Eötvös Loránd University

Faculty of Informatics

Dept. of Software Technology and Methodology

Automatic rail tie recognition and error detection using LiDAR point clouds

Supervisor:
Cserép Máté
Assistant Lecturer

Author:
Ertl Dénes
Computer Science MSc

Budapest, 2023

Contents

Chapter 1

Introduction

Rail ties are vital parts of the railroad infrastructure, as they both bear the weight of the rail tracks and support the trains passing over them. Over the years, this mechanical stress combined with the alternating weather conditions can deteriorate the condition of the rail ties, and consequently, the rail tracks can get damaged, too. Ignoring these defects can lead to serious accidents in the long term. To prevent this, railway companies perform regular checkups on the quality of the rail ties. These inspections consume an immense amount of human and financial resources. The main goal of my research is the precise and automatic detection of defects with the help of point cloud processing algorithms. Identifying these discrepancies could significantly reduce the amount of resources spent on the manual examination of railroads.

The methodology presented in this thesis demonstrates great potential in accurately detecting rail ties, as well as identifying faults according to user-defined threshold values. For the testing datasets, which in total cover an approximately 250 meters long section of railway, the proposed method successfully identified rail ties with an accuracy of 98.8% having 0% false positives and 1.2% false negatives. The algorithm was implemented with a strong focus on configurability and scalability. This ensures that the system can be easily adapted and customized to suit different requirements and scenarios.

The original dataset, which was provided by Hungarian State Railways (in Hungarian: *MÁV Magyar Államvasutak Zrt.*) covers approximately 18.5 km long and 130 m wide area of Hungarian rural railroad.

Chapter 2

Background

2.1 LiDAR technology

LiDAR, which stands for Light Detection and Ranging is a remote sensing technology that uses lasers to measure the distance between the sensor and the surrounding objects. This technology works by emitting a pulse of laser light and measuring the time it takes for the light to reflect off an object and return to the sensor. By repeating this process rapidly and collecting the reflected light data, LiDAR systems can generate a 3D point cloud of the scanned area with high accuracy and precision.

LiDAR has numerous applications in various fields, including topography and terrain mapping, urban planning, forestry, archaeology, and autonomous vehicles. Due to its ability to scan areas quickly and accurately, LiDAR has numerous advantages over traditional surveying methods. However, LiDAR systems can be expensive and require significant technical expertise to analyze the data. Despite these challenges, LiDAR is becoming increasingly popular due to its versatility and efficiency.

2.2 Methods of measurement

Regarding LiDAR, the following three main data collection methods can be distinguished:

1. Aerial Laser Scanning (ALS)
2. Mobile Laser Scanning (MLS)
3. Terrestrial Laser Scanning (TLS)

2.2.1 Aerial Laser Scanning

Aerial Laser Scanning captures 3D data of the Earth's surface using LiDAR sensors mounted to an aircraft. ALS is commonly used to create detailed and accurate digital elevation models (DEMs) and terrain maps used in various fields, including geology, hydrology, and cartography.

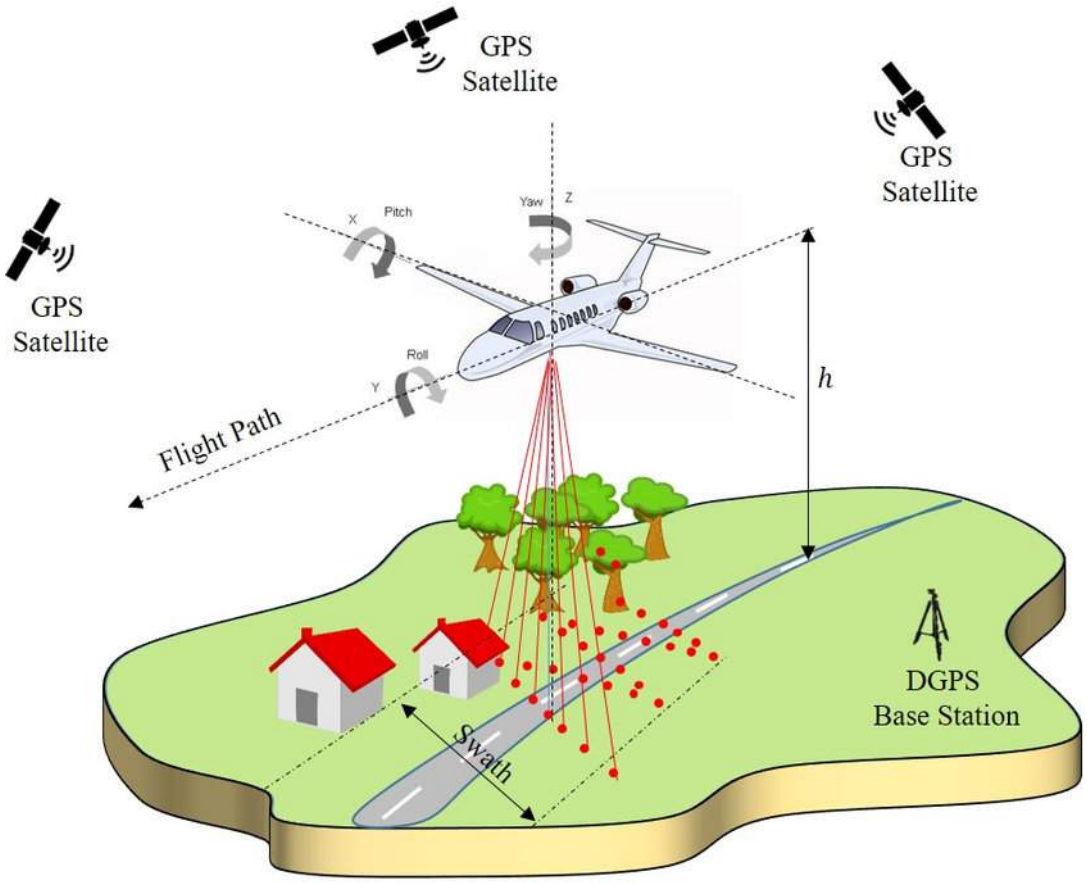


Figure 2.1: Aerial Laser Scanning visualized[1]

2.2.2 Mobile Laser Scanning

Mobile Laser Scanning captures 3D data of the environment using LiDAR sensors mounted on vehicles, such as cars or trains. MLS is commonly used to map out structures in urban areas. It can also aid in managing the infrastructure of a city by monitoring changes in buildings and roads.

[1]Aravind Harikumar, PhD thesis, `http://eprints-phd.biblio.unitn.it/3782/1/PhD_Thesis_Harikumar.pdf`

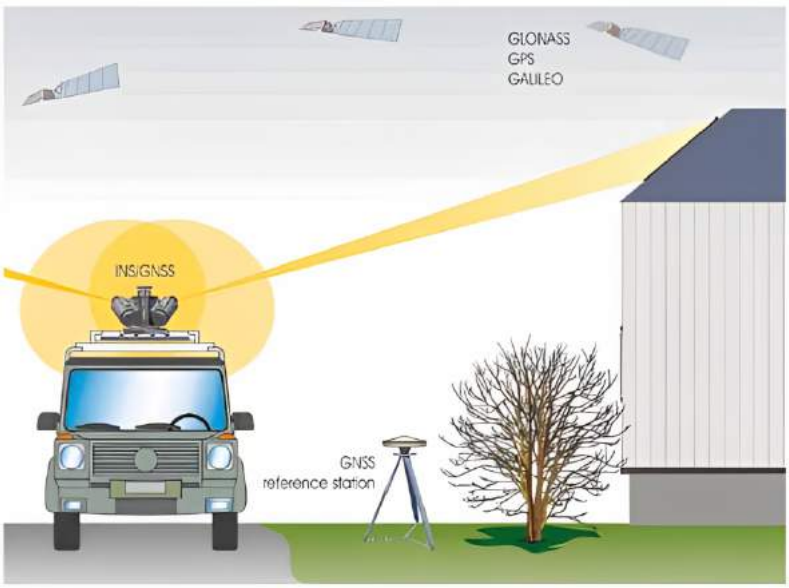


Figure 2.2: Mobile Laser Scanning visualized[2]

2.2.3 Terrestrial Laser Scanning

Terrestrial Laser Scanning captures 3D data of an object or environment using LiDAR sensors mounted on a tripod. TLS is commonly used in architecture and construction to create 3D models of buildings.

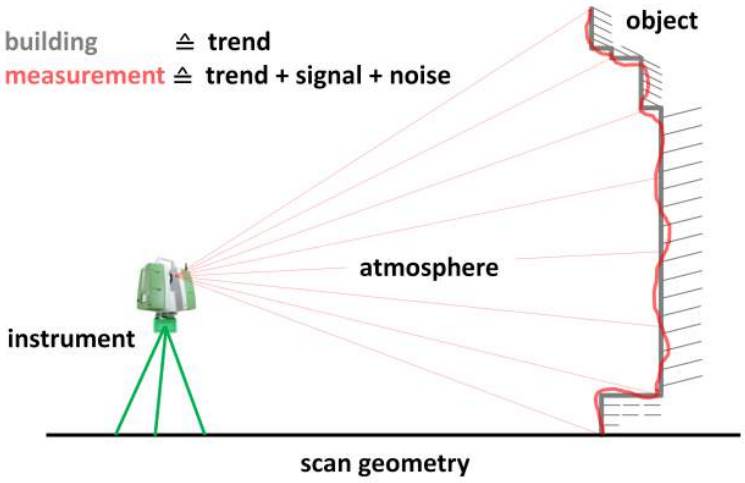


Figure 2.3: Terrestrial Laser Scanning visualized[3]

[2]Hanyun Wang et al.,2012 International Conference on Computer Vision in Remote Sensing -https://ieeexplore.ieee.org/document/6421248

[3]https://www.gib.uni-bonn.de/research/correlations-at-terrestrial-laser-scanning/correlations-at-terrestrial-laser-scanning

2.3 Rail ties

Since this paper focuses on detecting rail ties, it is essential to understand their importance and basic functionality.

Rail ties, also known as railroad ties or sleepers (rail ties in this paper), are rectangular wooden or concrete components used to support railway tracks. They are placed perpendicular to the rails, which are secured to them with spikes or other fasteners. They are essential components of the railway infrastructure, as they provide stability and support for the rails, ensuring the safety and efficiency of rail transportation.

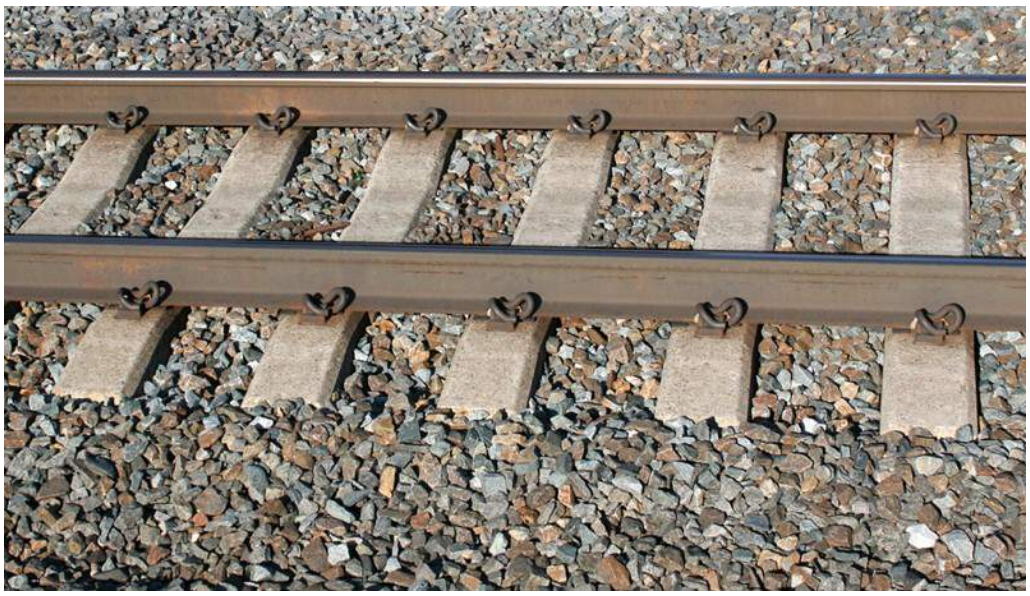


Figure 2.4: Concrete rail ties[4]

2.4 Usage of LiDAR in railways

Due to its high accuracy and ability to create detailed point clouds, the technology is widely used by many countries around the world to maintain railway infrastructure. Inspected parts of the infrastructure are for example tracks, sleepers, cables, switches, and signals. These data help to identify anomalies, such as track misalignment and defects, enabling timely maintenance and ensuring safe operations.

[4] `https://lancemindheim.com/wp-content/uploads/2015/09/ConcTiesA.jpg`

Chapter 3

Literature review

The purpose of chapter is to provide a comprehensive overview of the literature and research on the already existing methods of railway infrastructure segmentation. Since the available research on rail tie segmentation is relatively scarce compared to other rail infrastructure segmentation, the primary objective is to identify pre-existing methodologies that can be adapted for rail tie segmentation. Based on the related research, it was concluded that a highly accurate rail track detection algorithm is required to ensure reliable rail tie detection. Thus, the review also covers multiple papers on rail track segmentation.

3.1 Extraction of railroad objects from high-resolution ALS data

This article is a collaborative effort written by Neubert et al.[1]. It proposed two methodologies for detecting rail tracks.

3.1.1 Usage of an adapted RANSAC algorithm

With this method, the first step is to filter the points based on their height from the ground. This allows the algorithm to only consider points in the height of the rails. After this, a second filter is applied where the deviation in the height of points relative to the height of neighboring points is examined. The new point cloud will only contain points that fall within this interval.

RANSAC (Random Sample Consensus) is a robust algorithm used for fitting models to data that may contain outliers. It was first proposed by Fischler and Bolles [2].

The goal of this method is to use RANSAC to identify simple geometries in the dataset, such as lines and curves. Firstly, the data is fragmented into multiple parts (tiles) along the trajectory of the rail tracks. In each tile, the RANSAC method is used to identify the previously mentioned geometries. Finally, the separated tiles are connected back together. During this process, a curve-fitting function based on the Least Squares Method is applied. As a result, a curve fitted to the rail tracks is computed.

3.1.2 Rail object detection using 2D cuts

A knowledge-based classification method was also designed where 2D cuts (stripes) are taken from the dataset perpendicular to the trajectory of the rail. This created a so-called *2D profile* of the stripes. These stripes are later compared to a reference track profile. In order to follow the trajectory of the tracks, the following conditions were derived from the typical geometry of the area surrounding the track bed:

1. The track bed around the track is nearly horizontal and flat.
2. The railway loading gauge is free of objects.
3. The standard track gauge is known, meaning that rail tracks are always within a fixed distance of each other.

In the last step of the hierarchical analysis, the local terrain model is compared to the reference profile. If the second condition is fulfilled this comparison is performed for all areas. The results are the points that represent the track axis.

3.2 Rail object detection using height deviation

This method was published by Mostafa Arastounia in 2015 [3]. The classification of points is based on their local neighborhood.

Given the 3D points p and q. Point p belongs to the neighborhood of q if their Euclidean distance is smaller or equal to a given radius r:

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2} \leq r \tag{3.1}$$

Since finding all neighbors in a dense point cloud can be computationally demanding, a K-dimensional tree data structure is constructed to accelerate this process.

The objects are detected in the following order:

1. Recognition of the trackbed.
2. Recognition of the rail tracks.
3. Recognition of the power cables.
4. Recognition of the masts and cantilevers.

In case of rail tie detection, only the first and the second steps are significant. The points of the trackbed are classified based on the standard deviation of their neighborhood. The points of the trackbed have the smallest height variation to prevent the trains from colliding with objects between the rail tracks. After finding the trackbed, the same method can be used to find the rail tracks by detecting peaks in the standard deviation of height. Finally, the identified rail tracks are clustered and the accidentally disconnected clusters belonging to the same rail track are connected with a region-growing algorithm.

3.3 Rail object detection using eigendecomposition

Mostafa Arastounia has also published this methodology in 2017 [4], improving on the shortcomings of the earlier approaches. Previous methods assumed that the altitude of the area is roughly constant. This can cause problems in mountainous areas. The new method is more universal, since it uses eigendecomposition to identify rail track points. Points that are close to the ground are selected as rail track points. Since this method only uses local neighborhoods to classify rail tracks, it does not depend on the surface variation. False positives are filtered by applying a 2D Hough

transformation to the rail points, since it detects points lying on the same line and also computes the parameters of the respective line.

It is worth mentioning that the concurrent growth of rail tracks significantly enhances the speed of the method. Concurrency in the methodology used in the current paper was heavily inspired by this work.

3.4 Rail tie condition monitoring using ultrasonic ranging

The following method was proposed by Datta et al.[5]. Instead of LiDAR, ultrasonic sensors were used to measure the time-of-flight from the sensor mounted to the bottom of the rail carriage. The aim of the method was to measure the negative bending of rail ties (tension on the top surface) under train loads. The results obtained during this preliminary study indicated that the precision and accuracy are adequate to measure these kinds of phenomena. The aforementioned results are important for the current research, as rail ties are occasionally elevated from the track ballast by only a few centimeters. Since LiDAR is considered more accurate [6] than ultrasound, it is safe to assume that accuracy will not be an issue for rail tie detection.

3.5 Machine vision for the inspection of wooden rail ties

In the following thesis, an error detection method for wooden rail ties is proposed by Sajja Pasha Mohhamad [7]. The method uses Support Vector Machines (SVMs) for pattern recognition to classify rail ties into good and bad classes.

The method uses the following features for classification:

- Number of cracks
- Length of the cracks
- Width of the cracks
- Exposed length of the fastener bolts/screws

The features were extracted from grayscale images with the use of Canny Edge Detection [8] for the exposed length of the fasteners and Morphological Skeleton Operation for the cracks of the rail ties.

In the classification of 150 rail ties, the error rate was calculated to be 4.67%, out of which false positives accounted for 0% and false negatives accounted for 4.67%. The results suggest that with suitable features and training data, similar outcomes could be obtained for concrete rail ties.

3.6 Segmentation of railway tracks using deep learning

This article from Piotr Bojarczak and Waldemar Nowakowski [9] proposes a more universal method for the segmentation of the surface of railway tracks using deep neural networks. This method uses image-based data instead of LiDAR-based data and a Fully Convolutional Neural Network with 8 times upsampling (FCN-8) to detect railway track elements. The network was able to identify a wide variety of rail elements including the rail track, wood and concrete ties, and 4 different types of fasteners with great accuracy. In this research, the most important aspects are accuracy and precision of concrete rail tie detection. The neural network achieved a pixel accuracy of 96.5% and a precision of 90.5%. These numbers seem more reliable than the ones in the previously mentioned paper, considering the fact that the training used 5000 examples of concrete ties compared to the 150 wooden ties in the work of Sajja Pasha Mohhamad [7].

3.7 Conclusions

As it was mentioned at the beginning of this chapter, reliable detection of rail tracks is a prerequisite for the implementation of rail tie detection. The performance of the reviewed rail track segmentation methods is shown in Table 3.1 below.

Upon examination of each method, it was concluded that Arastounia's 2017 method is the most suitable for the needs of the current research due to its great precision and accuracy combined with its relatively low sensitivity towards height variations in comparison to other methodologies.

	Presision%	Accuracy%
Neubert et al. - RANSAC	>90%	-
Neubert et al. - 2D cuts	>90%	-
Arastounia 2015	97.1	96.4
Arastounia 2017	97.4	97.5

Table 3.1: The accuracy and precision of the examined methods of track detection

For the rail tie detection, the examined methods were mostly based on raster data except for the work of Datta et al. where ultrasonic ranging was used. Because of the lack of research on LiDAR-based data, it is irrelevant to compare the performance of this technology to the image-based methods. However, it is important to note that the explored methods imply that similarly precise and accurate results can be achieved with the dataset available to this research.

Chapter 4

Dataset

The datasets used in the current research were provided by the Hungarian State Railways. Data collection was carried out via a Riegl VMX-450 high-density mobile mapping system that was installed on a railway vehicle, traveling at a velocity of 60 km/h. The sensors are capable of producing up to 1.1 million points per second with an average error of approximately 3 millimeters. The dataset contains approximately $1.54 * 10^9$ points and covers about 18.5 km long and 130 m wide area of Hungarian rural railroad. For this research, 8 different samples were taken from different parts of the dataset (Table 4.1). These samples are divided into 2 groups based on the amount of track ballast located on the surface of the rail ties.

There are two main reasons behind this grouping:

- Rail ties that are covered by a greater amount of track ballast have a higher chance of having or developing defects in the future.
- Detecting rail ties that are covered by track ballast is challenging, testing the method on this type of data is necessary.

As a preprocessing step, the wider surroundings of the rail tracks were cropped around the rail tracks (Figure 4.1). At present, this step is carried out manually, however, automating this process has the potential to be a valuable improvement in the future. Chapter 5 provides a detailed discussion about the necessity of this step.

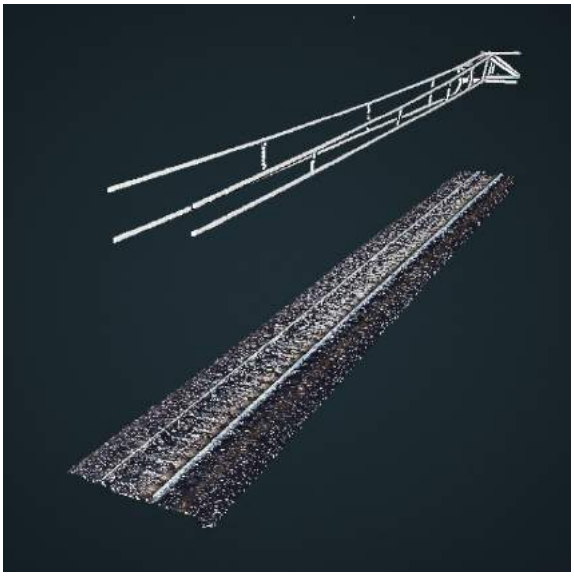
#	Total No. points	No. points after cropping	Section lenght in meters
1	1,871,224	492,644	24
2	1,945,590	419,848	26
3	3,168,634	812,957	52
4	1,084,746	300,421	18
5	950,613	250,798	14
6	2,665,050	651,452	38
7	2,289,283	643,285	33
8	1,231,579	324,668	20

Table 4.1: The datasets used for testing

(a) Sample with high vegetation before cropping

(b) Sample with high vegetation after cropping

(c) Sample with low vegetation before cropping



(d) Sample with low vegetation after cropping

Figure 4.1: Original and cropped samples with different amounts of vegetation

Chapter 5

Metrics of surface analysis

Before delving into the topic further, it is necessary to clarify a set of terms. The terminology used in this work was partly sourced from CloudCompare[1]. CloudCompare is a freely available software designed for editing, analyzing, and comparing 3D point clouds and meshes.

5.1 Normal change rate

This metric was taken from the surface analyzing tools of CloudCompare. Normal change rate is a real number greater or equal to zero. It represents the rate of change of a surface. The process of calculating the value of normal change rate for a point is the following:

5.1.1 Calculation of the covariance matrix

First, covariance matrix C is constructed.

$$C = \begin{bmatrix} Var(x) & Cov(x,y) & Cov(x,z) \\ Cov(x,y) & Var(y) & Cov(y,z) \\ Cov(x,z) & Cov(y,z) & Var(z) \end{bmatrix} \tag{5.1}$$

where

$$Var(a) = \frac{\sum_{i=1}^{n}(a_i - \overline{a})^2}{n-1} \tag{5.2}$$

[1] `https://www.danielgm.net/cc/`

and

$$Cov(a,b) = \frac{\sum_{i=1}^{n}(a_i - \overline{a})(b_i - \overline{b})}{n-1} \tag{5.3}$$

with n representing the number of neighbors contained in the neighborhood given by Equation 3.1.

5.1.2 Eigendecomposition of the covariance matrix

Following the computation of the covariance matrix, the eigenvalues and eigenvectors are determined by solving the equation below:

$$CV = \lambda V \tag{5.4}$$

Rearranging the terms gives the following equation:

$$(C - \lambda I)V = 0 \tag{5.5}$$

therefore

$$det(C - \lambda I) = 0 \tag{5.6}$$

where C is the covariance matrix, V is the matrix of the eigenvectors, λ representing the matrix of eigenvalues, and I being the identity matrix.

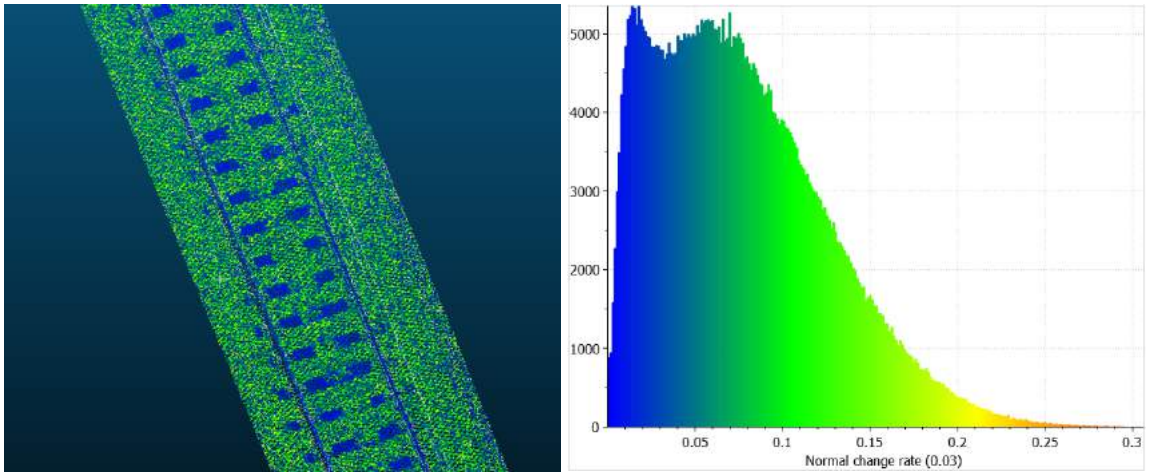
5.1.3 Calculating normal change rate from eigenvalues

After obtaining the 3 eigenvalues of the covariance matrix, the normal change rate is calculated in the following way:

$$\gamma = \frac{\min_{1 \leq i \leq 3} |\lambda_i|}{\sum_{i=1}^{3} |\lambda_i|} \tag{5.7}$$

where λ_i represents the i-th eigenvalue and γ represents the normal change rate. The code implementation of this calculation can be found in the source code of CloudCompare.

It is clearly visible in Figure 5.1a that smoother surfaces like rail tracks and ties have a lower normal change rate value than the track ballast around them.



(a) Normal change rate calculated for each point in the point cloud

(b) Distribution of normal change rate values with neighborhood radius of 3 cm

5.2 Roughness

Roughness is another metric taken from the toolset of CloudCompare. The roughness value of a point is defined as the Euclidean distance from the least-squares fitted plane of its local neighborhood. The process of calculating roughness is the following:

5.2.1 Calculating Least Squares Plane

Given n 3D data points $\{(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n)\}$, a 3x3 symmetric matrix A is computed with the following entries:

$$A = \begin{bmatrix} \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i * y_i & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i * y_i & \sum_{i=1}^{n} y_i^2 & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} y_i & n \end{bmatrix} \tag{5.8}$$

After this a 3-dimensional vector b is computed with the following entries:

$$b = \begin{bmatrix} \sum_{i=1}^{n} x_i * z_i \\ \sum_{i=1}^{n} y_i * z_i \\ \sum_{i=1}^{n} z_i \end{bmatrix} \tag{5.9}$$

After this, the Least Squares Plane is determined by solving the following equation:

$$Ax = b \tag{5.10}$$

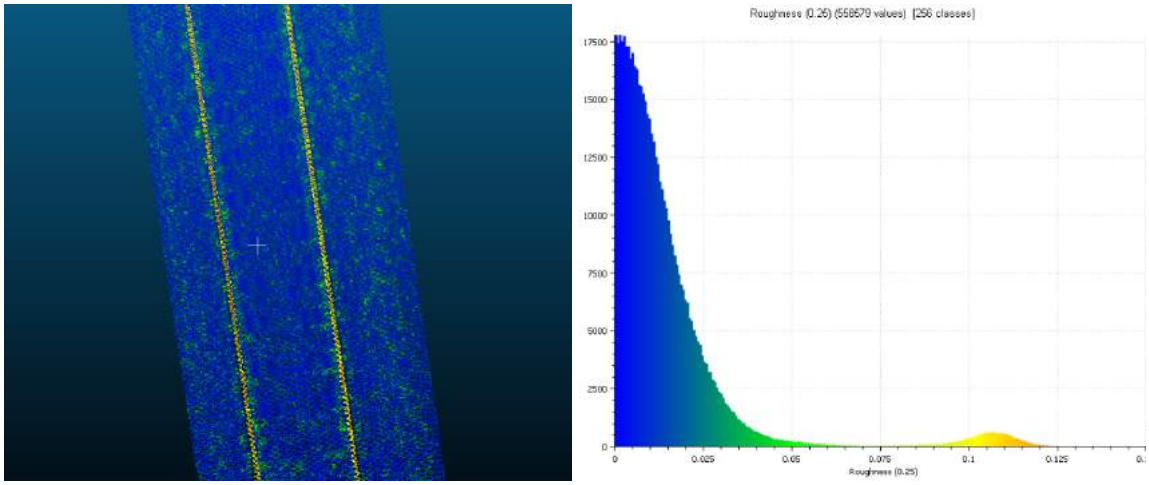
Where A is the matrix from Equation 5.8 and b is the vector from Equation 5.9.

5.2.2 Calculating point to plane distance

Given point (x_0, y_0, z_0) and a plane given by the equation $Ax + By + Cz + D = 0$, the distance of the point and plane is given by the following formula:

$$d = \frac{|Ax_0 + By_0 + Cz_0 + D|}{\sqrt{A^2 + B^2 + C^2}} \tag{5.11}$$

Roughness value is especially useful when identifying rail tracks as clearly shown in Figure 5.2a.



(a) Roughness calculated for each point in the point cloud

(b) Distribution of roughness values with neighborhood radius of 25 cm

Chapter 6

Methodology

The following chapter will examine the algorithm developed for rail tie detection in detail. The algorithm draws inspiration from Arastounia's proposed algorithms from 2015[3] and 2017[4] and introduces new methods for railway infrastructure segmentation as well. This chapter will provide a thorough overview of the algorithm by describing each step, beginning with a broad overview and gradually delving deeper into specific parts of the algorithm.

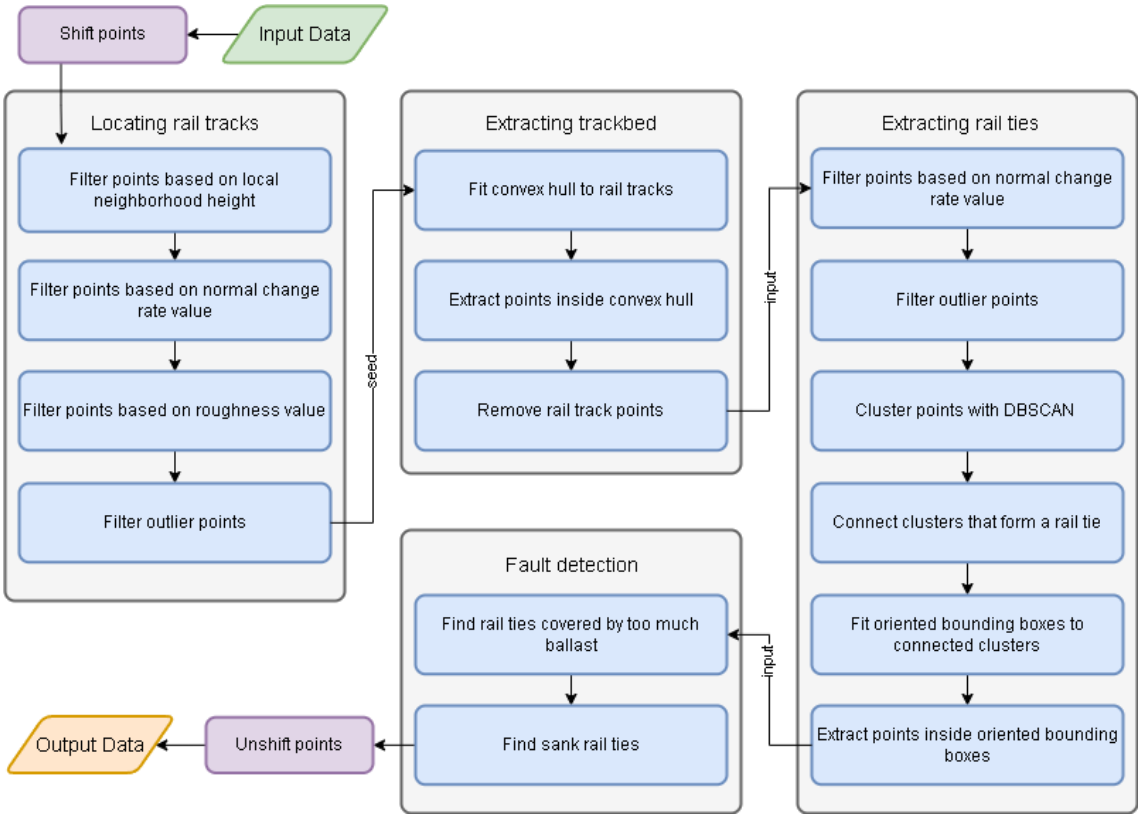


Figure 6.1: Flowchart of the algorithm

6.1 Shifting the points

Since this dataset is georeferenced, the points can have coordinates with magnitudes on the order of 10^5. To ensure the best precision, the amount of floating-point error propagation induced by these georeferenced coordinates needs to be reduced. The easiest way to achieve this is by moving the entire point cloud closer to the origin. This can be done the following way:

Given n 3-dimensional data points in the point cloud represented by $\{(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n)\}$. Taking the components of the first point in the point cloud, a shift vector $\overrightarrow{v}$ is constructed:

$$\overrightarrow{v} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \tag{6.1}$$

Using shift vector $\overrightarrow{v}$, function f is defined as below:

$$f(x, y, z) = (x - \overrightarrow{v}_x, y - \overrightarrow{v}_y, z - \overrightarrow{v}_z) \tag{6.2}$$

By applying this function to each point, a shifted point cloud represented by $S = \{f(x_1, y_1, z_1), f(x_2, y_2, z_2), ..., f(x_n, y_n, z_n)\}$ is obtained.

It is important to mention that shift vector $\overrightarrow{v}$ is preserved throughout the computations. After every computation is finished, the points are shifted back to their original positions. By doing this, georeferencing data is retained.

6.2 Partitioning the 3-dimensional space

In order to find the local neighborhood of a point, a naive algorithm would have a time complexity of $\mathcal{O}(n^2)$ where n is the number of points in the point cloud. Since the point cloud is dense with millions of points, this time complexity is undesirable. To accelerate this process, an octree [10] is used. An octree is a tree data structure used to represent a 3-dimensional space by recursively dividing it into eight octants. Each octant represents a smaller portion of space (Figure 6.2). This data structure is

utilized in the upcoming sections of this chapter when referring to the computation of the local neighborhood of a point.

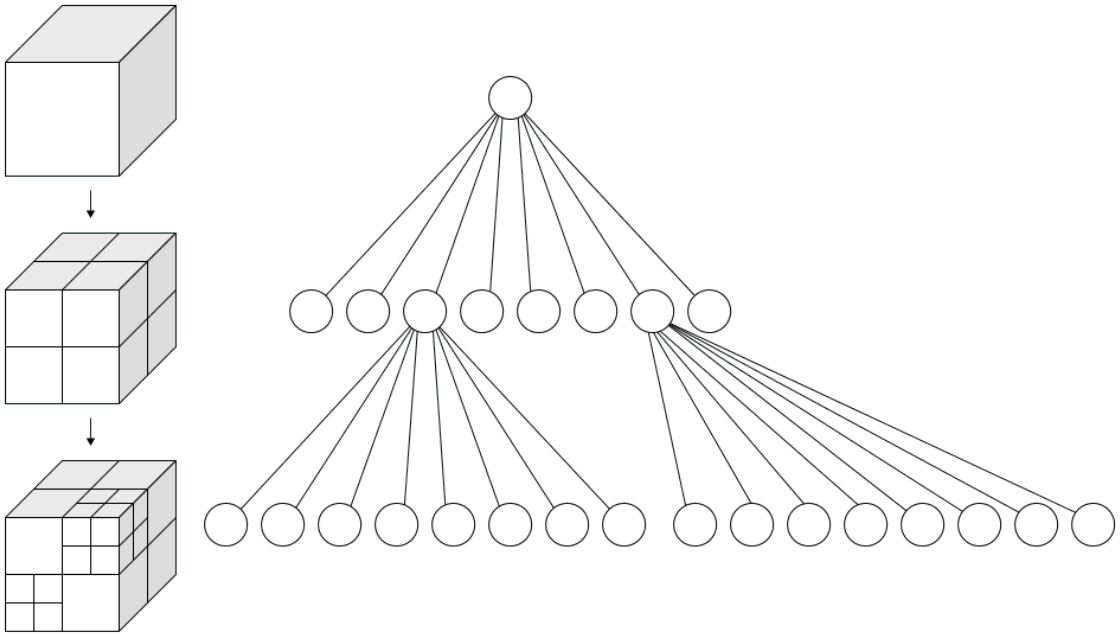


Figure 6.2: Octree data structure visualized [1]

6.3 Locating rail tracks

The scope of this phase is to locate rail tracks. This data will later be used to extract the trackbed where most of the rail ties are located. Note, that in this paper, the term trackbed only refers to the area located between the rail tracks.

6.3.1 Filter based on local height

In the first step, points that are too far from the ground plane are filtered. This is achieved similarly to the roughness value calculation discussed in Section 5.2. The only difference is that the neighborhood is calculated by finding points inside a bounding box instead of a sphere. The bounding box is defined by a center point as well as a minimum and maximum point. The minimum and maximum points are calculated as follows:

[1] WhiteTimberwolf - `https://commons.wikimedia.org/wiki/File:Octree2.svg`

Given a center point c, a side length of l, and n 3D data points in the point cloud $P = \{(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n)\}$ (Figure 6.3)

$$min = \begin{pmatrix} c_1 - \frac{l}{2} \\ c_2 - \frac{l}{2} \\ \min_{1 \leq i \leq n} P_{i3} \end{pmatrix} \qquad max = \begin{pmatrix} c_1 + \frac{l}{2} \\ c_2 + \frac{l}{2} \\ \max_{1 \leq i \leq n} P_{i3} \end{pmatrix} \tag{6.3}$$

This process is equivalent to projecting the point cloud to the XY plane and taking every point inside a square with center point c and a side length of l.

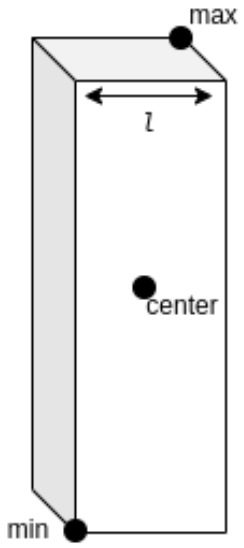


Figure 6.3: Bounding box visualized

After this, the distance to the least-squares fitted plane is calculated the same way as discussed in Section 5.2. In the current algorithm, l parameter is set to 15 cm, and the threshold distance from the fitted plane to 0.5 meters. In Figure 6.4, the removal of the catenary cable is clearly visible.

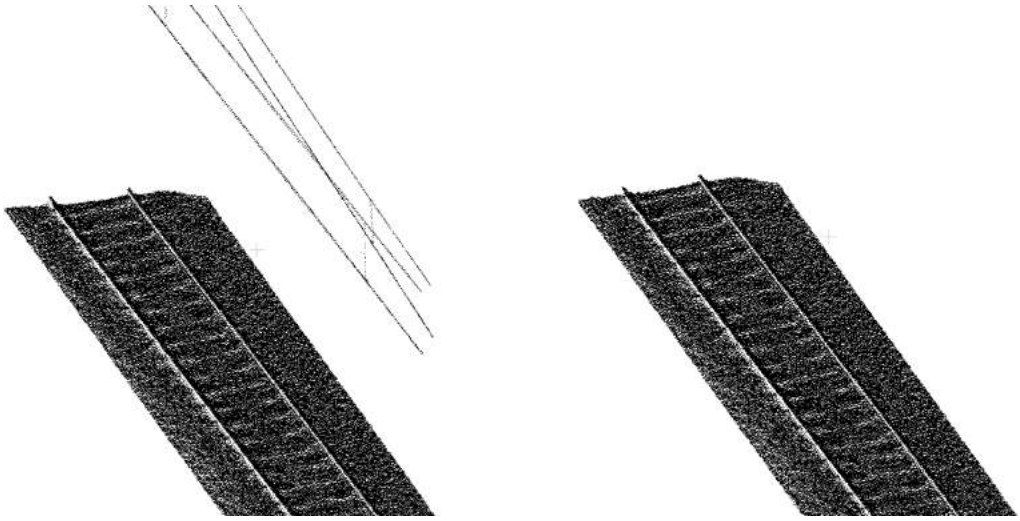


Figure 6.4: Point cloud before and after local height based filtering

6.3.2 Filter based on normal change rate

After filtering by height, the next step is filtering by normal change rate (Section 5.1). For this algorithm, 0.05 was found to be a suitable threshold value with a local neighborhood radius of 3 cm. Every point with a bigger normal change rate is filtered out from the point cloud. By applying this filter, most of the ballast is removed and only the rail tracks and the rail ties remain (Figure 6.5).

It is worth mentioning that an estimation mechanism for the radius parameter is also implemented. The key idea is that a certain number of points is needed for the normal change rate filter to work reliably. The algorithm uses an iterative method with binary search to approximate a radius where the mean number of points inside the radius is relatively close to this desired number.

The closeness is defined in the following way:

Given a positive real number *desired* representing the desired number of mean neighbors and a positive real number *current* representing the estimation of the latest iteration. The two numbers are considered close enough if the following condition is satisfied:

$$|desired - current| < \varepsilon \tag{6.4}$$

The execution of a single iteration is the following:

1. Take the radius and the mean number of neighbors calculated in the last approximation. If this is the first iteration, use the initial radius to calculate

this data.

2. If the approximation is good enough (Equation 6.4), the algorithm is finished. Otherwise, increase or decrease the radius by a *step* (equals to half of the initial radius by default), based on whether the current approximation is smaller or bigger than the desired one. In order to make the approximation converge, the *step* variable is divided by two in every iteration.

3. Take n samples with the new radius and calculate the mean number of neighbors.

4. If the maximum number of iterations is reached, the algorithm is terminated, and the best radius estimation is returned.

Since the distribution of the points inside the point cloud is roughly uniform, it is safe to assume that the obtained radius will produce valid results for the entire point cloud.

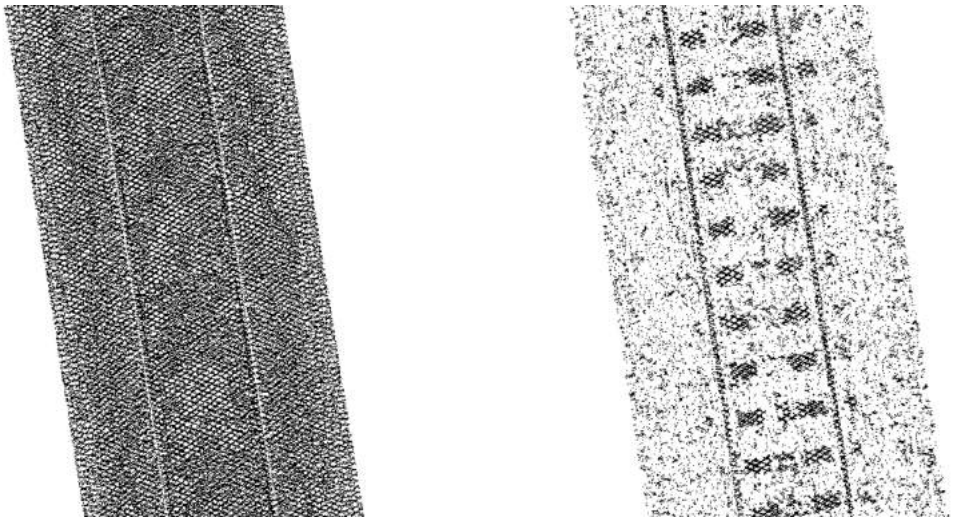


Figure 6.5: Point cloud before and after normal change rate filtering

6.3.3 Filter based on roughness

After obtaining the normal change rate filtered cloud, the rail tracks must be found. The rail tracks will be extracted by filtering based on the roughness value of the points. Points smaller than the threshold value of 0.1 will be removed. The radius used for the filtering is set to 50 cm. After applying the filter, only the rail tracks remain (Figure 6.6).

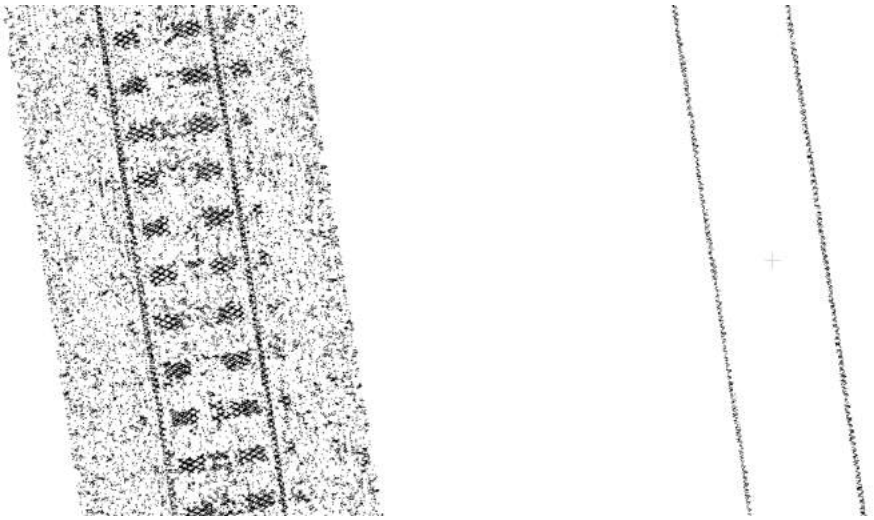


Figure 6.6: Point cloud before and after roughness-based filtering

After a thorough examination, it can be concluded that applying the roughness filter directly to the original cloud would yield similar results to those achieved by combining filtering based on normal change rate and roughness. The reason for this extra step comes from the different radius parameters used by the filters. Since normal change rate provides reliable results with a radius of only 3 cm compared to the 50 cm needed for roughness, the time complexity can be considerably reduced without significant loss of data (Figure 6.7).

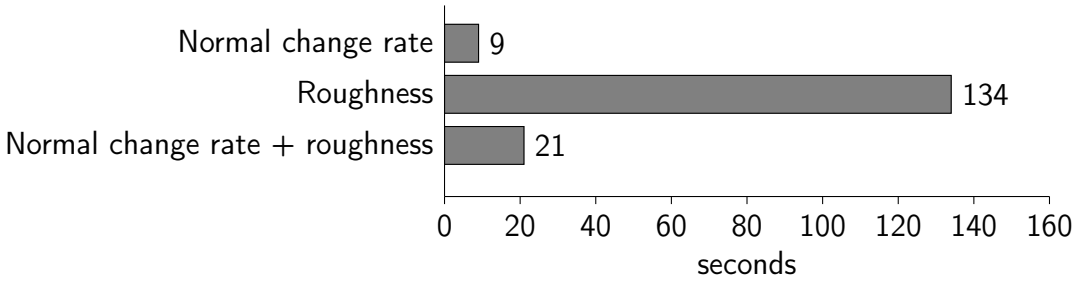


Figure 6.7: Filtering runtimes for approximately 800,000 points

6.3.4 Filter based on outliers

As a last step before the next phase, outliers are removed. In this context, outliers are points where the number of neighbors for a radius is less than a predefined threshold value. The radius, in this case, is 10 cm with a threshold of 16 points. This filter is essential before fitting a convex hull. Without this step, points outside the trackbed would also be included.

Figure 6.8: Point cloud before and after outlier-based filtering

6.4 Extracting the trackbed

In this phase, the trackbed is extracted from the point cloud using the rail track points obtained in Section 6.3.

6.4.1 Fitting convex hull to rail tracks

To find the track bed, the first step is to fit a convex hull [11] to the rail tracks. In 3 dimensions, a convex hull is the smallest convex polyhedron that contains all given points in a set.

6.4.2 Filter points inside the convex hull

After the construction of the convex hull, the original input cloud is used with the vertices of the hull to extract the points. This newly created cloud still contains the rail tracks, hence they still need to be removed.

6.4.3 Remove rail tracks

The process of removing the rail tracks can be accomplished easily. Given that the points of the rail tracks form a subset of the previously extracted point cloud (Section 6.4.2), computing the difference between the two point clouds will return the points that need to be removed. To compute the difference, it is necessary to determine the criteria for considering two points equal. In this case, two points are considered equal if their Euclidean distance (Equation 3.1) is less than 5 cm. This

distance helps to avoid accidentally including parts of the rail tracks, which can reduce the reliability of the next phase. In Figure 6.9 the result of this process is shown with the black area representing the track bed.

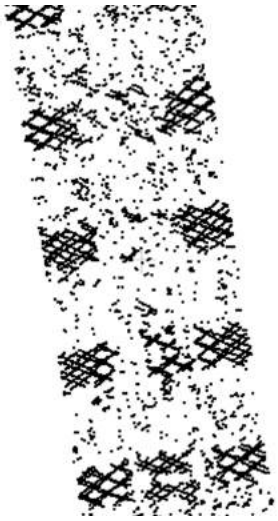
Figure 6.9: Point cloud after removing the rail tracks

6.5 Extracting rail ties

During this phase, the extraction of rail ties is performed on the trackbed obtained in Section 6.4. The filters utilized in this phase have been used in earlier stages of the process, thus they will only be discussed briefly.

6.5.1 Filter based on normal change rate

Similarly to Section 5.1, the points of the trackbed are filtered based on normal change rate, with the difference of using a threshold value closer to zero. Instead of 0.05 the new threshold is only 0.025. Through empirical observations, it has been determined that this new threshold reduces noise while preserving sufficient amount of data to ensure the reliable detection of rail ties. An observation that can be made by examining both Figure 6.10a and 6.10b is the consistent absence of the middle portion of the rail ties. This phenomenon can be attributed to the design of concrete ties, which incorporates a slight curvature instead of a completely flat surface (see on Figure 2.4). This curve allows the ties to better resist lateral forces generated by train movements.



(a) Trackbed filtered with normal change rate threshold of 0.05

(b) Trackbed filtered with normal change rate threshold of 0.025

6.5.2 Filter based on outliers

In this step, outliers are removed the same way as in Section 6.3.4, which ensures a clearer separation between different rail ties.

6.5.3 Cluster points with DBSCAN

After filtering the outliers, clustering can begin. After considering multiple different clustering methods, it was determined that DBSCAN (density-based spatial clustering of applications with noise) [12] is the best choice for this type of data. One of the main reasons for this decision is that DBSCAN does not require the number of clusters to be predefined and can discover clusters with arbitrary shapes. Moreover, the algorithm effectively handles noise by identifying and treating outliers separately. This makes it suitable for rail tie detection.

The basic workings of the algorithm are the following:

1. Before starting the computation, the algorithm requires two parameters: epsilon (ε) and minimum points (min_points). Epsilon determines the radius around a data point, while min_points specifies the minimum number of points within that radius for a point to be considered a core point.

2. For each data point in the dataset, the algorithm calculates the number of neighboring points within the radius of ε. If this number is greater or equal to

min_points, the point is classified as a core point. Otherwise, it is considered either a border point or an outlier.

3. Starting with a core point, the algorithm explores its neighborhood and recursively finds all reachable core points. A point is considered reachable if it can be reached through a series of core points within the ε radius.

4. The algorithm forms a cluster by connecting reachable points. It iterates through the core points, expanding the cluster by including their reachable points. This process continues until there are no more reachable points.

5. Border points that are not reachable from any core point are assigned to the nearest cluster if they fall within their ε radius. Otherwise, they are classified as outliers.

6. Any remaining unassigned points are labeled as outliers, since they do not meet the criteria of being core or border points.

In this case, the minimum number of points was set to 16 while epsilon was 3 cm. After running the algorithm, the clusters are created as expected. Generally, a rail tie consists of two separate clusters (Figure 6.11) because of the curvature discussed in Section 6.5.1.

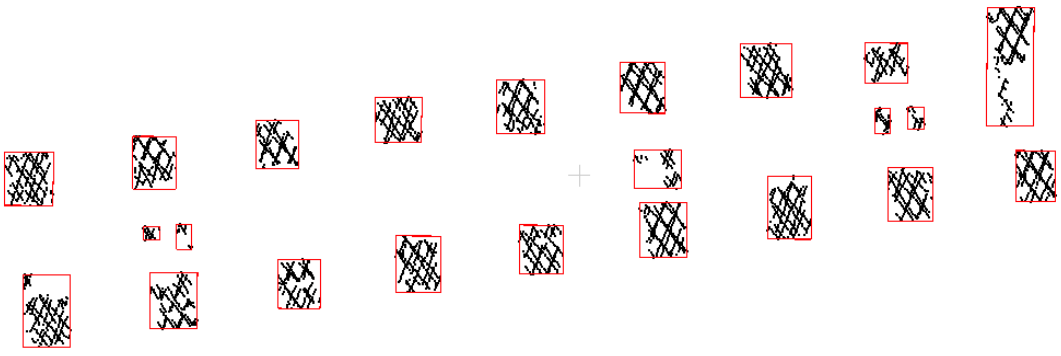


Figure 6.11: Clusters computed with DBSCAN

6.5.4 Connect clusters

To group the clusters accurately, it is necessary to compute multiple vectors for each cluster. These vectors visible in Figure 6.12 describe the expected orientation of the rail tie closest to the cluster.

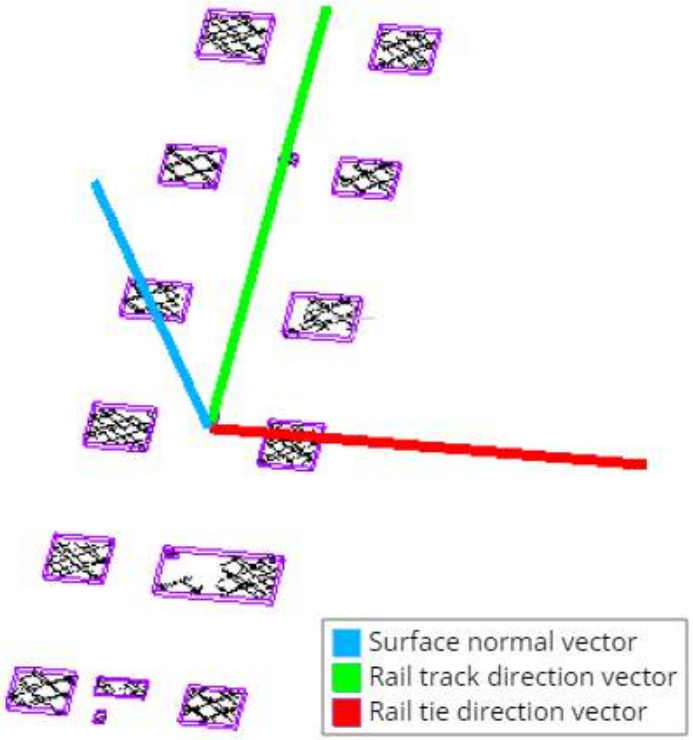


Figure 6.12: Vectors used to describe the orientation of rail ties

Compute surface normal vectors

Firstly, a normal vector of the surface is estimated for every cluster. This is done by fitting a plane to the cluster and its neighboring clusters within a 1-meter radius (Section 5.2.1). After calculating the equation of the plane, taking its coefficients as a vector and normalizing them will yield the normal vector of the plane.

Compute rail track direction

Secondly, the direction of the rail track is computed for each cluster. This is done by taking the neighboring clusters in a 5-meter radius and calculating a best-fitting line with RANSAC. Following this, the direction vector of the line is obtained and normalized.

Compute cluster centroids

Thirdly, the centroid of each cluster is computed. Given the n data points $\{(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n)\}$ of a cluster, centroid c is obtained by calculating the mean value for each dimension of the data (Equation 6.5).

$$c = \begin{pmatrix} \frac{\sum_1^n x_n}{n} \\ \frac{\sum_1^n y_n}{n} \\ \frac{\sum_1^n z_n}{n} \end{pmatrix} \tag{6.5}$$

Compute rail tie direction

Lastly, the rail tie direction can be easily derived using the normal vector and the rail track direction vector associated with the cluster. The idea is based on the assumption that rail ties are always oriented perpendicular to rail tracks.

Given a normal vector represented by $\overrightarrow{n}$ and a rail track direction vector represented by $\overrightarrow{s}$. Since $\overrightarrow{n}$ and $\overrightarrow{s}$ are approximately perpendicular, a third vector $\overrightarrow{t}$ orthogonal to both of them can be constructed by taking their cross product:

$$\overrightarrow{t} = \overrightarrow{n} \times \overrightarrow{s} \tag{6.6}$$

This new vector $\overrightarrow{t}$ represents the direction vector of the rail ties near the examined cluster. Given that both $\overrightarrow{n}$ and $\overrightarrow{s}$ are approximations, it is important to note that resultant vector $\overrightarrow{t}$ can possess minor imprecisions. However, for the purpose of rail tie detection, this error is considered insignificant.

Estimate rail tie width

As an additional step, the algorithm provides the option to estimate the width of the rail ties before attempting to connect the clusters. This can be useful because the specific requirements for rail tie width may vary depending on the particular railway system, geographic location, and engineering specifications.

The process of estimating rail tie width is the following:

1. The algorithm sorts the clusters in descending order based on the number of points.

2. After sorting the clusters, the top 10% is selected.

3. For each cluster, the algorithm iterates through its points looking for a pair of points a and b where the direction vector from point a to point b is approximately parallel to the rail track direction vector calculated in Section 6.5.4. Two vectors are parallel if the magnitude of their cross product equals zero. In this particular case, vectors are considered parallel if their cross product magnitude falls below the threshold of 0.15.

4. Following the collection of point pairs for each cluster, the algorithm proceeds by selecting the pairs with the maximum distance and averages them. This mean value is the result of the estimation.

Connect clusters based on rail tie direction vector

With the completion of the necessary calculations for the clusters, the process of grouping them by rail tie can begin. The algorithm used for this is a modified version of the DBSCAN algorithm mentioned in Section 6.5.3. In this modified version, the cluster centroids are clustered. Given two clusters with centroids c_1 and c_2, c_1 is reachable from c_2 if the following conditions are met:

1. The Euclidean distance between c_1 and c_2 is less than 1.435 meters. This particular threshold is based on the fact that in the majority of European countries, the standard track gauge is 1.435 meters. It is not possible for two clusters with greater distances to be part of the same rail tie.

2. The direction vector from c_1 to c_2 is either:

    - Approximately parallel to the rail track direction vector calculated in Section 6.5.4, meaning that the magnitude of their cross product falls below 0.185.
    - The Euclidean distance between c_1 and c_2 is less than the rail tie width estimated in Section 6.5.4. In case this estimation was not performed, the algorithm falls back to the default value of 20 cm.

It is important to mention that this relation between clusters is commutative. If c_1 is connected to c_2 then c_2 is connected to c_1. Similarly, if c_1 is not connected to c_2 then c_2 is not connected to c_1.

Figure 6.13 demonstrates an example of this relation for a single cluster outlined with purple. Clusters colored in green meet the criteria to be considered reachable, while the red and blue clusters do not.

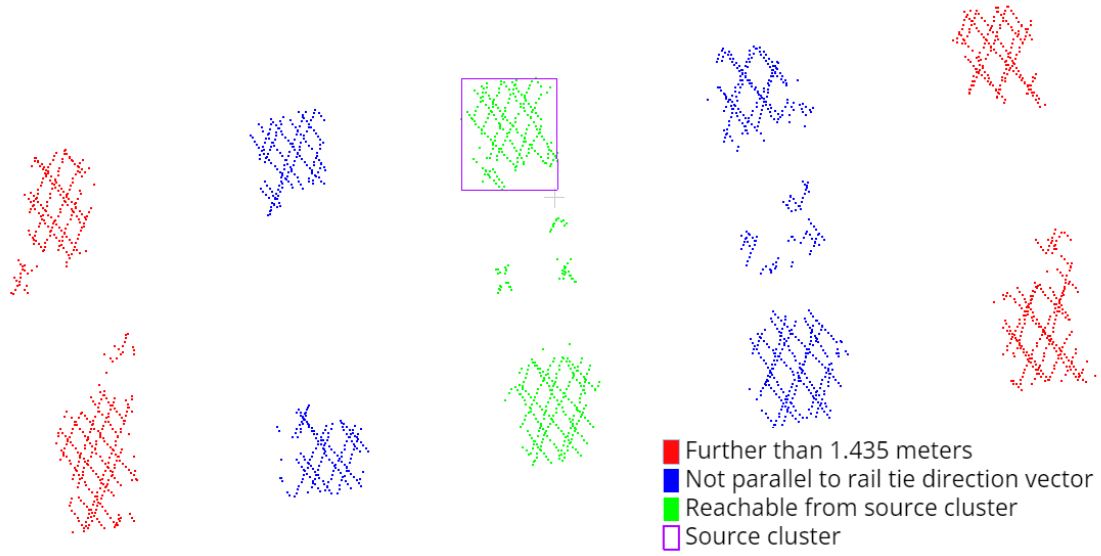


Figure 6.13: Identification of connected clusters for a single rail tie

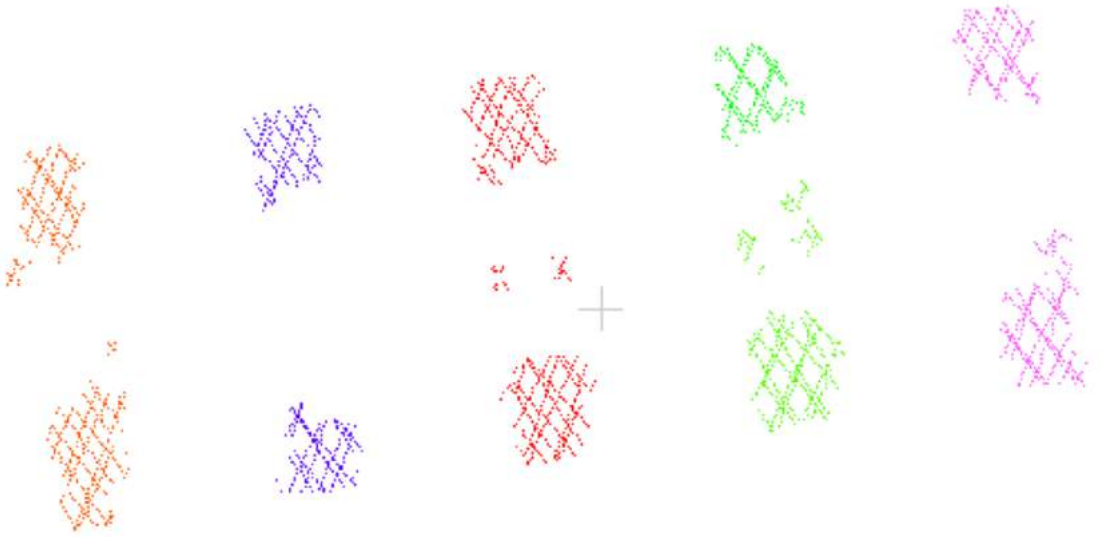
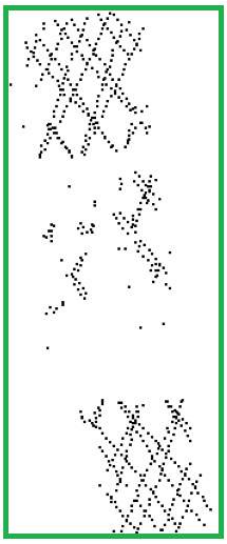


Figure 6.14: Connected rail ties represented with different colors

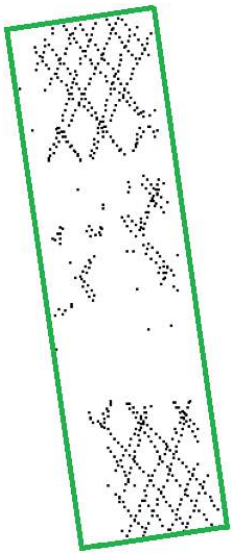
6.5.5 Fit oriented bounding box to grouped clusters

In order to find the obscured parts, an oriented bounding box is fitted to the grouped clusters. An oriented bounding box (Figure 6.15b) is similar to an axis-aligned bounding box (Figure 6.15a) but can be rotated freely in three-dimensional

space. Therefore, it is not restricted to align with the coordinate axes. This attribute of oriented bounding boxes is especially useful as their shape closely resembles that of the rail tie itself.



(a) Axis-aligned bounding box fitted to rail tie

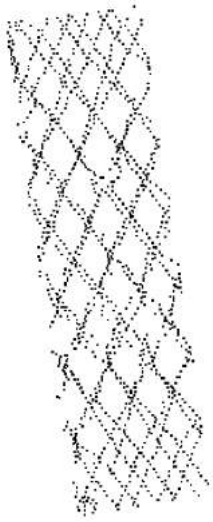


(b) Oriented bounding box fitted to rail tie

6.5.6 Extract points inside oriented bounding box

After the oriented bounding box is created, every point within it is classified as a rail tie point.

(a) Rail tie with missing points



(b) Rail tie points extended with points from the oriented bounding box

This process is repeated for every rail tie. Consequently, a significant portion of the points obscured by track ballast is identified (Figure 6.17).

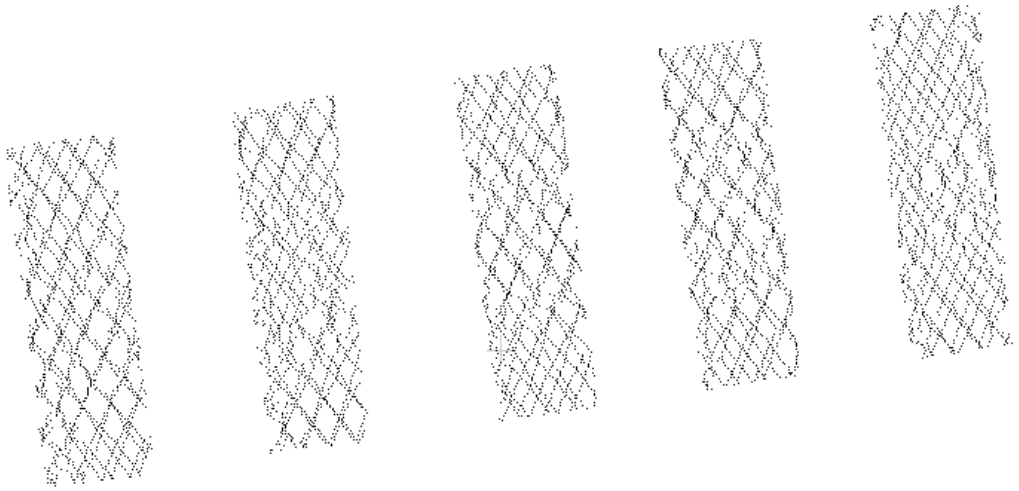


Figure 6.17: Rail ties after identifying obscured points

With this phase finished, the rail ties are obtained.

6.6 Fault detection

During the fault detection phase, the identified rail ties are examined based on the amount of track ballast located on their surface and the deviation from the height of neighboring rail ties. If a rail tie fails to meet the criteria for either of these factors, it is classified as defective. It is important to note that the algorithm in its current state lacks the ability to identify rail ties that are completely obscured by track ballast, meaning that it is unable to conduct fault detection of hidden rail ties.

6.6.1 Find rail ties covered by too much track ballast

After connecting the identified clusters in Section 6.5.4, the mean and standard deviation of the number of points in the connected cluster groups are calculated. A rail tie is considered defective if the following inequality is true:

$$n < \mu - \sigma * m \tag{6.7}$$

Where the variables represent the following:

$\boldsymbol{n}$: The number of points in a cluster group.

$\boldsymbol{\mu}$: The mean number of points in the cluster groups.

$\boldsymbol{\sigma}$: The standard deviation of the number of points in the cluster groups.

$\boldsymbol{m}$: The positive multiplier applied to the standard deviation. This determines the extent to which the threshold is adjusted. Values within the range of 0 to 1.0 will result in a tighter threshold, while values above 1.0 will lead to a looser threshold.

An example of these kinds of defects can be seen in Figure 6.18 and Figure 6.19, both showing the same area but with different m parameters.

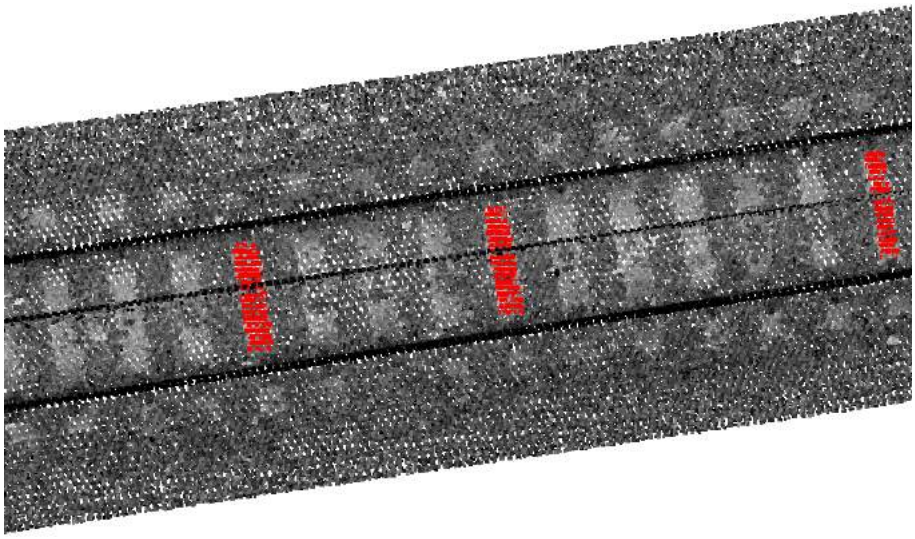


Figure 6.18: Rail ties covered by an excessive amount of track ballast ($m = 1.0$)

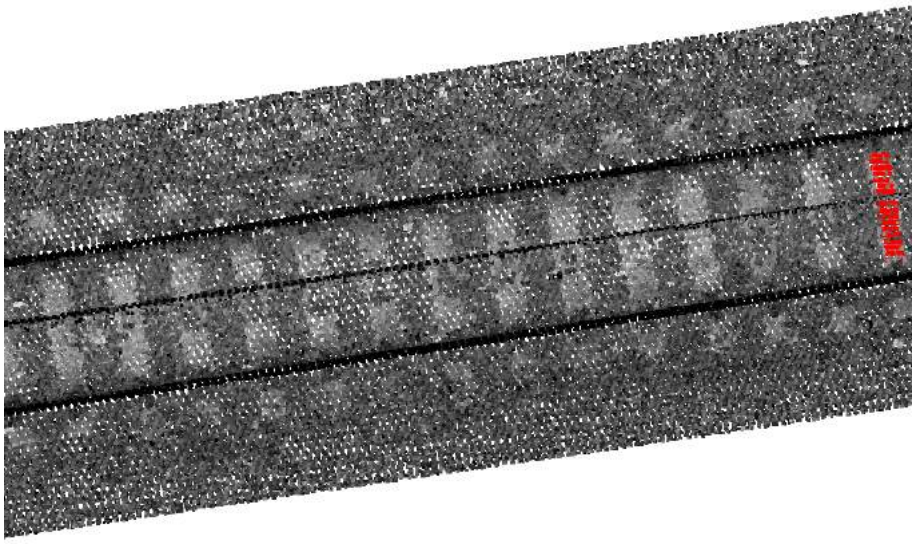
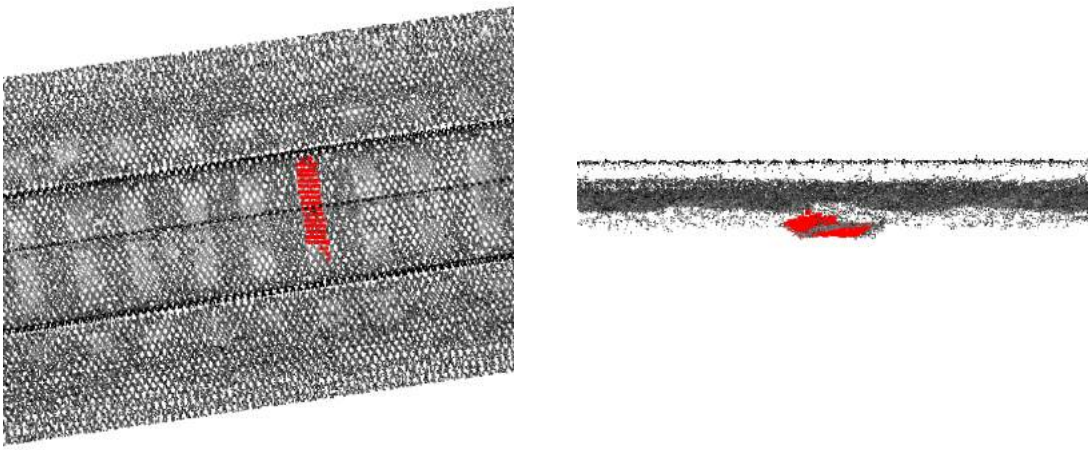


Figure 6.19: Rail ties covered by an excessive amount of track ballast ($m = 2.0$)

6.6.2 Find sunk rail ties

To find rail ties that are sunk into the trackbed, the first step is to get the mean height of the rail ties surrounding the examined rail tie. In case of significant deviation from this mean value, the rail tie is classified as faulty.

For demonstration purposes, this threshold was set to 5 cm, and a testing dataset has been created where one of the rail ties has been intentionally positioned lower than its surroundings (Figure 6.20b).



(a) Sunk rail tie (top view)

(b) Sunk rail tie (side view)

Chapter 7

Implementation

The implementation extends an already existing framework named *railroad*, developed by Cserép, Hudoba, and Vincellér [13]. This framework allows the step-by-step manipulation of an input cloud, as well as saving the final output. Due to performance considerations, the framework is implemented in C++. The project builds on multiple open-source libraries. Point Cloud Library(PCL) [14] is used to store and search point clouds effectively. OpenCV [15] is used for the projections and transformations. LasLib and LasZip are used for reading, writing, and compressing point cloud files. For the preprocessing of the data LASTools [16] is used.

7.1 Contribution

The majority of the contribution of the current research to the framework is the implementation of the following 6 filters in form of classes:

NormalChangeRateFilter: Computes the normal change rate value for each point as described in Section 5.1, and filters them based on a user-defined predicate.

RoughnessFilter: Computes the roughness value for each point and filters them based on a user-defined predicate as described in Section 5.2.

LocalHeightFilter: Computes the distance of each point from the least squares fitted plane of its neighborhood as described in Section 6.3.1.

TrackBedFilter: Given a seed with the points of the rail track, this filter removes points that are not part of the trackbed as described in Section 6.4.

OrientedBoundingBoxFilter: Crops the points not contained in any of the oriented bounding boxes (Section 6.5.5) provided as a parameter.

RailTieFilter: This filter combines the above 5 filters to obtain the points of the rail ties.

7.2 Code availability

The source code of the program is provided as an attachment to the thesis. Additionally, it is accessible on GitHub at:
`https://github.com/GISLab-ELTE/railroad`

7.3 Configuration availability

Due to the high number of parameters, the framework has been extended with a configuration file loader. This allows the user to specify the configuration parameters in a file without the need to recompile the entire program. The configuration file is provided as an attachment to the thesis. The configuration can be loaded by setting the *–algorithmConfig <input>* parameter of the executable in the CLI.

Chapter 8

Results and conclusion

The 8 samples used for the accuracy and runtime benchmarks were manually selected with the intention to test the robustness and fault tolerance of the proposed method. The length of the railway covered by the samples ranges from ∼20 meters up to ∼60 meters. The samples together cover approximately 250 meters of rail infrastructure.

The algorithm was benchmarked on a PC with the following specifications:

- CPU: AMD Ryzen 7 5700X 8-Core 3.4 GHz
- RAM: 32 GB (3600 MHz)
- Operating System: Ubuntu 20.04 LTS

8.1 Multithreading

It is important to note that the majority of the algorithm was implemented with multi-threading. This means that the workload is divided equally between the number of physical threads available in the specific CPU. Therefore the runtime of the algorithm highly depends on the hardware. In the above setup, the CPU had 16 physical threads, consequently the benchmarks were performed with 16 threads. To demonstrate the performance gains achieved through the utilization of multi-threading, the processing time of a sample with approximately 850,000 points was measured with 1,2,4,8, and 16 threads (Figure 8.1).

Taking the results into consideration, it is evident that the most significant performance gain (71%) was achieved by increasing the number of threads from 1 to

4. Subsequently, a notable runtime reduction of 35% was observed when comparing the utilization of 8 threads to 4. After this, increasing the thread count to 16 only yielded a 16% improvement compared to 8 threads.

These diminishing returns for higher thread counts can be attributed to the turbo boost technology [17] built into most modern CPUs. This technology allows some of the CPU cores to reach speeds beyond the base operating frequency for a short period of time. By running the computation in parallel on all of the available cores, this boost period is shortened. In the event of longer computation durations, it is expected that the gaps between the runtimes would continue to grow.

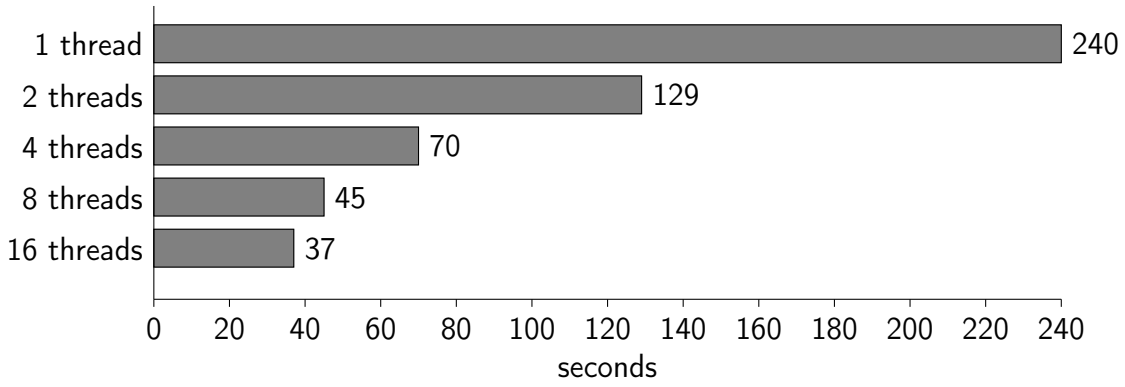
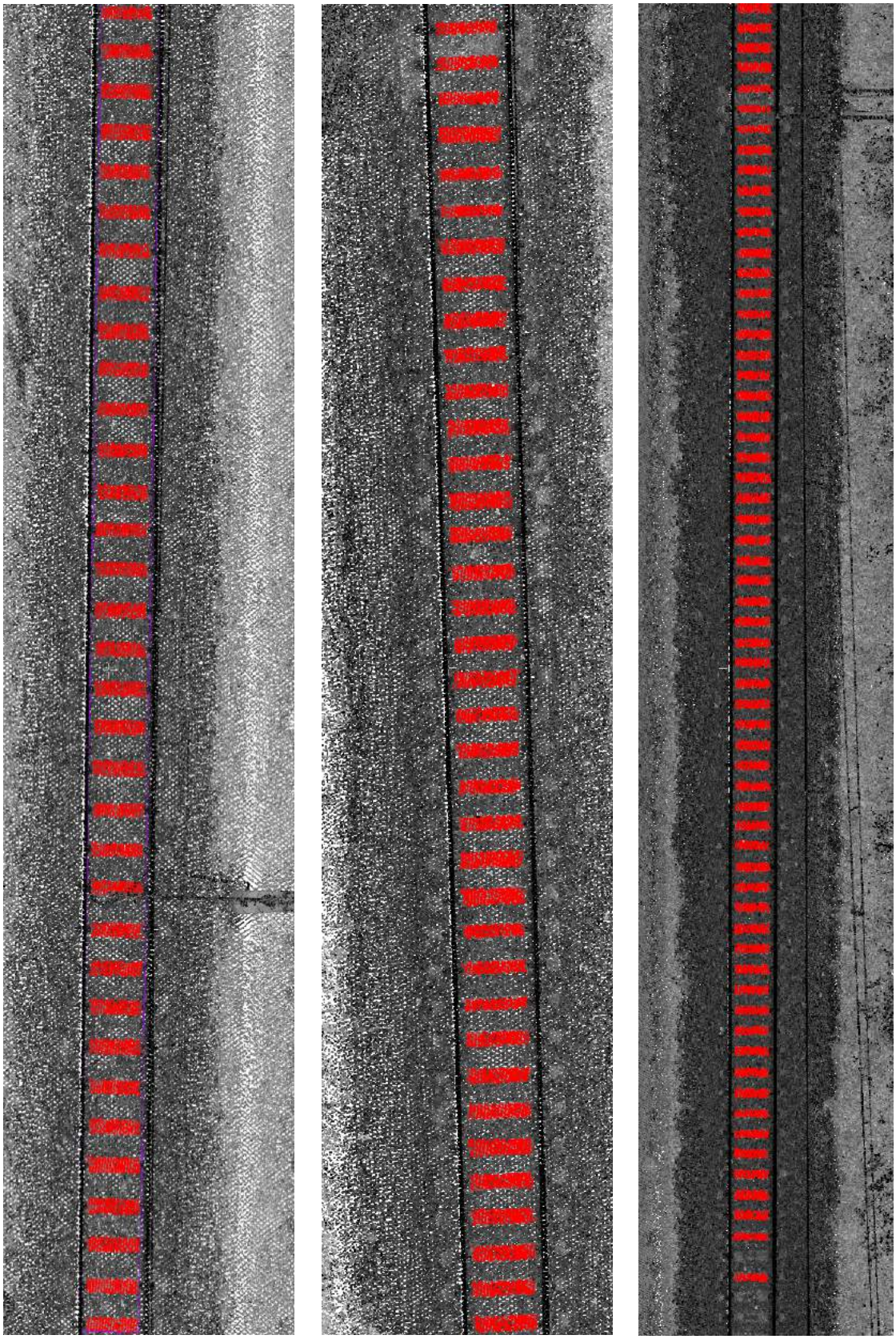


Figure 8.1: Runtime comparison in seconds with different number of threads (approximately 850,000 points)

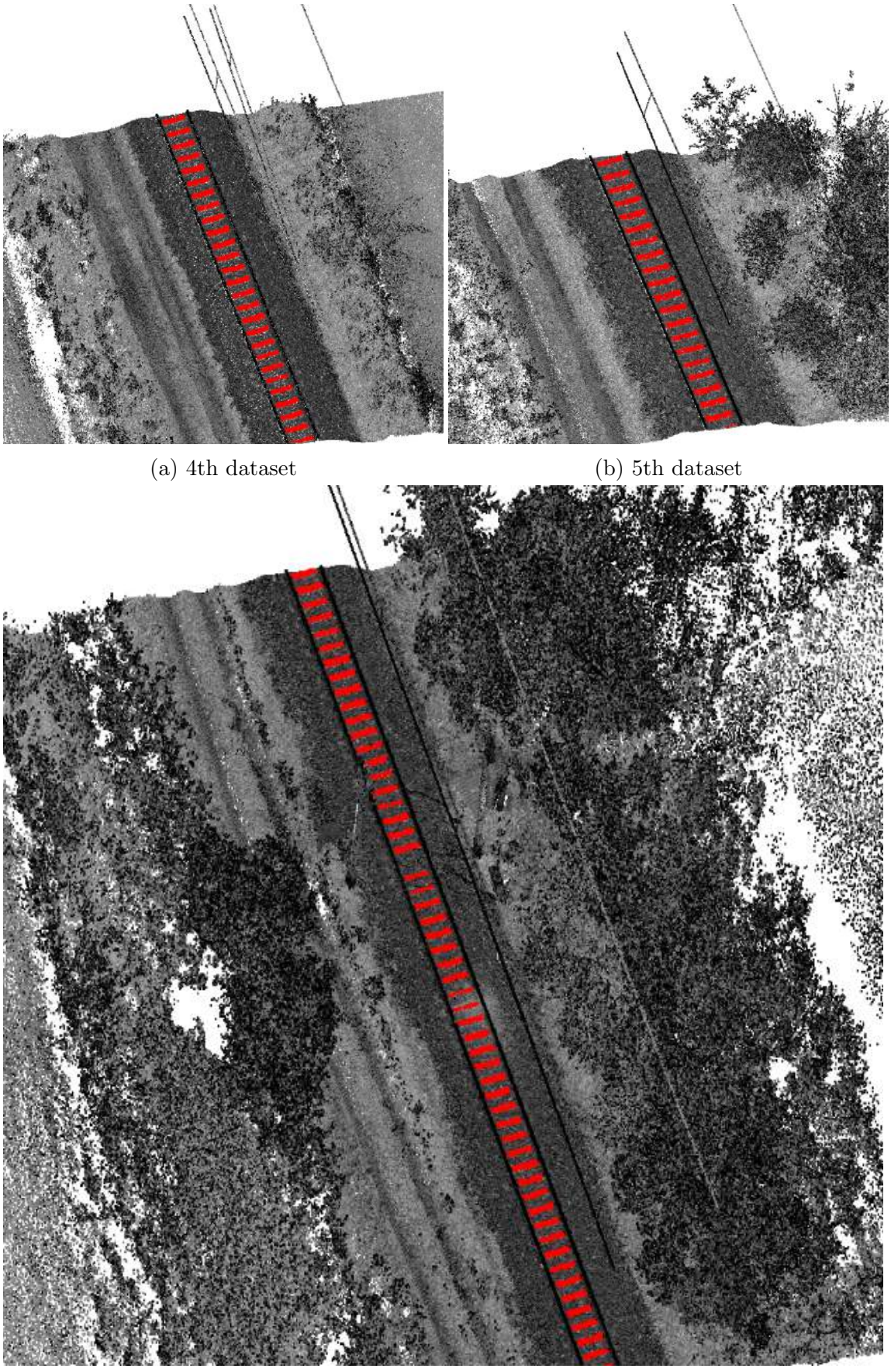
8.2 Rail tie segmentation results

In this section, the segmentation results of the algorithm will be presented and evaluated. In Figures 8.2, 8.3, and 8.4, the segmentation results of the 8 datasets can be seen with the identified rail ties colored in red. In Table 8.1, the runtime and accuracy results can be observed.



(a) 1st dataset (b) 2nd dataset (c) 3rd dataset

Figure 8.2: Rail tie segmentation results for the 1st, 2nd and 3rd datasets

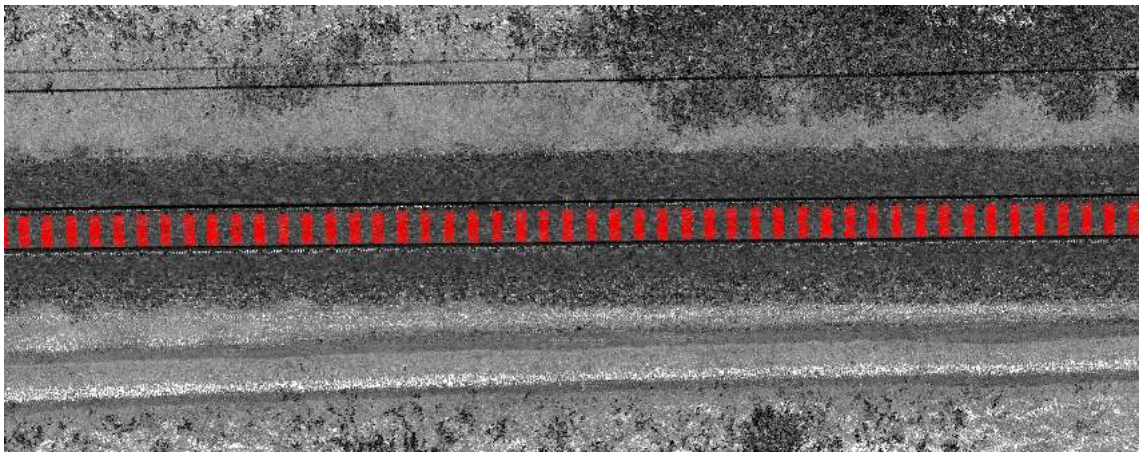


(a) 4th dataset

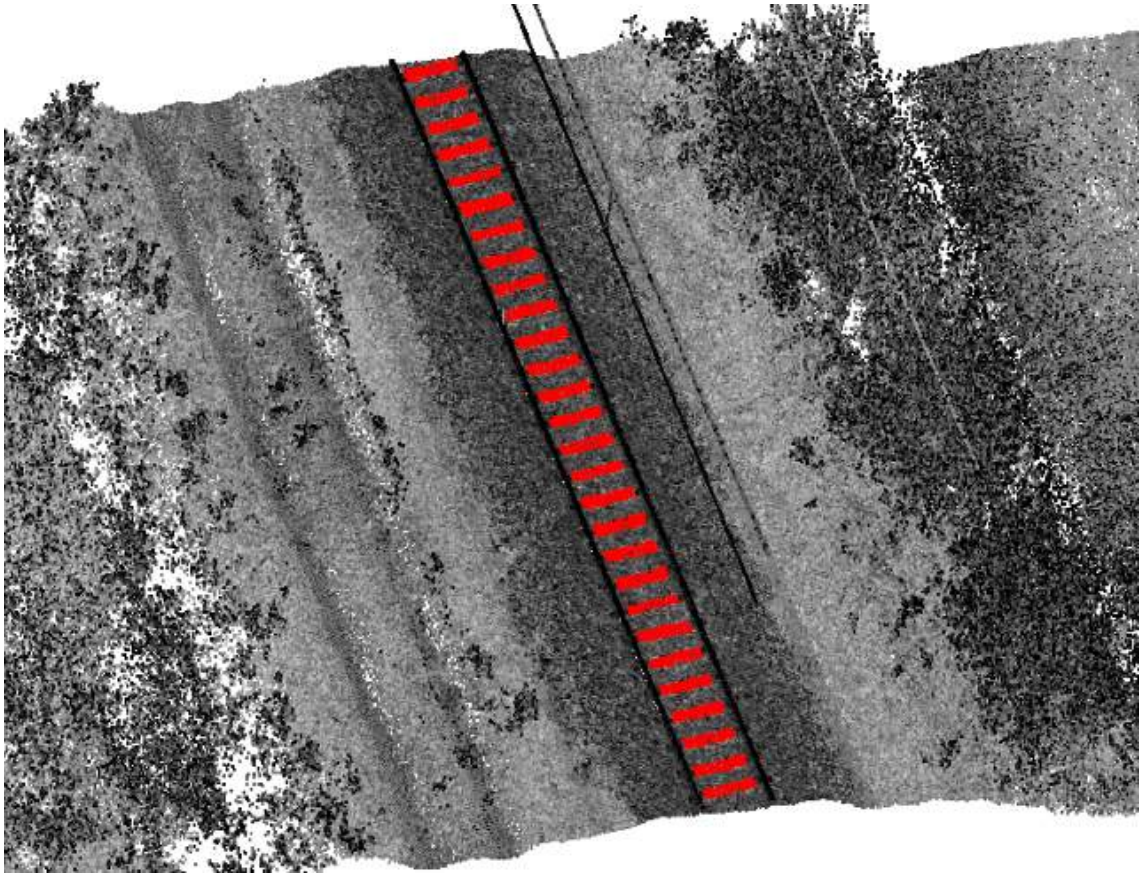
(b) 5th dataset

(c) 6th dataset

Figure 8.3: Rail tie segmentation results for the 4th, 5th and 6th datasets



(a) 7th dataset



(b) 8th dataset

Figure 8.4: Rail tie segmentation results for the 7th and 8th datasets

Upon examining the figures above, it is clearly visible that the algorithm demonstrates a high level of accuracy in correctly identifying the rail ties. However, a few instances of false negatives are noticeable in datasets 3 (Figure 8.2c) and 6 (Figure 8.3c). Due to the fact that not enough points were identified to reconstruct the entire surface of the rail tie, some of them deviate in size from their neighbors as well. For the testing data, 98.8% of the rail ties were correctly located (Table 8.1).

#	No. points	Runtime in seconds	Total No. rail ties	No. indentified rail ties	False positives	False negatives	Accuracy (%)
1	492,644	24	35	35	0	0	100%
2	419,848	22	40	40	0	0	100%
3	812,957	36	68	65	0	3	∼95.6%
4	300,421	15	27	27	0	0	100%
5	250,798	11	21	21	0	0	100%
6	651,452	29	57	56	0	1	∼98.2%
7	643,285	28	51	51	0	0	100%
8	324,668	16	28	28	0	0	100%
				Total	**0%**	**∼1.2%**	**∼98.8%**

Table 8.1: Accuracy and runtime results of the datasets

8.3 Conclusion

The goal of the current thesis was to propose a robust, LiDAR-based methodology for the segmentation of rail ties, while also introducing new ways for detecting faults in them. The results discussed in Section 8.2 prove that the rail ties can be identified with great efficiency and accuracy. With the usage of eigendecomposition for surface analysis, the algorithm has a high fault tolerance against height deviations of the surrounding terrain. It is also worth mentioning that the ability of dynamic parameter tuning makes the algorithm more flexible than other methods with fixed parameters. Moreover, the incorporation of multi-threading in the algorithm enhances its scalability compared to single-threaded ones.

On the other hand, there is still room for improvement. While the algorithm can locate the rail ties with great accuracy, the detected objects do not always align with the shape of the rail ties. Chapter 9 will discuss potential improvements regarding this issue.

An important finding worth mentioning comes from the domain of rail track segmentation. Even though the framework already has rail track segmentation described in the work of Albert Demján [18], roughness-based filtering (Section 5.2 and 6.3.3) was able to detect rail tracks faster and more reliably for the testing datasets used in this thesis. Additionally, this type of filtering was not affected by the curvature of the rail tracks. While the obtained results are promising, further research is necessary to validate these findings.

Chapter 9

Future work

Despite the promising results, there are numerous areas where the proposed method could be improved, both in performance and accuracy.

9.1 Pattern recognition for missing rail tie detection

In some cases, the rail tie is completely covered by track ballast, rendering the current algorithm incapable of detecting it. To address this issue, a pattern recognition algorithm could be implemented to detect missing rail ties as well as predict their position based on the successfully detected surrounding rail ties.

9.2 Improved edge detection of rail ties

During the outlier filtering phase, the points containing the edges of the rail ties can be incorrectly filtered out for not having enough neighboring points to be classified as inliers. To resolve this issue, an edge-detection method, such as Canny Edge Detection [8] could be implemented to restore the original edges of the rail ties.

9.3 GPU support for massively parallel computing

As it was already discussed in Section 8.1, parallel computation can greatly enhance the performance of point cloud processing algorithms. With the rise of GPGPU (General-Purpose Computing on Graphics Processing Units)[19] frameworks, such as CUDA [20] and OpenCL [21], transferring the responsibility of heavy

computations to the GPU can be easily implemented. This improvement has the potential to accelerate the computation time of currently implemented algorithms from minutes to seconds.

Chapter 10

Acknowledgements

Firstly, I would like to express my gratitude to my supervisor, Máté Cserép, for his guidance and support. His expertise and feedback helped me immensely throughout the research and writing process of my thesis.

I would also like to thank the Hungarian State Railways (MÁV Magyar Államvasutak Zrt.) for providing the datasets that were essential for the research and the code implementation of the proposed methodology.

Last but not least, I would like to thank Csaba Bajnóci and Zoltán Németh for their assistance and professionalism in railway-related questions, including specifications, regulations, and fault thresholds.

Bibliography

[1] Marco Neubert et al. "Extraction of railroad objects from very high-resolution helicopter-borne LiDAR and orthoimage data". In: (Aug. 2008). URL: https://www.isprs.org/proceedings/xxxviii/4-c1/sessions/session9/6718_neubert_proc_pap.pdf.

[2] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Commun. ACM* 24.6 (1981), 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: https://doi.org/10.1145/358669.358692.

[3] Mostafa Arastounia. "Automated Recognition of Railroad Infrastructure in Rural Areas from LIDAR Data". In: *Remote Sensing* 7.11 (2015), pp. 14916–14938. ISSN: 2072-4292. DOI: 10.3390/rs71114916. URL: https://www.mdpi.com/2072-4292/7/11/14916.

[4] Mostafa Arastounia. "An Enhanced Algorithm for Concurrent Recognition of Rail Tracks and Power Cables from Terrestrial and Airborne LiDAR Point Clouds". In: *Infrastructures* 2.2 (2017). ISSN: 2412-3811. DOI: 10.3390/infrastructures2020008. URL: https://www.mdpi.com/2412-3811/2/2/8.

[5] Diptojit Datta et al. "Railroad Sleeper Condition Monitoring Using Non-Contact in Motion Ultrasonic Ranging and Machine Learning-Based Image Processing". In: *Sensors* 23.6 (2023). ISSN: 1424-8220. DOI: 10.3390/s23063105. URL: https://www.mdpi.com/1424-8220/23/6/3105.

[6] Chandan K J and Akhil vm. "Investigation on Accuracy of Ultrasonic and LiDAR for Complex Structure Area Measurement". In: June 2022. DOI: 10.1109/ICOEI53556.2022.9777233.

[7] Mohammad Sajjad Pasha. "Machine vision for automating visual inspection of wooden railway sleepers". In: 2007. URL: https://www.diva-portal.org/smash/get/diva2:518382/FULLTEXT01.pdf.

[8] John Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698. DOI: 10.1109/TPAMI.1986.4767851.

[9] Piotr Bojarczak and Waldemar Nowakowski. "Application of Deep Learning Networks to Segmentation of Surface of Railway Tracks". In: *Sensors* 21.12 (2021). ISSN: 1424-8220. DOI: 10.3390/s21124065. URL: https://www.mdpi.com/1424-8220/21/12/4065.

[10] Donald Meagher. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. Oct. 1980.

[11] Ask Neve Gamby and Jyrki Katajainen. "Convex-Hull Algorithms: Implementation, Testing, and Experimentation". In: *Algorithms* 11.12 (2018). ISSN: 1999-4893. DOI: 10.3390/a11120195. URL: https://www.mdpi.com/1999-4893/11/12/195.

[12] Borbély Dávid and Tarczali Tamás. "Examination of drone imagery processing algorithms (in Hungarian)". In: 2021.

[13] Máté Cserép, Péter Hudoba, and Zoltán Vincellér. "Robust Railroad Cable Detection in Rural Areas from MLS Point Clouds". In: July 2018. DOI: 10.7275/z46z-xh51.

[14] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)". In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 1–4. DOI: 10.1109/ICRA.2011.5980567.

[15] OpenCV. *Open Source Computer Vision Library*. 2015.

[16] Martin Isenburg. *LAStools - efficient tools for LiDAR processing*. URL: http://lastools.org.

[17] James Charles et al. "Evaluation of the Intel® Core™ i7 Turbo Boost feature". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 188–197. DOI: 10.1109/IISWC.2009.5306782.

[18] Demján Albert. "Object extraction of rail track from VLS LiDAR data". 2020. URL: http://hdl.handle.net/10831/56224.

[19] David Luebke et al. "GPGPU: General-Purpose Computation on Graphics Hardware". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: Association for Computing Machinery, 2006, 208–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188672. URL: https://doi.org/10.1145/1188455.1188672.

[20] Ramandeep Singh Dehal et al. "GPU Computing Revolution: CUDA". In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. 2018, pp. 197–201. DOI: 10.1109/ICACCCN.2018.8748495.

[21] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

List of Figures

List of Tables