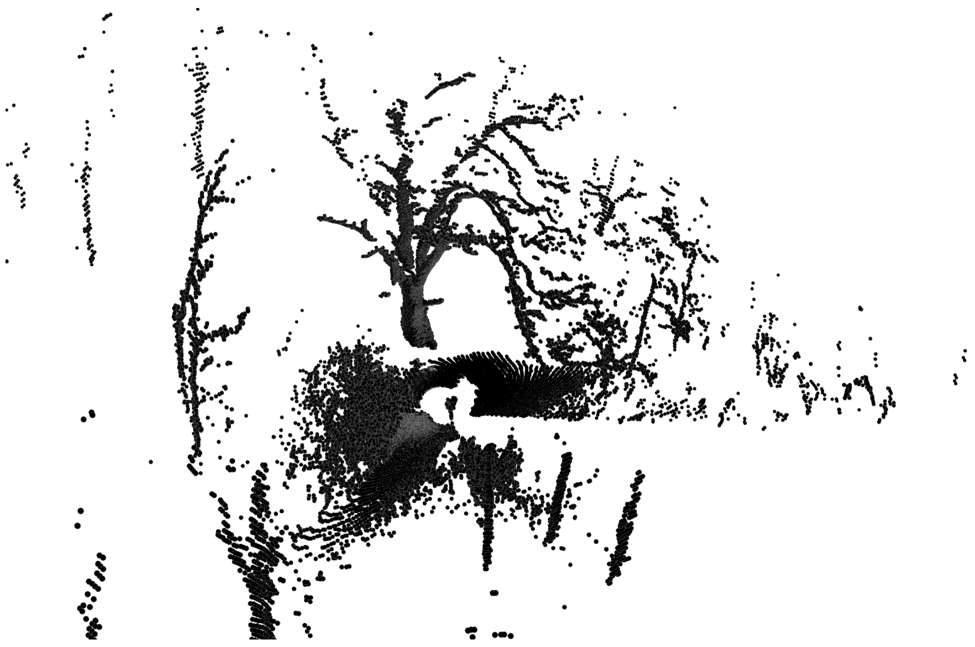


Figure 3.4: Ground Removal, Correct Result

3.5 Region Growing Segmentation

With the point cloud now being rid of the points belonging to the ground plane, the Region Growing Segmentation algorithm[RHV06] implemented in the *pcl::RegionGrowing* class can be run. It divides the points into multiple smaller parts - clusters. The algorithm takes multiple parameters - minimum cluster size, maximum cluster size, a method for finding nearest neighbors (the same KdTree as in Listing 3.2), input data, normals corresponding to the input data, and a curvature and smoothness threshold. After running *reg.extract(clusters)*, individual clusters are then saved as indices to *std::vector<pcl::PointIndices> clusters*.

```
pcl::RegionGrowing<pcl::PointXYZ, pcl::Normal> reg;

reg.setMinClusterSize(50);
reg.setMaxClusterSize(1000000);
reg.setSearchMethod(tree);
reg.setNumberOfNeighbours(50);
reg.setInputCloud(cloud_filtered);
reg.setInputNormals(filtered_cloud_normals);
reg.setSmoothnessThreshold(30.0 / 180.0 * M_PI);
reg.setCurvatureThreshold(0.6);

std::vector<pcl::PointIndices> clusters;
reg.extract(clusters);
```

Listing 3.6: Region Growing Segmentation

The exact values for the smoothness and curvature thresholds were determined by trial and error and greatly depend on the dataset, its features, and the expected output. As previously stated, the Region Growing Segmentation algorithm writes the indices to an array of indices (*pcl::PointIndices*); however, pointers to the indices (*pcl::PointIndices::Ptr*) are required when working with other algorithms. Therefore, performing any necessary conversions before continuing any further is essential.

The result of Region Growing Segmentation can be visualized using colors - each color represents a different cluster. Figure 3.5 shows what that might look like.

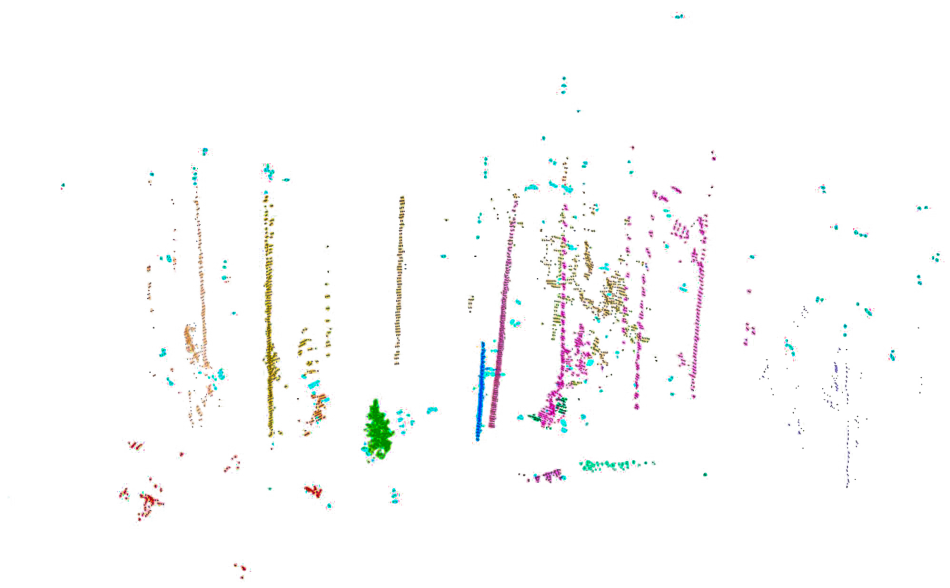


Figure 3.5: Colored Clusters

3.6 Cylinder Segmentation

With the point cloud divided into clusters, it is possible to apply the Cylinder Segmentation algorithm to each cluster separately. This, again, uses the *pcl::SACSegmentationFromNormals* class, just like Planar Segmentation (Listing 3.3). The approximation of the *pcl::SACMODEL_CYLINDER* model is also done by the RANSAC method. In this case, several parameters are set - the weight of the normal vectors, the maximum number of iterations of the algorithm, the maximum distance of a point from the current cylinder model to still be considered part of the cylinder, and the maximum and minimum radius of the cylinder. In this context, a cylinder represents a tree trunk.

```
pcl::SACSegmentationFromNormals<PointT, pcl::Normal>
    cyl_seg;
pcl::PointIndices::Ptr inliers_cylinder(new
    pcl::PointIndices);
pcl::ModelCoefficients::Ptr coefficients_cylinder(new
    pcl::ModelCoefficients);

cyl_seg.setOptimizeCoefficients(true);
cyl_seg.setModelType(pcl::SACMODEL_CYLINDER);
cyl_seg.setMethodType(pcl::SAC_RANSAC);
cyl_seg.setNormalDistanceWeight(0.1);
cyl_seg.setMaxIterations(10000);
cyl_seg.setDistanceThreshold(0.4);
cyl_seg.setRadiusLimits(0.05, 0.25);
```

Listing 3.7: Cylinder Segmentation

After the initial setup, Cylinder Segmentation is run on each cluster and its corresponding normal vectors. When a cylinder is identified and segmented, all points that fall onto its model are saved as an individual PCD file (one PCD file, one tree trunk) and then removed from the current cluster (along with the relevant normals). This process is repeated until the current cluster is empty, i.e., all the points have been removed from it. This way, as many tree trunks as possible are found in each cluster. To ensure a higher chance of a segmented cylinder actually representing a tree trunk, only cylinders containing more than 90 points are considered valid and saved as a PCD file. Figure 3.6 shows all cylinders put together.

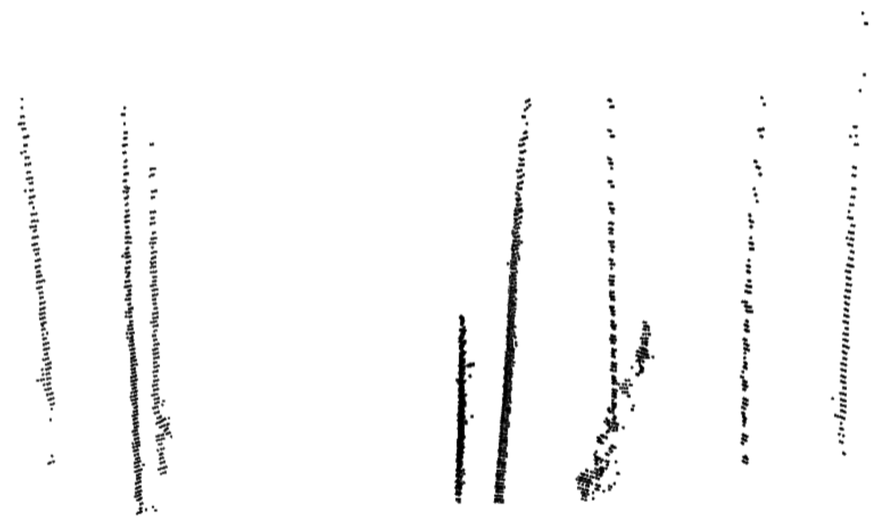


Figure 3.6: All Cylinders

Chapter 4

Trunk and Main Branches Detection

Detecting the main branches of a particular tree is a natural step in the current process. It has proven an order of magnitude more difficult compared to simple tree trunk detection as it presented a number of new challenges. The sequence of the steps is more complex and differs from the one in Section 3.1. The revised outline looks like this:

1. Load the input point cloud.
2. Estimate normal vectors of the points, as described in Section 3.3.
3. Remove ground using the filter from Subsection 3.4.2.
4. Recalculate normal vectors of the remaining points.
5. Start Region Growing Segmentation to divide the point cloud into multiple clusters (updated in Section 4.1).
6. Select the cluster with the complete tree in it.
7. Perform Cylinder Segmentation to obtain cylindrical models of the trunk and main branches.
8. Identify adjacent branches.
9. Create a graph of the tree using the axes of the cylinders.
10. Test said algorithm on multiple scans fused together using Iterative Closest Point.

4.1 Whole Tree Region Growing Segmentation

Now that the dataset has been processed accordingly, the Surface Normal Estimation and the Region Growing Algorithm can be run. The implementation is identical to the ones in Listing 3.2 and Listing 3.6, respectively. The output of this algorithm is again an array of clusters. The cluster with the whole tree in it is then selected. This particular point cloud is set as the input cloud for the next step - Cylinder Segmentation.

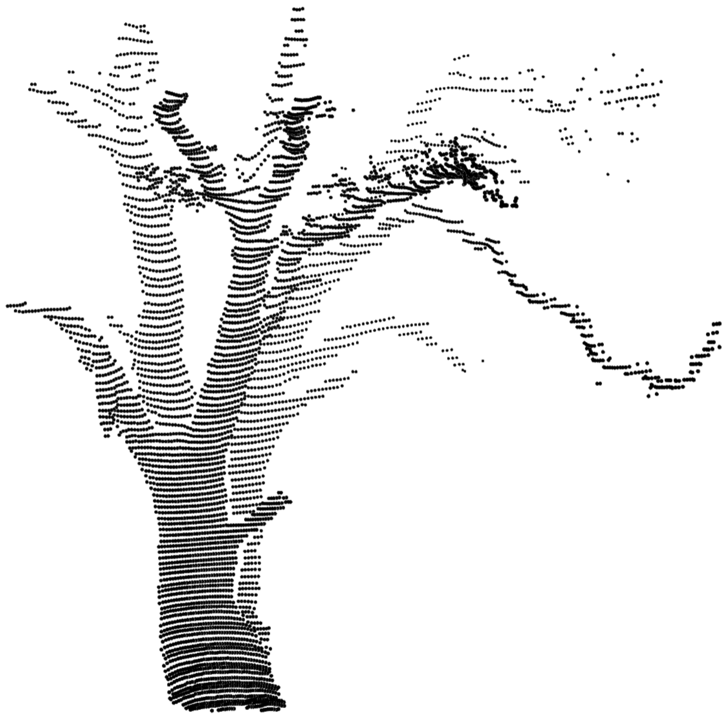


Figure 4.1: Example of a Selected Cluster

4.2 Tree Cluster Cylinder Segmentation

At first glance, it might seem that the trunk and main branches segmentation resembles the approach used in Figure 3.6. That thought is, however, mostly inaccurate. Instead of segmenting multiple clusters, the goal is to segment just one cluster repeatedly to extract as much data as possible. As this part of the program is rather complex, a C++ struct is created to hold data relevant to each cylinder.

```cpp
typedef struct My_Cylinder
{
    int id;
    pcl::PointIndices::Ptr originalIndices;
    pcl::ModelCoefficients::Ptr coefficients;
    pcl::SampleConsensusModelCylinder<PointT,
   pcl::Normal>::Ptr cylinder_model;
    My_OBB *cyl_obb;
    std::vector<My_Cylinder *> overlapping_cylinders;
    pcl::PointCloud<pcl::PointXYZ>::Ptr projectedCyl;

} My_Cylinder;
```

Listing 4.1: MyCylinder Struct

Each cylinder then holds the following information:

- **id** - an identifier unique to the cylinder.
- **orignalIndices** - indices of points as they were in the original cloud at the time of loading.
- **coefficients** - coefficients of the cylinder model - a vector of the axis, a point on that axis, and the diameter of the cylinder.
- **cylinder_model** - stores the points of the cylinder and provides useful methods (more in Subsection 4.2.3).
- **cyl_obb** - a custom C++ struct for an oriented bounding box of the cylinder (more in Subsection 4.3.1).
- **overlapping_cylinders** - an array of cylinders that overlap with the current one, i.e., are adjacent.
- **projectedCyl** - a point cloud of all points (that fall onto the model of the cylinder) projected onto the cylinder's axis.

First, Cylinder Segmentation is performed on the whole point cloud, meaning a cylinder can be found in any part of the scan. If, at this point, the algorithm does not find a cylinder, the program stops, as there are either no more cylinders remaining or the points do not represent a cylinder. However, if a cylinder exists, it is further processed.

4.2.1 Euclidean Cluster Extraction

Due to the shape of the tree and the fact that the cylinder model is not constrained in either direction of its axis, the model often intersects with multiple branches or a trunk and a branch. This creates a cylinder with multiple isolated sets of points, as seen in Figure 4.2. This behavior is undesirable as the intention is only to find points from one cylinder.

The solution to this problem is to use Euclidean Cluster Extraction. The algorithm works as described in [Rus09] :

1. create a kd-tree representation for the input point cloud dataset P;
2. set up an empty list of clusters C and a queue of the points that need to be checked Q;
3. then for every point $p_i \in P$, perform the following steps:
   - add p_i to the current queue Q;
   - for every point $p_j \in Q$ do:
     - search for the set P_i^k of point neighbors of p_i in a sphere with radius $r < d_{th}$.

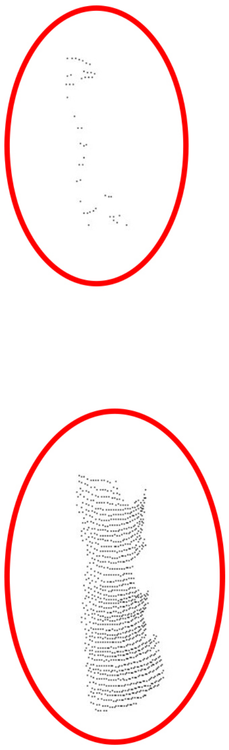


Figure 4.2: Isolated Sets of Points Belonging to One Cylinder

    - for every neighbor $p_i^k \in P_i^k$, check if the point has already been processed, and if not, add it to Q;
  - when the list of all points in Q has been processed, add Q to the list of clusters C, and reset Q to an empty list;

4. the algorithm terminates when all points $p_i \in P$ have been processed and are now part of the list of point clusters C.

Using this algorithm, it is possible to select the biggest cluster in the current cylinder. This cluster then represents the cylinder. The remaining points are removed from the current cylinder but remain in the point cloud. In all the steps where points are removed or reassigned, the same operations are performed on the corresponding normals, so no discrepancies happen.

4.2.2 Cylinder Radius Optimization

The next step in the segmentation is to make the cylinder as precise as possible to best represent the actual trunk/branch. The idea is to incrementally decrease the radius of the segmented cylinder while keeping a certain number of points from the original cylinder. Half of the points are a reasonable threshold for this dataset. This procedure is needed because the algorithm seems to favor cylinders with greater radii, which, unfortunately, means less accurate results. The following algorithm is at least partially able to mitigate this fact. At the start of the cylinder segmentation, the lower and upper bounds for the radii are set to 0.025 and 0.9, respectively. This is to include all possible cylinders.

The algorithm then works as follows:

1. Start initial Cylinder Segmentation to obtain a cylinder
2. Perform Euclidean Cluster Extraction to find the biggest cluster
3. Set this cluster as the current cylinder
4. While $newMinRadius > thA$ and $curPointCount > thB$
    - Perform Cylinder Segmentation on the current cylinder with

    $$newMaxRadius = currentCylinderRadius - 0.001$$

    $$newMinRadius = newMaxRadius * radiusRatio$$

    - If the newly segmented cylinder contains at least half of the points of the cylinder from 1, set it as the current cylinder and go to 4.
5. Repeat for all cylinders

To specify, $currentCylinderRadius$ is the radius of the last found cylinder, $curPointCount$ is the number of points in the last found cylinder, $radiusRatio$ is some preset ratio by which the radius is made smaller(4/5 in this case), and thA and thB are thresholds chosen with respect to the current dataset.

This algorithm produces a more accurate output because the found cylinders better resemble the actual parts of the tree. It generally also detects more cylinders, as shown in Figure 4.3.

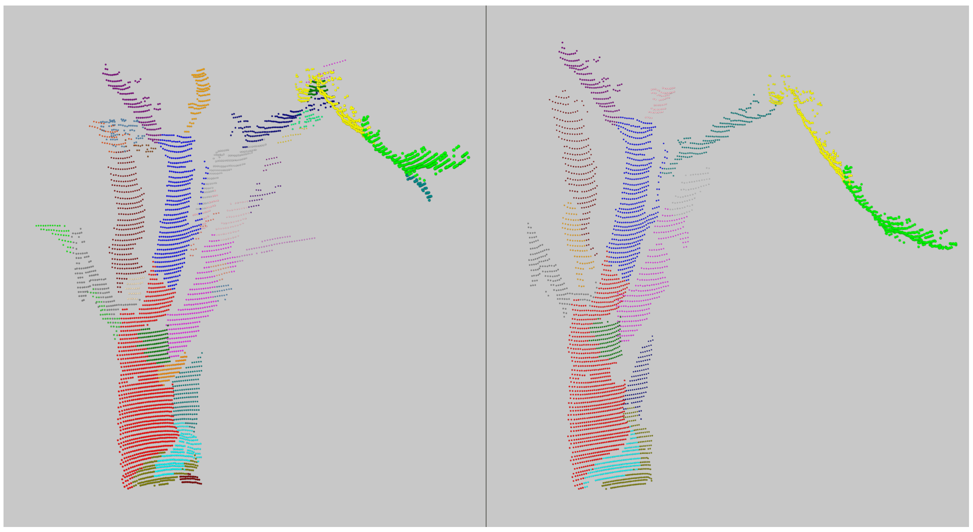


Figure 4.3: Cylinder Radius Optimization (left) vs No Optimization (right)

4.2.3 Adding Points to Optimized Cylinders

In most cases, the optimal cylinder radius is different from the original radius. At this point, it is advantageous to reevaluate and check whether any points from the original cloud fall onto the new model of the cylinder. These points can be added to the current cylinder cloud, thus creating a more complex cylinder. The PCL class *pcl::SampleConsensusModelCylinder<PointT, pcl::Normal>* implements a method *doSamplesVerifyModel* which verifies whether a point (sample) verifies a given model. Each point from the original cloud is then checked, and if it fulfills the given criteria, it is added to the current cylinder.

Due to the fact mentioned in Subsection 4.2.1, it is imperative to rerun Euclidean Cluster Extraction to eliminate potential gaps in the cylinder.

After concluding this step, the points belonging to the cylinder are removed from the original cloud, and the finalized cylinder is ready to be used further; no other alterations are necessary.

The above-mentioned steps:

- Cylinder Segmentation
- Euclidean Cluster Extraction
- Cylinder Radius Optimization
- Adding Points
- Final Euclidean Cluster Extraction

are repeated until no new cylinders are found in the original cloud.

4.3 Adjacent Branches Identification

At this point, the trunk and the branches are segmented. However, there is no information on how all these branches are connected, which is crucial to creating a tree graph/model. As mentioned previously, each *My_Cylinder* struct has an array called *overlapping_cylinders* to store all adjacent cylinders. In our case, cylinders are considered adjacent when they are attached to each other on the real tree. Simply put, two adjacent cylinders are just two connected parts of a branch. There are many ways to determine which two cylinders are adjacent, each with its pros and cons. Next, three approaches are described along with the results they produced on the given dataset.

4.3.1 Oriented Bounding Box

As defined in [GML00], an oriented bounding box (OBB) is an arbitrarily oriented rectanguloid. An axisaligned bounding box (AABB) is a rectanguloid whose faces are aligned with the coordinate axes of its parent coordinate system. Whereas an AABB can be represented with just minimum and maximum extents along each axis, an OBB representation must encode not only position and widths but also orientation. The advantage OBBs have over AABBs as bounding volumes is that they can bind their enclosed geometry more tightly.

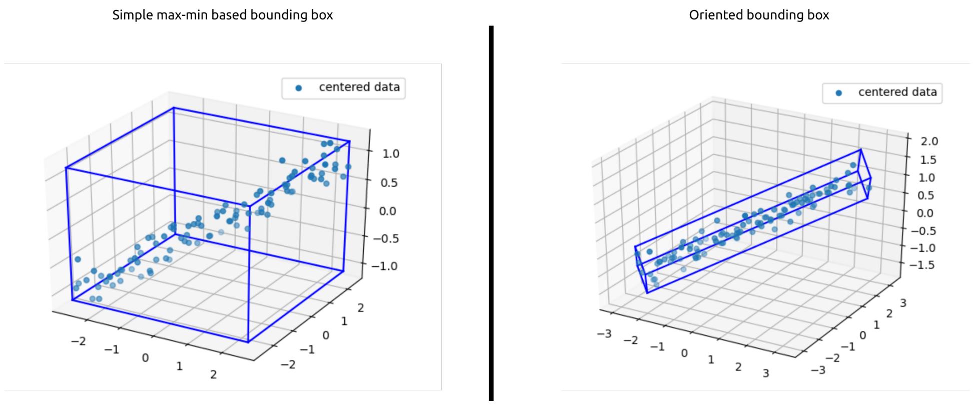


Figure 4.4: A Comparison of AABB and OBB of a Point Cloud[Kak21]

In PCL, OBBs can be calculated with *pcl::computeCentroidAndOBB*, which returns the centroid, the center, OBB dimensions, and a rotational matrix of the OBB.

The idea behind implementing this approach is to create an Oriented Bounding Box for each cylinder and then, using the Separating Axis Theorem[GML00], find OBBs that overlap. Those OBBs can, therefore, be considered adjacent branches. As per [GML00], an axis **n** is a separating axis of two point sets A and B if and only the images of A and B under axial projection onto **n** are disjoint. The separating axis theorem states that two polytopes, A and B, are disjoint if and only if there exists a separating axis which is either perpendicular to a face of one of the polytopes or is perpendicular to an edge taken from each.

In a three-dimensional space, 15 separating axes exist. Three represent the face normals of box A, three represent the face normals of box B, and the nine remaining axes are calculated as all possible combinations of cross products of the face normals of box A and box B. The axes are shown in Table 4.1. The lower index always corresponds to one of the face normals of the boxes[Huy08][Ebe99].

Axis Nr.	Axis Label
1	A_0
2	A_1
3	A_2
4	B_0
5	B_1
6	B_2
7	$A_0 \times B_0$
8	$A_0 \times B_1$
9	$A_0 \times B_2$
10	$A_1 \times B_0$
11	$A_1 \times B_1$
12	$A_1 \times B_2$
13	$A_2 \times B_0$
14	$A_2 \times B_1$
15	$A_2 \times B_2$

Table 4.1: Possible Separating Axes

As stated above, to declare two boxes as not overlapping, at least one separating axis must exist. Therefore, the projections of box A and box B onto the specified axis must not overlap at least in one of the cases. When checking for overlap, the following applies (according to [GML00]):

Two intervals are disjoint if and only if the separation of their midpoints is greater than the sum of their half-widths (radii). The relevant comparison is $s > r_A + r_B$. $I_n(A)$ is the projection of box A onto an axis **n** and $I_n(B)$ is the projection of box B onto an axis **n** as visible in Figure 4.5.

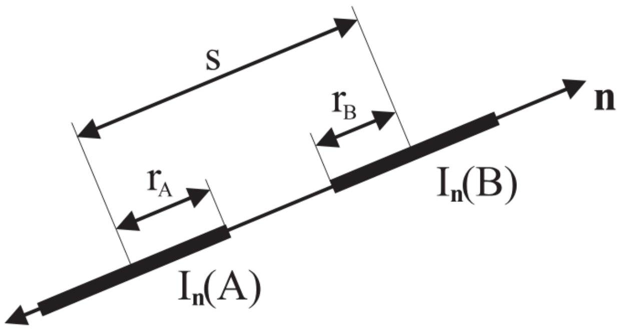


Figure 4.5: Checking for Disjoint Intervals[GML00]

The way the projections $I_n(A)$ and $I_n(B)$ are obtained is explained in [Huy08]:

Suppose box A has the following properties:

- P_A: coordinate position of the center of A
- $\mathbf{A}_x$: unit vector representing the x-axis of A

- $\mathbf{A}_y$: unit vector representing the y-axis of A
- $\mathbf{A}_z$: unit vector representing the z-axis of A
- W_A: half width of A (corresponds with the local x-axis of A)
- H_A: half height of A (corresponds with the local y-axis of A)
- D_A: half depth of A (corresponds with the local z-axis of A)

The projection of half of box A onto axis L is given by the equation:

$$\begin{aligned}\frac{1}{2}|\text{Proj}(\text{BoxA})| &= |\text{Proj}(W_A\mathbf{A}_x)| + |\text{Proj}(H_A\mathbf{A}_y)| + |\text{Proj}(D_A\mathbf{A}_z)| \\ &= |(W_A\mathbf{A}_x)\cdot\mathbf{L}| + |(H_A\mathbf{A}_y)\cdot\mathbf{L}| + |(D_A\mathbf{A}_z)\cdot\mathbf{L}|\end{aligned}$$

The process for box B is analogous. Those projections are then equal to r_A and r_B, respectively. The distance between the centers of the OBBs is $s = P_B - P_A$.

If an OBB is deemed to be overlapping with the OBB of the current cylinder, the corresponding cylinder is added to the array of overlapping cylinders. In Figure 4.6, the pink points belong to the current cylinder, whereas the black points are from cylinders whose OBBs overlap with the current cylinder's OBB. The OBBs themselves are displayed as cuboids.

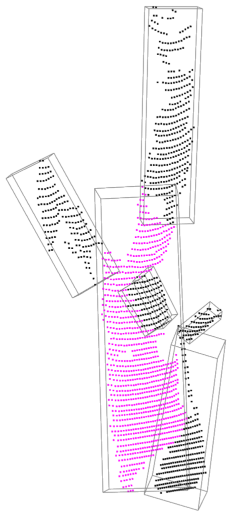


Figure 4.6: Example of Overlapping OBBs

While this approach is based on a robust theory, it is unsuitable for this application. The main problem is that due to the irregular shapes that the points in the cylinders make, the bounding boxes sometimes overlap when the cylinders are not adjacent (as demonstrated in Figure 4.7), or, on the other hand, do not overlap, even though the cylinders are right next to each other. While this does not necessarily render it unusable for this purpose, another approach has shown superior results.

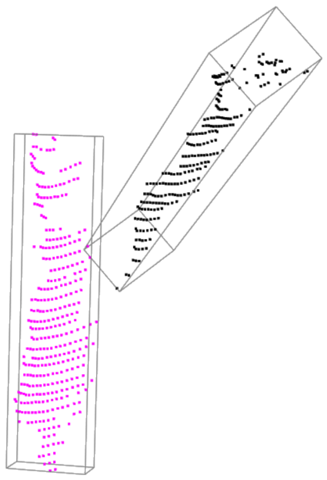


Figure 4.7: An Incorrect Evaluation of Overlaps

4.3.2 Nearest Neighbor Search

A different method of classifying adjacent branches is to find the shortest distance between two points of the cylinders and then compare that value to a previously set threshold. If that comparison holds true, the two cylinders can be considered adjacent.

Suppose we have cylinder A, whose adjacent cylinders we want to find. Take some cylinder B, and for each point in A, find its nearest neighbor in points from B, and if the distance between them is smaller than the shortest distance found until now, store it as the shortest distance.

Repeat this process for cylinders C, D, etc. When finished, take cylinder B and start the process again, finding the nearest neighbor in cylinders C, D, E, etc. Repeat for all pairs of cylinders.

Nearest neighbors are found with *pcl::KdTreeFLANN<pcl::PointXYZ>::Ptr* and its method *nearestKSearch*, where K is set to one, thus only finding the one nearest neighbor.

Each cylinder should then have one value for each of the remaining cylinders, denoting the shortest distance between any two points on the two cylinders.

This technique is simple but computationally demanding with respect to the number of cylinders, as it needs to find the nearest neighbor for all points in a cylinder. The main problem, however, is when two branches are relatively close to each other at just one point in space, they satisfy the threshold condition, even though they are not connected. Fine-tuning the threshold value is complicated as well.

4.3.3 Endpoint Nearest Neighbor

The last approach also utilizes Nearest Neighbor Search but in a different way. First, each cylinder is projected onto its axis, creating a sort of line segment. Then, thanks to the cylinder coefficients, the endpoints of the line segment can be calculated.

As is given by the parametric equation of a line, any point on a line can be described as mentioned in [Wik24a]:

$$\begin{aligned} x &= x_0 + at \\ y &= y_0 + bt \\ z &= z_0 + ct \end{aligned}$$

where:

- (x, y, z) is a point on the line with an independent variable t which ranges over the real numbers.
- (x_0, y_0, z_0) is a fixed point on the line.
- a, b, and c are related to the slope of the line, such that the direction vector (a, b, c) is parallel to the line.

To calculate t:

$$t = (x - x_0)/a = (y - y_0)/b = (z - z_0)/c$$

If t is calculated for each projected point, we can find the lowest and highest value of t. Those values belong to the points on either end of the line segment, i.e., the points that are the furthest away from the base point in each direction.

The next step is to take the newly obtained endpoints and find their respective nearest neighbors in the points on the axis of some other cylinder. Note the difference: in Subsection 4.3.2, the nearest neighbor is found for each point; in this case, however, the nearest neighbor is found only for the two endpoints. The distances between the endpoints and their nearest neighbors can then be compared to a threshold, deciding whether the two cylinders from which those points come are close enough to be considered adjacent. The threshold is set as **1.2r** for the single scan and **1r** for the merged scans (see Section 6.2), where **r** is the radius of the current cylinder. In the merged scan, the model becomes too crowded with the connections between branches due to the higher number of cylinders, which is why the threshold is set lower. Those are the values that produce the results in Figures 5.4 and 5.5, and Figures 6.3 and 6.4, respectively. As can be seen, the threshold serves as a way to set how much detail (i.e., the number of connections made) is in the final model. This step is repeated for all cylinders.

This method proved to be the most reliable out of those mentioned previously. It eliminates the problem of two cylinders being close to each other at just one point (usually close to the middle). It also does not suffer from the shortcomings of OBBs relating to false positives/negatives, as the points are always projected onto a line, and the distance threshold can be adjusted as needed.

Chapter 5

Tree Graph

The previous section describes a method for detecting whether two cylinders are adjacent. This chapter proposes an algorithm to connect the individual segments representing the axes of the adjacent cylinders. The connected cylinders should form a semi-continuous structure that resembles the original tree - a tree graph.

As branches generally differ in shape and size, the relative positions of the cylinders the branches represent are expressed by various scenarios. These possibilities need to be taken into account when trying to connect them. Connecting two branches means inserting new points between the two cylinder axes.

5.1 Cylinder Axes as Skew Lines

Due to the accuracy of the measurements and cylinder translation algorithms, all axes of the following cylinders can be considered skew lines. The interconnection of the two line segments representing the axes of the cylinders depends on the relative positions of the points that represent the distance of these skew lines.

To obtain these points, we take the two cylinder axes, their respective vectors, and points on the axes and create parametric equations of the lines:

$$Cylinder\ axis\ 1 : l_1 = p_1 + t_1\vec{d_1}$$
$$Cylinder\ axis\ 2 : l_2 = p_2 + t_2\vec{d_2}$$

where p is a point on the line and $\vec{d}$ is a vector of direction of a line l.

Next, we must find a vector $\vec{n}$ perpendicular to both lines simultaneously. This is done by calculating the cross product of $\vec{d_1}$ and $\vec{d_2}$[Wik24a]:

$$\vec{n} = \vec{d_1} \times \vec{d_2}$$

As further mentioned in [Wik24a]: The plane formed by the translations of Line 2 along $\vec{n}$ contains the point p_2 and is perpendicular to $\vec{n_2} = \vec{d_2} \times \vec{n}$.

Therefore, the intersecting point of Line 1 with the above-mentioned plane, which is also the point on Line 1 that is nearest to Line 2, is given by

$$c_1 = p_1 + \frac{(p_2 - p_1) \cdot \vec{n_2}}{\vec{d_1} \cdot \vec{n_2}} \vec{d_1}$$

Similarly, the point on Line 2 nearest to Line 1 is given by (where $\vec{n_1} = \vec{d_1} \times \vec{n}$)

$$c_2 = p_2 + \frac{(p_1 - p_2) \cdot \vec{n_1}}{\vec{d_2} \cdot \vec{n_1}} \vec{d_2}$$

Now that the points are calculated, there are three possibilities as to where they are with respect to the endpoints of the cylinder axes.

1. Point c_1 lies **between** the endpoints of cylinder axis 1. Point c_2 lies **between** the endpoints of cylinder axis 2.

2. Point c_1 lies **between** the endpoints of cylinder axis 1. Point c_2 lies **outside** of the endpoints of cylinder axis 2, or vice versa.

3. Point c_1 lies **outside** of the endpoints of cylinder axis 1. Point c_2 lies **outside** of the endpoints of cylinder axis 2.

In case number **1**. the solution is fairly straightforward, points c_1 and c_2 are simply connected with a line as illustrated in Figure 5.1. Note that the sketches are in 2D; they do not represent the situation to its full extent, but they are sufficient for this purpose. The axes are denoted as l_1 and l_2, with the blue segments representing the projected points.

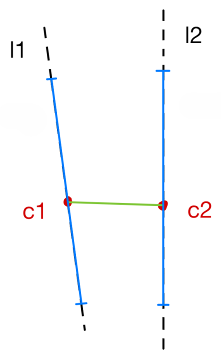
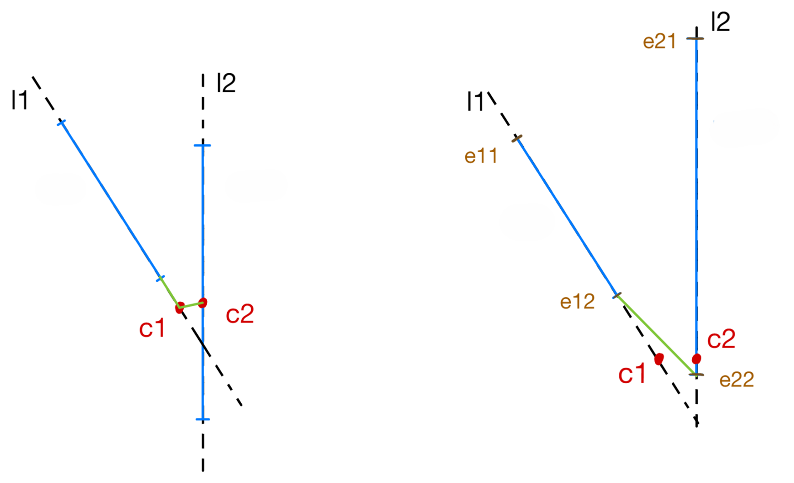


Figure 5.1: Skew Lines: Case 1

In case number 2, there are two options as to what can happen. We will focus on the situation where point c_1 lies **outside** of the endpoints of cylinder axis 1 and point c_2 lies **between** the endpoints of cylinder axis 2. The process, for when the opposite is true, is analogous.

Let us define the distance between c_1 and the endpoint of axis 1 closest to it as $minDist1$ and $distThresh$ as some arbitrary threshold.

If $minDist1 < distThresh$ holds true, c_1 and the endpoint (of axis 1) closest to it are connected. Then, c_1 and c_2 are connected. Simply put, the



(a) : $minDist1 < distThresh$ **(b) :** $minDist1 > distThresh$

Figure 5.2: Skew Lines: Case 2a and 2b

axis of cylinder 1 is extended to point c_1, and then c_1 and c_2 are connected. See Figure 5.2a.

If $minDist1 > distThresh$ is the case, the endpoint (of axis 1) closest to c_1, let us call it e_{12}, is connected to point e_{22}, an endpoint from axis 2 that is closest to e_{12}. See Figure 5.2b.

In case number 3, we extend both axes and then connect c_1 and c_2. That is, take c_1, find the endpoint (of axis 1) closest to it, and connect them. Do the same for c_2 and the endpoint (of axis 2) closest to it. Finally, connect the two closest points, c_1 and c_2.

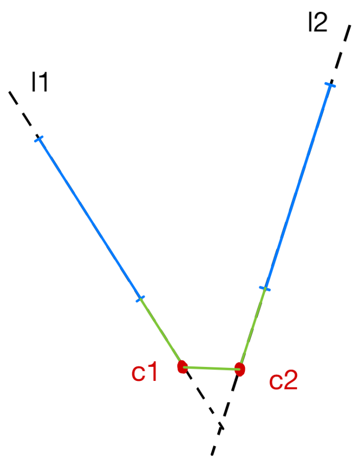


Figure 5.3: Skew Lines: Case 3

Combining these steps produces a semi-continuous model (tree graph) of the tree, as shown in Figure 5.4. Semi-continuous means that the model is made up of multiple clusters of interconnected branches. However, not all clusters are connected. The green lines represent the axes of the cylinders, and the pink lines are segments drawn into the point cloud as a result of connecting individual axes. To better grasp how the model relates to the original scan, see Figure 5.5.

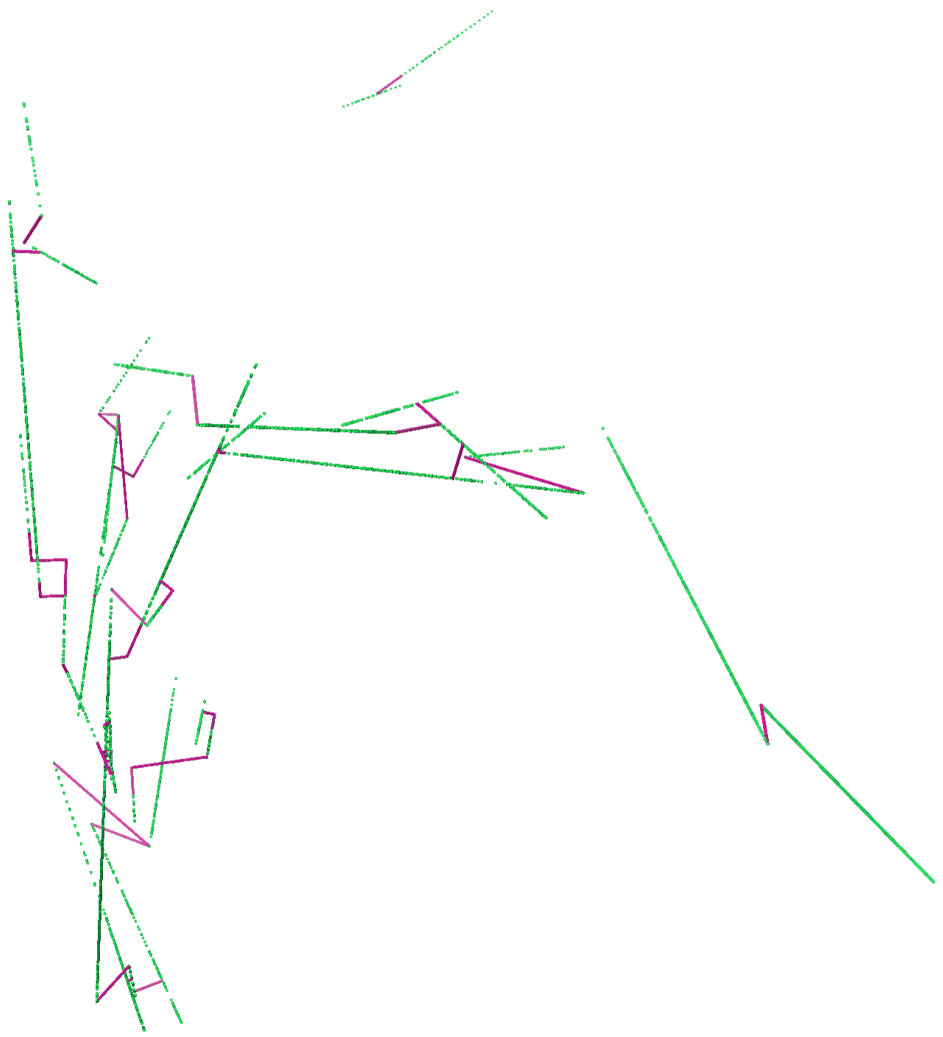


Figure 5.4: Tree Graph

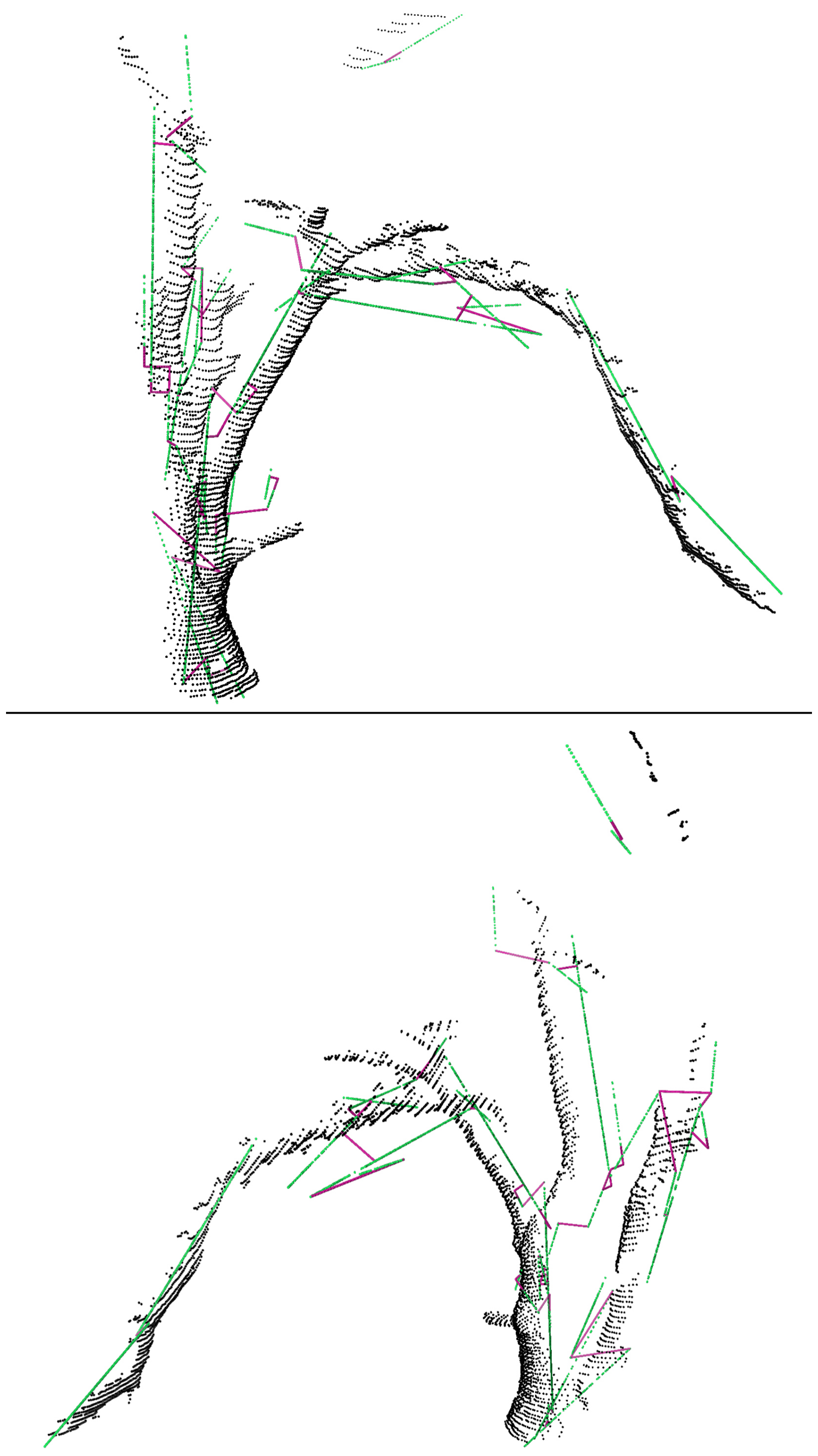


Figure 5.5: Tree Graph with Points

Chapter 6

Fusing Multiple Scans Together

The algorithm described above works as intended on a single scan from one position. In this chapter, we will test whether this algorithm performs comparably on a point cloud merged from multiple scans from different positions. The Iterative Closest Point algorithm seems well suited for this purpose[BM92]. In [Rus09], ICP is described as an iterative descend method that tries to find the optimal transformation between two datasets by minimizing the Euclidean distance error metric between their overlapping areas. ICP uses pairs of nearest 3D points in the source and model set as correspondences and assumes that every point has a corresponding match. ICP has drawbacks, such as being susceptible to local minima, having a small convergence basin, and generally needing a high number of iteration steps until convergence can be reached.

6.1 Input Dataset Transformation

Because ICP considers pairs of points in both clouds, it can become computationally demanding as the size of the dataset increases. Therefore, the dataset must remain relatively small. Consequently, only the segmented tree (as seen in 4.1) is used to merge the point clouds instead of the original dataset. In Table 6.1, the respective sizes (in number of points) of the original scans and final segmented trees are stated. Five scans were chosen for the merging.

Cloud Nr.	Original Size	Final Size	Reduction in %
1	131072	4655	96.45
2	131072	6778	94.83
3	131072	5650	95.69
4	131072	5415	95.87
5	131072	5288	95.97

Table 6.1: Scan Sizes

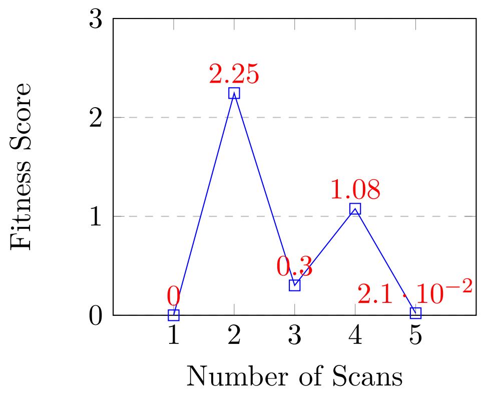
6.2 PCL ICP

Point Cloud Library implements a version of Iterative Closest Point in *pcl::IterativeClosestPoint* based on Singular Value Decomposition and a version *pcl::IterativeClosestPointNonLinear*, which uses Levenberg-Marquardt optimization. Both methods yield similar results, as is evident in Figure 6.1.

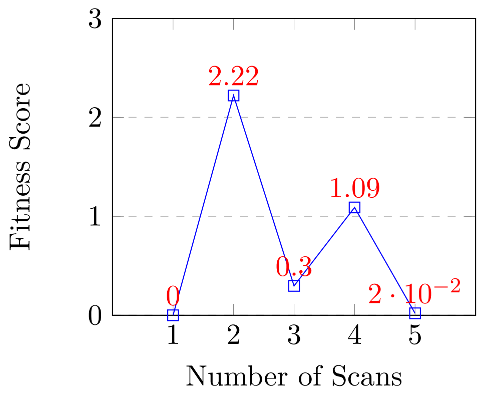
According to [Lib], the algorithm has several termination criteria:

1. Number of iterations has reached the maximum user imposed number of iterations (via setMaximumIterations)
2. The epsilon (difference) between the previous transformation and the current estimated transformation is smaller than a user imposed value (via setTransformationEpsilon)
3. The sum of Euclidean squared errors is smaller than a user defined threshold (via setEuclideanFitnessEpsilon)

The sum of Euclidean squared errors (called *Fitness Score* in PCL) can, therefore, be used to evaluate the performance of the algorithm on a given dataset. The lower the value, the better aligned the scans should be. In Figure 6.1, fitness scores are displayed for 1-5 merged scans. This shows that the best alignment was achieved when combining five scans. The results are, however, partially inaccurate due to the fact that some points are present only in some scans and not in others. Because of that, the fitness scores for when two scans are merged reach relatively large numbers. This is apparent in Figure 6.2, where the green cloud contains a large branch that is not present in the red cloud. All the points in the green branch then make up a large portion of the error, as they are far away from any red points. This artifact diminishes when more scans with that branch are added.



(a) : Non Linear ICP



(b) : Linear ICP

Figure 6.1: Fitness Score Evaluation

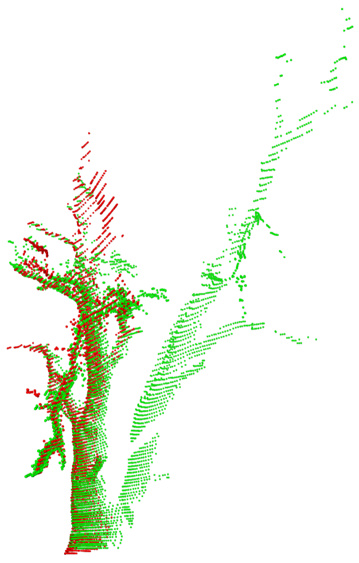


Figure 6.2: An Odd Branch

6.3 Voxel Grid Filtering

When using point clouds that are created by fusing multiple scans together, the number of points rapidly increases. To keep the computation time of the segmentation program reasonable, the point cloud must first be filtered/down-sampled. That can be done using the Voxel Grid filter. As explained in [Lib], the VoxelGrid class creates a 3D voxel grid (think about a voxel grid as a set of tiny 3D boxes in space) over the input point cloud data. Then, in each voxel (i.e., 3D box), all the points present will be approximated (i.e., downsampled) with their centroid. This approach is a bit slower than approximating them with the center of the voxel, but it represents the underlying surface more accurately.

This step is only necessary when using multiple merged scans, not just individual ones. It is important to mention that filtering the point cloud changes the final model, as different cylinders are segmented due to the fact that some points are now missing compared to the original point cloud. This is visible in Figures 6.3 and 6.4, where the models differ even though they are segmented from the same original cloud. For comparison, a photo of the tree that was scanned to create the dataset is shown in Figure 6.5.

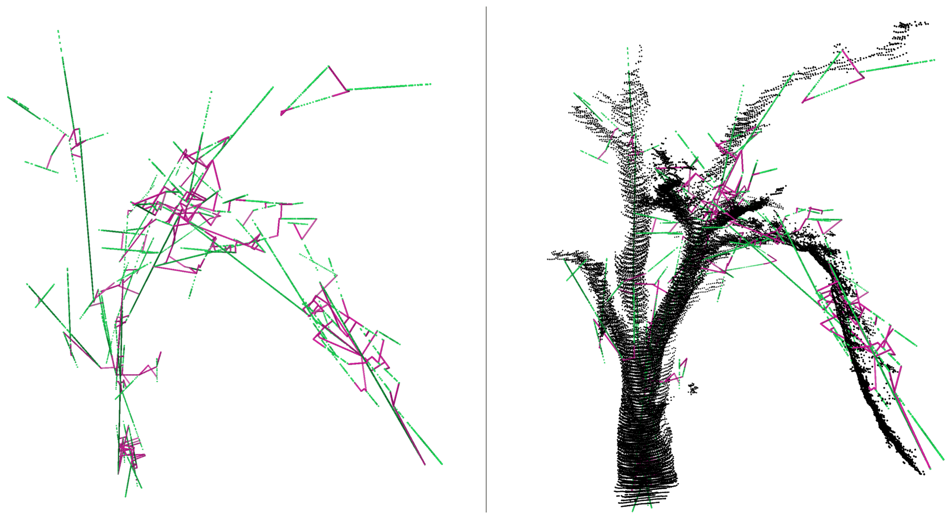


Figure 6.3: ICP Tree Graph w/o Voxel Filter

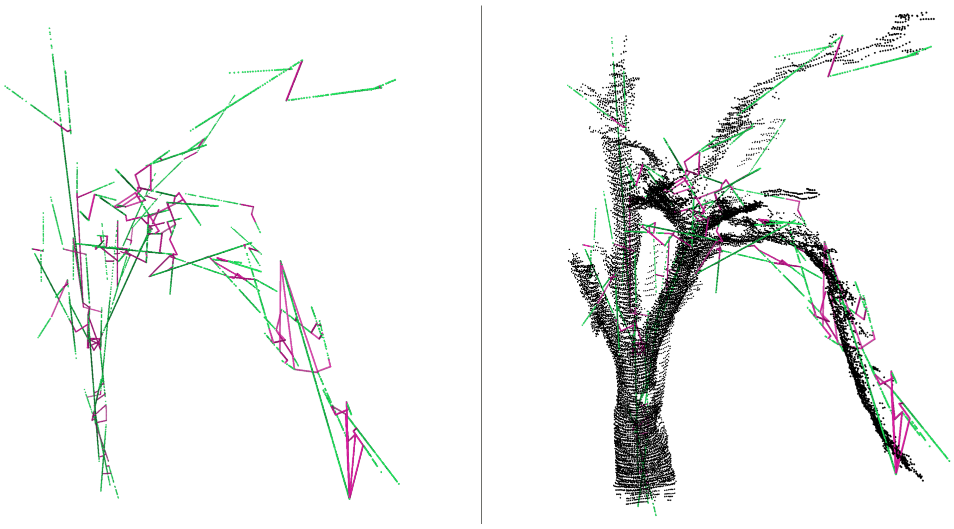


Figure 6.4: ICP Tree Graph with Voxel Filter

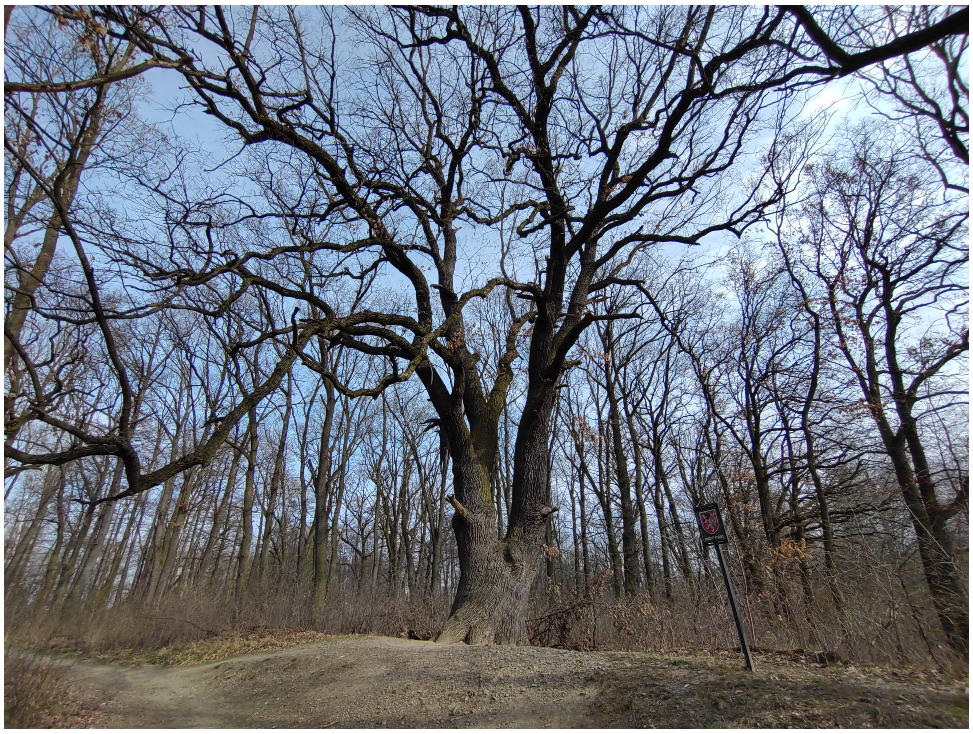


Figure 6.5: A Photo of the Tree

Chapter 7
Conclusion

This thesis first proposed an algorithm for detecting trunks of individual trees in a multi-tree environment (e.g., a forest) using point clouds from a LIDAR scan. This was done by segmenting and removing a plane representing the ground and then separating the point cloud into smaller clusters with Regional Growing Segmentation. Those clusters were then individually segmented with Cylinder Segmentation to approximate individual tree trunks.

This algorithm was then extended to detect and segment the trunk and main branches of an individual tree. Due to the uneven terrain in the dataset of the individual tree, a new method for removing the ground was presented. Points belonging to the tree were then identified using Region Growing Segmentation.

Next, adjacent branches in the scan were detected and connected based on a distance threshold. Furthermore, a method for fusing multiple scans of a tree was proposed.

Finally, the algorithm was tested on the merged scan of a tree, creating a model of the trunk and main branches of a protected tree of exceptional age.

In future research, it might be beneficial to utilize a more complex method for removing the ground around the tree, as the current one is not very robust and might not work when deployed on more complex datasets. Improvements could be made to methods regarding the detecting and connecting of adjacent branches as well.

Bibliography

[BM92] P.J. Besl and Neil D. McKay. “A method for registration of 3-D shapes”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (1992), pp. 239–256. DOI: 10.1109/34.121791. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=121791.

[Ebe99] David Eberly. *Dynamic Collision Detection using Oriented Bounding Boxes*. Geometric Tools. Mar. 1999. URL: https://www.geometrictools.com/Documentation/DynamicCollisionDetection.pdf.

[GML00] Stefan Aric Gottschalk, Dinesh Manocha, and Ming C. Lin. “Collision queries using oriented bounding boxes”. PhD thesis. Department of Computer Science, The University of North Carolina at Chapel Hill, 2000. URL: https://www.researchgate.net/profile/Dinesh-Manocha/publication/2807460_Collision_Queries_using_Oriented_Bounding_Boxes/links/56cb392008ae5488f0daea80/Collision-Queries-using-Oriented-Bounding-Boxes.pdf.

[RHV06] Tehreem Rabbani, F.A. Heuvel, and George Vosselman. “Segmentation of point clouds using smoothness constraint”. In: *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 36 (Jan. 2006). URL: https://www.researchgate.net/publication/228340970_Segmentation_of_point_clouds_using_smoothness_constraint.

[Huy08] Johnny Huynh. *Separating Axis Theorem for Oriented Bounding Boxes*. Dec. 2008. URL: https://www.jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf (visited on 05/02/2024).

[ML09] Marius Muja and David G. Lowe. “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration”. In: *International Conference on Computer Vision Theory and Application VISSAPP’09)*. INSTICC Press, 2009, pp. 331–340. URL: https://www.cs.ubc.ca/~lowe/papers/09muja.pdf.

[Rus09] Radu Bogdan Rusu. “Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments”. PhD thesis. Computer Science department, Technische Universitaet Muenchen, Germany, Oct. 2009. URL: `https://mediatum.ub.tum.de/doc/800632/941254.pdf`.

[Dou+11] B. Douillard et al. “On the segmentation of 3D LIDAR point clouds”. In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 2798–2805. DOI: `10.1109/ICRA.2011.5979818`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5979818`.

[RC11] Radu Bogdan Rusu and Steve Cousins. “3D is here: Point Cloud Library (PCL)”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China: IEEE, May 2011. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5980567`.

[KHW16] Michael Kusenbach, Michael Himmelsbach, and Hans-Joachim Wuensche. “A new geometric 3D LiDAR feature for model creation and classification of moving objects”. In: *2016 IEEE Intelligent Vehicles Symposium (IV)*. 2016, pp. 272–278. DOI: `10.1109/IVS.2016.7535397`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7535397`.

[MN20] Ninad Mehendale and Srushti Neoge. *Review on Lidar Technology*. May 2020. URL: `https://ssrn.com/abstract=3604309` (visited on 04/21/2024).

[Kak21] Sai Sharath Kakubal. *3D Oriented bounding boxes made simple*. Apr. 2021. URL: `https://logicatcore.github.io/scratchpad/lidar/sensor-fusion/jupyter/2021/04/20/3D-Oriented-Bounding-Box.html` (visited on 05/02/2024).

[Wik23a] Wikipedia contributors. *Lidar — Wikipedia, The Free Encyclopedia*. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Lidar&oldid=1217352554` (visited on 12/21/2023).

[Wik23b] Wikipedia contributors. *Point Cloud Library — Wikipedia, The Free Encyclopedia*. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Point_Cloud_Library&oldid=1210510654` (visited on 12/20/2023).

[Wik24a] Wikipedia contributors. *Line (geometry) — Wikipedia, The Free Encyclopedia*. 2024. URL: `https://en.wikipedia.org/w/index.php?title=Line_(geometry)&oldid=1220165134` (visited on 05/03/2024).

[Wik24b] Wikipedia contributors. *Octree — Wikipedia, The Free Encyclopedia*. 2024. URL: `https://en.wikipedia.org/w/index.php?title=Octree&oldid=1218866408` (visited on 04/21/2024).

[Azu] Azurity. *vscode-pcd-viewer*. GitHub. URL: `https://github.com/azurity/vscode-pcd-viewer` (visited on 05/11/2024).

[Lib] Point Cloud Library. *Point Cloud Library Documentation*. Version v1.14.0-dev. URL: `https://pointclouds.org/documentation/` (visited on 04/30/2024).

[Tat] Tatsuya Yatagawa. *vscode-3d-preview*. GitHub. URL: `https://github.com/tatsy/vscode-3d-preview` (visited on 05/11/2024).

[Wei] Eric W. Weisstein. *Normal Vector*. URL: `https://mathworld.wolfram.com/NormalVector.html` (visited on 04/25/2024).