LiDAR Localisation in Unstructured Environments

Prepared by:
Kamryn Norton

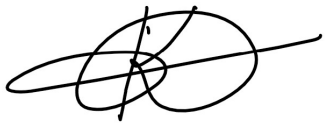
Prepared for: Dr Paul Amayo
EEE4113F
Department of Electrical Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Bachelor of Science degree in Mechatronics Engineering

October 26, 2023

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced, and is in my own words (except where I have attributed it to others).
3. This report and project is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.



October 26, 2023

Kamryn Norton

Date

Acknowledgements

> There are two ways of spreading light: to be the candle or the mirror that reflects it.
>
> —*Edith Wharton*

Keeping the words of Edith Wharton in mind, in this project, many people of significance have been the candle in my proverbial darkness. Throughout my degree and this project - which is the ultimate reflection of my academic journey thus far - I have encountered many challenges, both personally and professionally. I attribute most of my success and progress to the inspiration I draw and have drawn, from the people I choose to surround myself with, and I acknowledge them all.

I acknowledge my mentors from hyperTunnel: Mary-Jane Ochibe, Nicola Loxton, and Sid Shaikh, without whom I would not have the passion for the field of Engineering and Robotics that I consider an asset now. I could not have completed this degree without meaningful guidance and I have been astounded by the community and generosity shown by my peers, teachers, and colleagues.

To my supervisor, Dr Paul Amayo, for his support and guidance in this project, and since my early days in the degree as my electronics lecturer when I was first inspired by his research in robotics - I'd like to extend my gratitude. Additionally, Justin Pead, for his belief in me as a student and a tutor.

I acknowledge my friends - my chosen family - and I am so grateful to have had the support system that we have all created together, without which I would be lost! A special mention must go out to the groups we call '**The Aviary**': Meg (for all the 17 years!), Paige, Jess L, Jess P, and Caryn, '**The Joneses**': my engineers, and '**Not Using the Card**': Tash and Tris, and Rae Attridge - our coffees were my favourite.

To Jason Kreyfelt, for his support and being a listening ear; providing me with a safe space to be myself in both academia and activity. To his cat Bella, for keeping me warm and overseeing the writing of this document.

To Kerry and Andrew, and my chosen siblings, Cullum and Kieran, and Trinity: your support and influence on my whole life never go unnoticed and I am so thankful for you all (and the rusks from Kerry too!). And to Ginny and Brian, for being the pinnacle of a dependable and kind source of encouragement from so far away.

And finally, to my family: my Mom and role model, Caroline, you make each day brighter and I am grateful for our daily coaching catch-ups keeping me sane, for all my life. My Dad, Chris, for your support and words of wisdom, when I am in a tight spot - I appreciate all you do for me always. Lastly Ash, my 'little' sister - thank you for keeping me in check, and for your support even while you pursue your own dreams and degree. I appreciate you all beyond what I communicate on a page.

Abstract

This project aimed to develop and assess a SegMatch processing pipeline for localising a robot in unstructured environments using LiDAR data from the Husky's Velodyne sensor. Initially, a literature review was conducted to investigate existing technologies and methodologies. This was followed by the design and development of a data processing pipeline that was iteratively tested throughout its development using MATLAB in both structured and unstructured environments. The evaluation involved a comparative analysis of the final pipeline incorporating SegMatch and Normal Distribution Transform (NDT) registration methods based on data collected from these environments. In unstructured settings, NDT registration demonstrated better performance, particularly with full cloud data, while SegMatch struggled with accuracy and failed to localise. In structured environments, data corruption issues affected the integrity of the localisation process, providing inconclusive evidence towards the hypothesis validation. The timing analysis revealed a trade-off between processing speed and localisation accuracy, with faster processing observed in cropped data at the expense of reduced accuracy. The project partially validated the hypothesis, showing that NDT registration was more reliable in unstructured environments. The findings brought a need for further refinement of the SegMatch technique to light, and additional investigations are necessary in structured environments with improved data integrity.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Abbreviations

AMCL Adaptive Monte-Carlo Localisation

ARU African Robotics Unit

EKF Extended Kalman Filter

ES Euclidean Segmentation

FoV Field of View

GNSS Global Navigation Satellite System

ICP Iterative Closest Point

IDE Integrated Development Environment

INS Inertial Navigation System

ISS Intrinsic Shape Signature

Kd-Tree K-Dimensional Tree

LiDAR Light Detection and Ranging

M-LiDAR Mechanical LiDAR

NARF Normal Aligned Radial Feature

NDT Normal Distribution Transform

NN Nearest Neighbour

NSM Natural Segmentation and Mapping

OKPE Oriented Key Pose Extraction

PCL Point Cloud Library

PFH Point Feature Histogram

PMF Progressive Morphological Filter

RANSAC Random Sample Consensus

ROI Region of Interest

ROS Robot Operating System

SLAM Simultaneous Localisation and Mapping

SMRF Simple Morphological Filter

SS-LiDAR Solid-State LiDAR

UGV Unmanned Ground Vehicle

Chapter 1

Introduction

So the darkness shall be the light, and the stillness the dancing.

—*T.S. Eliot*

1.1 Background

This project, entitled 'LiDAR Localisation in Unstructured Environments', aims to provide insight into the current methods and practices of LiDAR localisation, with an emphasis on their application in unstructured environments. Following a review of the relevant literature available to date, this report details the methodology and processes adopted to achieve localisation in such environments. The areas of focus, particularly unstructured and non-urban terrains, include farmlands and zones previously traversed by the Husky robot, due to certain technical limitations and previous data being available. The final testing done was a comparative study between the unstructured and structured environments, and the performance of the developed data processing pipeline in the different scenarios.

1.2 Objectives

The overarching aim of the project is to enable the Husky robot, equipped with a LiDAR sensor, to identify where on a map it lies, based on key elements it sees around it at a point in time. The novelty of this project is in the development of a data processing pipeline necessary for the processing of the point cloud data, to enable the Husky to recognise elements in unstructured environments it has previously seen. This aim was developed after formulating the problem statement:

'*Robots and autonomously navigating vehicles have made significant progress in using LiDAR data, localisation and mapping in urban and structured environments, however, little research and development has been attributed to unstructured environments, where key environmental structures and features are difficult to identify, process, and utilise for place recognition.*'

1.3 System Requirements

The systems required for this project include both hardware and software components. The hardware is used to collect data and to run the processing pipeline on. The software is used to build the processing pipeline for the LiDAR point cloud data, and to complete the localisation.

Hardware

1. **Husky Robot**: The robot is used as the platform to drive through the different terrains and collect data for this project. It has external dimensions of 990 x 670 x 390 mm, and a weight of 50kg without the LiDAR sensor on it, and is taller when the sensor is mounted atop it.

Figure 1.1: Image of the Husky Robot: Taken after Data Collection

Figure 1.2: Image of Kamryn Norton with the Husky

2. **Velodyne Ultra-Puck VLP-32C**: The Velodyne sensor is a LiDAR sensor which has a 360° horizontal Field of View (FoV), and a 40° vertical FoV.

3. **Laptop**: Dell Inspiron 7391. It had an Intel® Core™ i7-10510U CPU (8M Cache, up to 4.90 GHz) and 16GB of RAM, and was running Windows 11 Home as its operating system. Ubuntu was also hosted on the laptop for purposes of installing and using external libraries.

Software

1. **Visual Studio Code**: This Integrated Development Environment (IDE) was chosen for familiarity reasons and because there is a large support base online for it. It can also be used with both Ubuntu and Windows, and through it, the terminal access for both the hosted Ubuntu environment as well as the Windows environment is seamless.

2. **Point Cloud Library & Dependencies**: This open-source C++ library and software is available online [15] and was used to do the initial methodology to implement the data processing pipeline. It has dependencies, such as Eigen, FLANN and VTK, which all needed to be installed.

3. **SegMatch & Dependencies**: the SegMatch Library [20] was necessary for the localisation through Segment Matching.

4. **MATLAB R2023a (academic)**: MATLAB was used for the last two pipeline design iterations as a necessity due to the first pipeline not being feasible. The three important toolboxes installed were the 'Automated Driving Toolbox','Computer Vision Toolbox', the 'LiDAR Toolbox', and the 'Navigation Toolbox' [10].

1.4 Scope & Limitations

This project is designed to investigate LiDAR localisation in outdoor (and unstructured) terrains, which has limited research and resources available. The scope of the project is to explore the current data processing techniques used for autonomous localisation and to implement a data processing pipeline which contributes to the autonomous navigation effort for the Husky by the African Robotics Unit (ARU). This contribution will enhance the knowledge and form a base for further research into the topic, and can be used by the ARU in future studies and expeditions with the robot.

Despite all efforts to ensure this project was as comprehensive as possible, every design and research effort has inherent limitations to it. The effectiveness of the LiDAR localisation is dependent on a number of factors, such as the small parameters in the methodology, and can differ significantly according to the data input to the processing pipeline. The data for this project, specific to the outdoors, was restricted to data already captured prior to the initiation of it. The indoor data for a comparative study was not limited as such, and collected as part of the project. Other limitations include processing speed, due to the need to change implementation methods and platforms from C++ to MATLAB, as well as the limited support available for localisation, specifically in outdoor terrains. The adaptability of the pipeline is also limited to the data available at the time of the project, and would require further study in the future.

1.5 Report Outline

The report follows an intuitive structure to aid in the understanding of all components of the processing pipeline. After a comprehensive study of the processing methods available to date in the Lierature Review (2), the design section sequentially follows the development of the data processing pipeline. This involves a sequence of pre-processing steps – which includes down-sampling, ground plane removal, feature extraction, and outlier elimination. Subsequently, the open-source SegMatch library, in combination with NDT registration, is implemented to segment the point clouds and match the new segments with previously identified ones in a created map, enabling loop-closure and localisation. The methodology also follows a comparative approach after the implementation of the pipeline, to evaluate the effectiveness of the processing in both structured and unstructured environments. From this, conclusions are drawn - summarising the key findings, insights gained, and recommendations for future studies.

Chapter 2

Literature Review

> The more we admit we do not know, the more opportunities we gain to learn.
>
> —*Mark Manson*

The field of robotics has witnessed impressive progress recently, with autonomous navigation and localisation emerging as critical fields of study and application - enabling robots to interact effectively in unstructured environments. As robots are applied to areas such as logistics, exploration, search and rescue, and transportation, their ability to navigate and accurately localise themselves emerges as a priority. The following review aims to provide a sound analysis of the current localisation techniques for robots, particularly focusing on their use in unstructured environments. The review will explore the evolution of these key areas, starting with autonomous navigation, followed by the significance of localisation, and ultimately leading to the challenges and advancements in the context of unstructured environments - including Simultaneous Localisation and Mapping (SLAM). The review will have an additional focus on LiDAR and its usefulness in the localisation of robots and UGVs will be explored.

2.1 Autonomous Navigation and Unstructured Environments

The field of robotic navigation has been around since the 1980s, even for uncharted terrain and outdoor environments [21] and is particularly prevalent in agriculture, forestry, mining, and human search and rescue applications - but it also features in autonomous vehicle navigation for motorists and in autonomous space exploration [22][23]. There are many ways of identifying an unstructured environment. It is usually outdoors, but could also apply to an indoor environment with many obstacles, and narrow or cluttered spaces [24]. According to Guan et al [22, p. 8139], an unstructured environment is one in which a robot needs to "navigate in uneven terrains or scenarios that lack a clear structure or well-defined navigation features." This definition is supported by the stance taken in the conference paper by Wang et al. [24] which suggests an unstructured environment is any uneven and unpredictable one which can include indoor ones, but it is often more synonymous with outdoor environments.

Guastella and Muscato [23] brilliantly outline how important it is for a robot to possess some perception of its environment, due to that perception being important for defining how the robot interacts with its environment [23]. This is supported by a statement made in another relevant paper which says: "Robots are operating more and more in unknown and unstructured environments, which requires a high degree of flexibility, perception, motion and control." [24, p. 1]. As part of their thesis, Bailey

mentions the importance of a robot being able to sense its position in an environment, especially as field robotics research grows and is shifting to the outdoors [25]. They also highlight the difficulties of this in unstructured environments, such as the uneven terrain and lack of defined structural lines [25]. In vegetated areas specifically, growth of the vegetation and natural materials can cause a drastic change in the appearance of an environment or cause occlusion of recognisable features of the environment [26], and so localisation becomes an important concept, especially where it needs to be adaptable depending on the type of environment the robot is in.

2.2 Localisation

Localisation is the process of a robot estimating its whereabouts within an environment, or its 'pose' whereas mapping refers to the reconstruction of this environment using sensors and sensor fusion [24]. This differs from robotic navigation in unstructured environments which needs to account for terrain recognition and analysis or use methods that map the robot's immediate environment to some specific control action [23]. Localisation ties directly into navigation, however, because a robot needs to first be able to detect where it belongs in a space before it can traverse that space [25]. In wheeled robots and UGVs, it is common to use a combination of odometry and other environmental sensors to do pose estimation or localisation of a robot [27][28][29].

Localisation is important because any task that requires a robot to move or operate in a space or environment will require the robot to have an awareness of its relative position in that space. In other words, localisation requires the robot to have a frame of reference. Here, it is evident that when in an unstructured environment, providing a frame of reference relies heavily on the ability of the robot to identify landmarks or point references in that environment, while also doing so accurately enough for it to gauge its position relative to that reference despite appearance changes due to clutter, occlusions, or seasonal changes [26]. In order to sense where a robot is in its environment, it needs sensors and mapping – and so it is important to investigate the relevant current methods of doing so.

2.3 Current Localisation Methods and Applications

Use of wheel odometry, and alternatively LiDAR or visual odometry, is commonly used as a method of localisation: the robot's position is a differential position relative to a starting point [30]. Dead reckoning is an estimation of a robot's location using one or many measurements describing its motion, however, this is undesirable for long-term solutions as each time an attempt at localisation is made, it includes an error that accumulates over time and change in position [25]. The concept of cumulative error associated with odometry mentioned above is supported by Peel et al. [30] and they affirm it to be part and parcel of the odometry method for localisation. Peel et al [30] mention how this simple localisation method is improved using filters, specifically Kalman filters, and the drawback of this improvement is said to be that the robot is no longer able to localise itself again if it loses its position. For a robot to re-localise, it would need some sort of reference. Incremental methods, or those that are differential relative to a unique starting point, are not ideal for unstructured environments due to their

specific challenges and detail [30].

Bailey [25] affirms the stance of Peel et al. [30] by describing the importance of absolute measurements above incremental ones, as it alleviates the issue of incremental error – and introduces priori map-based localisation which requires the area of navigation to have been mapped, with significant landmarks identified and located, prior to autonomous navigation of the robot [25]. The extension of this absolute measurement is to enable the robot to navigate autonomously and with absolute measurements without the need to first map and survey the land, and to be able to adjust to seasonal changes in the unstructured environment, which is how the concept of Simultaneous Localisation and Mapping is introduced. Alongside the priori method, two common approaches for mitigating this accumulation of error include particle filter and graph-based SLAM [31][28][29].

2.3.1 Simultaneous Localisation and Mapping (SLAM)

SLAM is when localisation is used at the same time as the mapping of the environment takes place [32][33]. SLAM is aimed at constructing a map of the environment surrounding a robot and simultaneously finding the robot's position and orientation in this global map [34]. Zou et al. [34] mention that SLAM techniques can be either visual-based or LiDAR-based. Interestingly, SLAM is susceptible to drift error, so loop closure by being able to recognize previously visited locations is being used to fix the accumulated error [35]. This links back to the point of previous papers [25][30]: that absolute measurements solve the issue of incremental or accumulated error.

Hess et al. [31] define Loop Closure using two-dimensional (2D)-LiDAR SLAM as a unique way of reducing computational resource requirements to compute loop closures [31]. They continue by stating that 2D loop closure using scan-to-map matching helps limit accumulated error, as well as demonstrating how the technique has resulted in the mapping of a large number of square meters and simultaneously sending those results to the operator, fully optimised and in real-time [31]. The SLAM 'algorithm' consists of a front and back end: the front end being data association and feature tracking, while the back end is responsible for the map estimation [33][36]. The back end also performs optimization to reduce error [36] and can provide feedback to the front end for loop closure [33].

2.4 Use of Light Detection and Ranging (LiDAR) for Localisation

LiDAR makes use of the time for a laser pulse to travel between the sensor and the target, and back to the sensor. The basic principle of LiDAR, outlined by Dubayah and Drake, is that information about the surroundings (the distance or range to the sensed objects) is calculated by measuring the time it takes for a pulse to travel to the objects and back to the sensor [37]. The laser pulse sent out is usually near-infrared for applications involving vegetation [37]. According to Lefsky et al., [38], LiDAR sensors directly measure the vertical distribution of vegetation material, and Dubayah and Drake [37] solidify this by mentioning this measurement of vertical vegetation distribution as a major strength of LiDAR

remote sensing. Additionally, three important characteristics of LiDAR sensors which categorise them are mentioned as [37]:

1. Whether they are range-recording or digitising the whole return signal.
2. The size of their footprints.
3. Their scanning pattern.
4. Their sampling rate.

Applied to forestry specifically, small-footprint LiDAR sensors are not ideal as they over-sample and miss the full height of canopies, and by using large-footprint LiDAR (digitises the complete return signal of the laser pulse and records a waveform that is related to the vertical distribution of the canopy structure), multiple measurements can be taken - relationships between which can also be used to model many of the other forest characteristics which are not directly recorded by LiDAR [37]. LiDAR applied to forestry provides an accurate baseline for applying LiDAR to unstructured environments generally. Some drawbacks of LiDAR for remote sensing in unstructured environments are found to include that: LiDAR wave-forms are more sensitive to structural changes through natural growth [38], measurements can be restricted by clouds and dense atmospheric haze (as they can attenuate the signal), there are limited datasets, and LiDAR data requires expert data processing [37].

Zou et al. [34] outline how LiDAR's measurements are generally of higher precision, are more stable in the presence of fast movement, and are more capable of dealing with any change in light intensity when compared to traditional cameras – and it is stated that the performance of LiDAR localisation can be more 'robust' in indoor environments. The limitation of this is that it only applies to indoor environments which are synonymous with structured ones. Another paper refers to the usefulness and popularity of LiDAR for localisation due to its superior precision, coverage, and longevity [39]. Physical LiDAR technology comes in two distinct forms: mechanical and solid-state. Mechanical LiDAR (M-LiDAR) is physically large and cumbersome with moving parts, whereas Solid-State LiDAR (SS-LiDAR) is based on a microprocessor only [39], and according to Nam and Gon-Woo [40], SS-LiDAR is not susceptible to mechanical noise (vibration) and interference that M-LiDAR is – and is cheaper by forgoing spinning elements. The drawback of SS-LiDAR is that it has a significantly reduced field of vision (FOV) when compared to M-LiDAR [40].

A paper by Ren et al. [41] details the use of Global Navigation Satellite System (GNSS) in conjunction with 2D LiDAR data for the navigation of robots, as the weak GNSS signals' informational stability can be improved by LiDAR sensing data, where LiDAR data was used as the input data to the SLAM when indoors, and the main navigation and localisation outdoors takes place using GNSS. The advantage of this is having a dual medium for data collection to carry out SLAM, however, the applicability to this project is minimal, and more informational, due to the use of mainly GNSS outdoors. The use of 2D LiDAR for localisation purposes brings about the necessity to differentiate between two and three-dimensional (2D&3D) LiDAR. 2D LiDAR sensors use a single axis of laser beams to record two dimensions of parameters, namely 'X' and 'Y' – and 3D LiDAR has an extra axis, for which data is collected using multiple lasers at different angles and projections [39]. 2D LiDAR is also limited to a

certain rotational range, however, 3D LiDAR can sense through a full 360-degree revolution [39]. 3D LiDAR, also known as 'multichannel LiDAR', can additionally provide significantly more information on point clouds [36] which provides information used to detect the elevation, slope, and texture and obstacles in the environment [22], specifically applicable to the unstructured environments that this project is concerned with. The point clouds are also collections of particles, which are dealt with using particle filters and feature extraction.

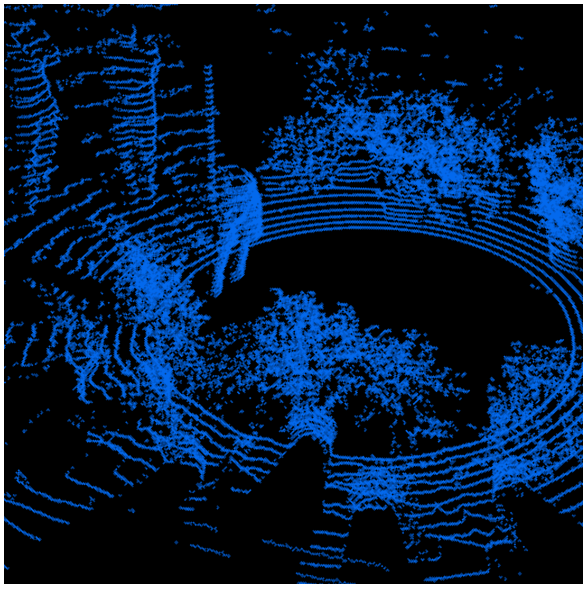


Figure 2.1: Sample point cloud data from the Husky Robot, courtesy of the African Robotics Unit [1]

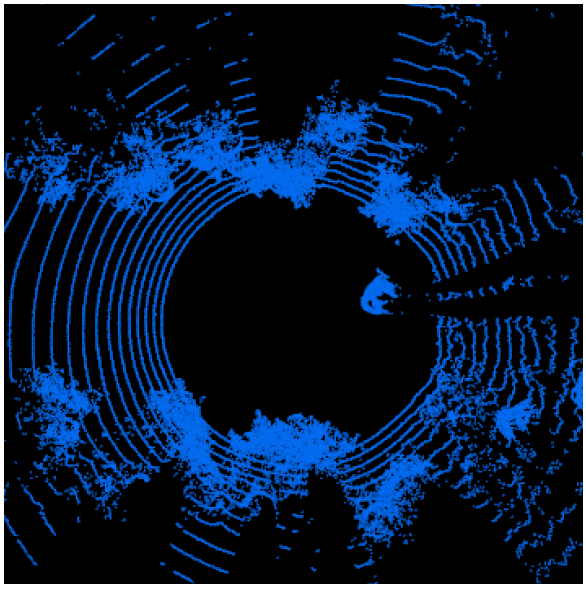


Figure 2.2: Bird's eye view of the sample point cloud data from the Husky Robot, courtesy of the African Robotics Unit [1]

In 2002, the predominant method for localisation was Bayes filters, which estimate the pose of the robot as well as the probability that this estimation is correct [42]. Thrun introduces the algorithm known as particle filters, also referred to as Monte Carlo localisation [42]. Particle filters approximate and predict the past and current states, and over time, create a more reliable estimate of where the object is likely to go based on a probabilistic distribution, using Byaes filters [42]. Peel et al. [30] summarise this as how the robot gets a 'guess' location which is a particle in a set of particles that the robot believes to describe its environment. A similar approach for localisation that uses LiDAR was used in the paper by Peel et al. [30] called Adaptive Monte-Carlo Localisation (AMCL), and they mention how an occupancy map is required for this type of localisation.

An occupancy map is created by the sensors on the robot and uses a binary algorithm to determine in which cell the robot lies, where a drawback is that there is no indication of the probability that the position is true [30]. Another paper solidifies that an occupancy map or grid is used to map out an environment, where the probability of the robot occupying a cell in this grid is calculated for each cell and represented as an occupancy grid [43]. Occupancy maps make use of an assumption that causes high uncertainty where a measure of spatial context could assist [43]. The assumption is that the cells are separated to be independent from one another – which is incorrect because cells may not

be distributed randomly [43]. Occupancy maps use Gaussian processing to match a probability to training data, and then a prediction can be made for unseen areas using a Bayesian framework and a combination of a sigmoid prediction, and an interpolation function [43]. A disadvantage of these polar grid methods is that for 3D-LiDAR data, they require the LiDAR points to be projected onto a grid, which results in a loss of information [44].

Two widely implemented approaches to SLAM are one that uses a Kalman Filter and one using a Particle Filter, rather than a probability-based Bayesian method. The filter approaches are advantageous because of their simplicity but lack the ability to implement loop closure [45]. Another drawback of this approach is that it is usually applied to linear systems, but the majority of the applications are non-linear [45]. Uncertainty due to non-linearity can be fixed by applying an Extended Kalman Filter (EKF) to first make the problem linear [45]. Taheri and Xia [45] continue to explain the implementation of an EKF-based SLAM algorithm, of which only the first two steps are applicable to the localisation problem at hand – the rest apply to creating a map as the robot explores a new area. The first two steps are building an elementary map based on sensor information, and the characterisation of landmarks around the robot, as well as completing a pose estimation [45].

An alternative to occupancy maps, while similar, includes pose graphs, which are becoming increasingly popular as a solution to localisation. A pose graph is a collection of poses estimated using extracted features and connected using feature similarities and transformations [28]. This is synonymous with a feature map, which "represent[s] the environment by the global locations of parametric features (such as points and lines)" [25, p. 11]. The features sensed by the robot are then extracted and matched to relevant features on the map. Here, the importance of feature extraction is not to be overlooked and includes segmentation methods.

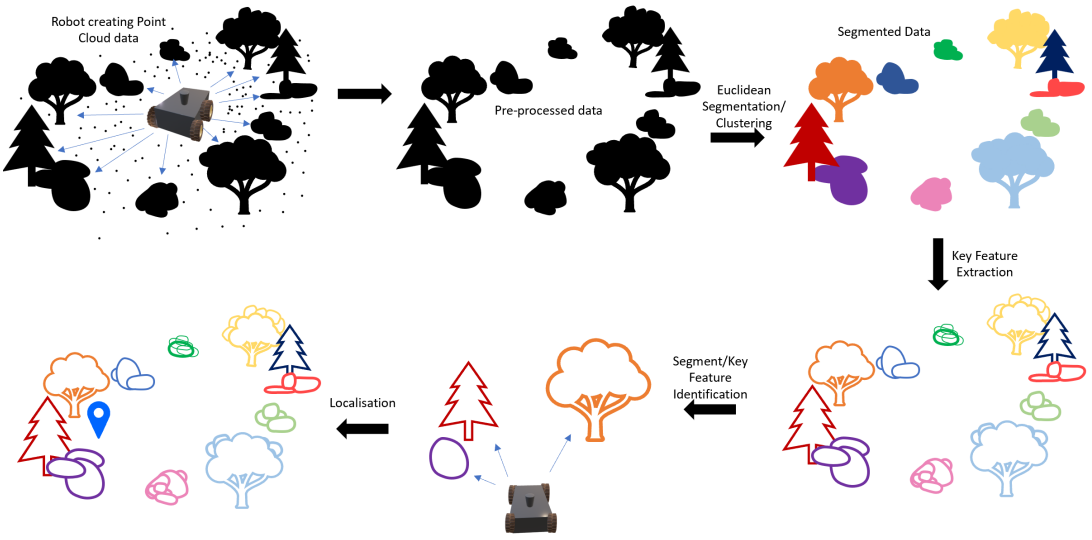


Figure 2.3: A summary of the process/algorithm used for LiDAR localisation

2.4.1 Segmentation Methods

Segmentation methods allow for the creation of point clouds so that feature extraction can be performed on collected data, and then matched to a mapping of the environment. A method called Colour Histograms is used for feature identification and segmentation on visual data [46], however, because this is not transferable to LiDAR, it was not explored further. In the earlier paper by Tinchev et al. [26], the method of localisation used, specifically with LiDAR, relied on segmentation as a precursor to feature extraction, specifically for 'natural data' [26, p. 8240]. They continue to describe their method by imposing the use of Natural Segmentation and Mapping (NSM): the use of Euclidean segmentation on data gathered from a 3D-LiDAR sensor to outline and separate individual natural objects in the point clouds [26], after which feature extraction is completed by using Oriented Key Pose Extraction (OKPE). OKPE incorporates the use of segmenting the source cloud (collection of input data), assigning a pose to each segment within the source cloud, and using this to help with pose recognition later on after feature matching has taken place. Similarly, Dong et al. [47] use a geometric segmentation method - which is synonymous with Euclidean segmentation, and they follow the same flow for pose estimation and localisation as Tinchev et al. [26] - going from segmentation, to feature matching, and using learned methods. The difference between the papers is that Dong et al. [47] use Monte-Carlo Localisation (outlined previously), whereas Tinchev et al. [26] make use of pre-filtered input clouds through a Progressive Morphological Filter (PMF).

A main disadvantage of the 'classical' approaches to localisation using LiDAR data is that there is a reliance on manually extracted features, which are not as effective as learned features [48][32]. Furthermore, Tinchev et al. [32] in a later paper discuss their approach using a deep learning method, which builds on their paper discussed above [26]. Because the methodology followed in the paper by Tinchev et al. [32] is carried out on a CPU, it is a fantastic example of SS-LiDAR combined with a learned approach to localisation. The paper moves through the same approach as the previous paper [26], however, the segmentation and segment matching are improved through learning a 'novel segment descriptor' [32, p. 3] and a measure of this descriptor's performance, as well as integrating probability into the pose estimation for robustness. Segmentation is highlighted to be important for the ability to complete feature maps, which in turn are necessary for the matching of a feature map to the pose estimation of a robot [32][26]. Segmentation quickly becomes a mathematical description of a method in machine learning.

Euclidean Segmentation

The use of Euclidean Segmentation (ES) methods is widely documented, and highly successful in a number of papers, and it has been chosen as the focal method for this paper because of this success in the research field so far. Point cloud data captured by LiDAR can be irregular and difficult to segment, and so doing efficient segmentation becomes difficult and computationally demanding [49]. The method creates groups of points within the point cloud, called clusters, based on the Euclidean distance between a singular point and all others - where the point is added to a cluster if it is within a certain distance threshold of other significant points, otherwise the point is added to a queue to be clustered later on [50]. Included in the ES provided by PCL is the capability to define the maximum and minimum

number of points that are in a cluster, and these are therefore user-defined and can constitute part of an experimental process, depending on the data being worked with [50]. ES and clustering was evaluated in [49] and compared to the classic Region Growing (RG) method for segmentation provided by PCL [15]. ES is faster and more efficient than RG for lower-density clusters but both have increasing running times as clusters to be evaluated grow larger [49].

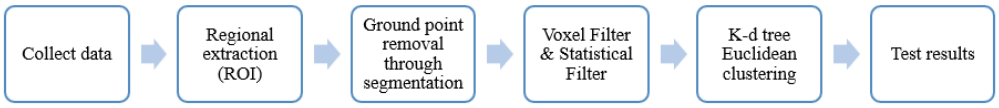


Figure 2.4: Generalisation of the Euclidean Segmentation Algorithm: Adapted from [2]

The ES algorithm follows the steps outlined in 2.4 where important aspects to note include that the data needing segmentation should have a defined Region of Interest (ROI), have the ground plane segmented out of the data, the data should then be filtered to remove outliers and then only can the ES take place [2]. The removal of the ground plane is important to remove any unwanted points in the point cloud, to reduce the processing requirements, and to enable enhanced visualisation of relevant segments after ES is performed [2] - see Figure 2.5. The ground points do not contribute to obstacle detection or segmentation of landmarks [2]. The above process (Figure 2.4) shows how the data needs a certain amount of pre-processing before ES can be run on it [2]. The purpose of identifying the ROI for the UGV that localisation is needed for is to reduce the amount of data that requires thorough processing and focus on a target area or object [2][51]. The process also relies on the data being structured and processed using a Kd-Tree implementation [52], which is further explored in Chapter 3. Ground plane segmentation does not always rely on ES, as there are many other algorithms, including the SMRF [5] and RANSAC [53] algorithms. The SMRF algorithm makes use of the aforementioned PMF by [54], however, they are differentiated by the SMRF having adaptations and improvements which help it perform better in more complex environments [5].

Filtering of the data to remove outliers and the ground plane has a similar effect to defining the ROI as it is an additional way of removing unwanted data, and outliers, and decreases the number of points in a point cloud to be processed [2][51]. Voxel grid sampling is a down-sampling method used to reduce the number of points but keep the shape and features of the point cloud constant [2]. Outlier removal relies on a statistical sampling method used by PCL which removes points that lie outside a standard Gaussian distance range from other points [55] [2].

Another interesting method used was to first apply the RANSAC algorithm to the data for noise fitting [56]. The algorithm is used to move any noise outliers to fit closer to the data [57][56]. RANSAC is often used for edge detection to reduce noise and optimise point cloud data, and forms part of a larger algorithm for edge detection in [57]. According to the originators of the algorithm, RANSAC fits a model to data that can be used for outlier detection or removal and can interpret or smooth any noisy data [53]. The methodology provided in [56] gives insight into how RANSAC can be used as an additional pre-filtering precaution to fit noise points and outliers closer to the point cloud data for better results in clustering and segmentation. Additionally in this paper, Euclidean clustering is compared to K-means clustering and DBSCAN, where Euclidean is slower, but overall remains an

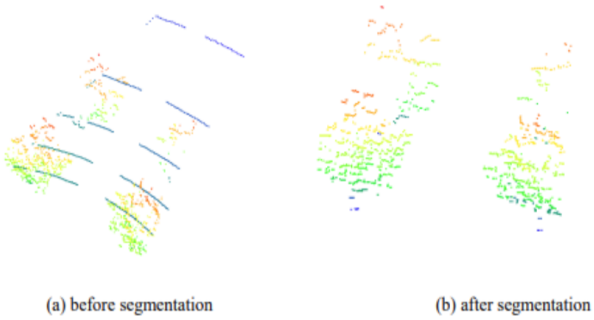


Figure 2.5: Figure Showing the Effect of Ground Plane Segmentation and Removal, taken from [2]

efficient and reliable clustering and segmentation algorithm [56].

Region Growing

As mentioned in the first paragraph of section 2.4.1, Region Growing is another type of method for clustering of LiDAR Point Cloud data, which is another geometry-based method, but not Euclidean-based [49]. Region Growing is used mainly for extraction of geometrical homogeneous surfaces, rather than the boundary detection discussed in [49]. A geometrical homogeneous surface is one for which there exists a transform between two points on the surface that does not change the appearance of the surface as a whole [58]. An assumption made for Region Growing is that objects are generally differentiable from one another visually, and this forms the basis for making use of the RANSAC algorithm to generate the points of importance, from which the region growing takes place [49]. RANSAC can additionally be used in plane segmentation, and when described in [56], the algorithm for plane segmentation is comparable to Region Growing.

Feature Extraction and Matching

To perform localisation on the segmented data, key features of the data first need to be identified, to provide a baseline map, as mentioned in Section 2.4.1. The robot can then perform loop-closure segment matching, and localisation by classifying the key segments it identifies as it travels into segments it has in its map. In the paper that this project follows closely, the authors' method for keypoint extraction makes use of segmentation and then appending an orientation description frame to each segment with a Gestalt descriptor [26]. In another similar work that describes multiple key feature extraction methods, the Gestalt descriptor was used in place recognition by computing it for dense regions within a point cloud, and utilising other algorithms with that data [59]. Bosse and Zlot [59] outline how the use of PCL to do feature descriptions based on surface normal estimates is unreliable for natural and unstructured environments. Contrasting to this, other sources suggest that computing surface normal estimates as part of the pre-processing step for 3D-LiDAR data can enhance the accuracy of (and forms the basis of) other algorithms [60].

The computation of surface normal estimates forms the basis for multiple keypoint extraction or feature description methods. These methods include Harris3D, available on PCL, Intrinsic Shape Signature (ISS), and range-based keypoint extraction, such as Normal Aligned Radial Feature (NARF) (which is also available on PCL [60]. Another feature descriptor that is used, based on information from the normals [3], is PFHs [60], which describe the local geometry of points in a 3D point cloud, in a robust

way [3] as seen in Figure 2.6. The way the PFH algorithm worked was further optimised to reduce the computational time for the rendering of PFHs but kept the descriptive quality - called Fast Point Feature Histograms (FPFH) [3]. The PFH functionality is available within the PCL's open-source capabilities [15].

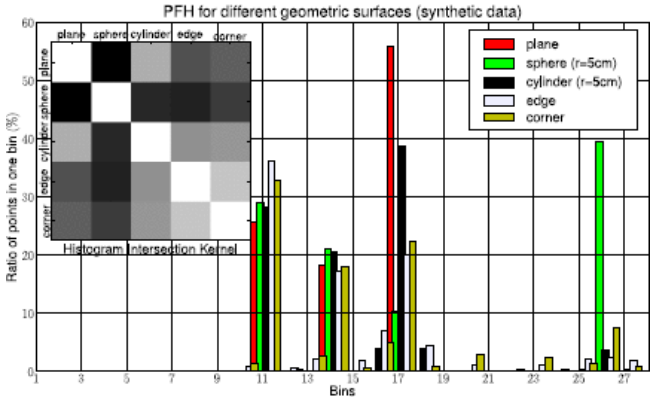


Figure 2.6: Example of PFH for points on primitive geometric surfaces, taken from [3]

2.5 Conclusion

Unstructured environments have introduced challenges that demand novel solutions, never seen before in commonly-tested indoor environments. The emergence of LiDAR-based navigation and localisation has enabled robots to interact with their environment more precisely. SLAM methods have further elevated the capabilities of robots by allowing them to create maps while localizing themselves. Segmentation methods have proven to be important in enhancing object recognition and feature mapping, paving the way for robots to interact with their surroundings more intelligently. While significant progress has been made, the broad challenges which can apply to any field of engineering research such as robustness, adaptability, and scalability, still apply. Real-time decision-making in dynamic environments, sensor fusion, and addressing accumulated error in localisation are challenges being addressed in this field. In conclusion, by using an amalgamation of new technologies and methods, including machine learning, probabilistic models or filter techniques, and LiDAR sensing, robotic localisation in unstructured environments becomes more tangible.

Chapter 3

Theoretical Development

As the project progressed, and different methods for point cloud processing needed to be evaluated or implemented, the processes, techniques, and algorithms all needed to be underpinned by a theoretical background for comprehensive understanding. An in-depth analysis of the methods required provided insight which was essential to effectively execute and explore the field of LiDAR point cloud processing, and autonomous navigation in unstructured environments.

This chapter focuses on the theoretical underpinnings of algorithms and processing methods used in the project in the different stages of design - it evaluates pre-processing methods such as filtering, ground plane segmentation, Euclidean clustering, and feature estimation, before delving into the theoretical foundations of the methods investigated for localisation and the validation of localisation accuracy.

3.1 Theoretical Development of Pre-Processing Techniques

3.1.1 Filtering Techniques

Point clouds can contain thousands of points and may have noisy or outlier points due to sensor and processing variations. Furthermore, measurement errors by these sensors can produce 'sparse outliers which corrupt the results even more' [55]. For the point clouds to be processed effectively in further methods, they must first be optimised and have the noise filtered out. Additionally, the downsampling of point clouds greatly reduces the amount of data needing processing in further steps and makes this further processing more efficient [2][51], as mentioned in the Literature Review. Different filtering techniques were discussed in Chapter 2, however, the ones focused on in this project are crop filtering to a ROI, Statistical Outlier Removal, and Voxel Grid Downsampling.

A cropping filter is applied to the data by defining X, Y, and Z coordinate limits, and removing any points in the point cloud which do not fall within those limits. Because this method is simple and was investigated by purely implementing it, it is discussed further in Section 4.3.2.

Statistical Outlier Removal

The Statistical Outlier Removal method is important in the applications of point cloud processing because sparse outliers can further complicate the estimation of geometric and Eigen-based features of the point cloud, which is elaborated on in Section 3.1.4. Taken directly from PCL's documentation, the description of statistical outlier removal is: '*sparse outlier removal is based on the computation of the distribution of point to neighbours distances in the input dataset. For each point, we compute the mean distance from it to all its neighbours. By assuming that the [resulting] distribution is Gaussian with a mean and a standard deviation, all points whose mean distances are outside an interval defined by the global distances mean and standard deviation can be considered outliers and trimmed from the dataset.*' The parameters that can be used for this method are [55]:

1. Mean K: The number of neighbours to consider when analysing each point.
2. Standard Deviation Threshold Multiplier: Multiplies the distance threshold between the point in question and other points. For a standard deviation multiplier of 2, points are considered inliers if they are within a distance of 2 standard deviations of the mean distance to the point in question.

As an example of this filtering, the PCL implementation of Statistical Outlier Removal [55] was applied to a random (cropped) point cloud in the 'Farm Dataset' [1], with a Mean K of 10, and a Standard Deviation Threshold Multiplier of 0.2. These values were chosen for the example because of the dense nature of the data (points are likely close together), and were simply chosen for demonstration purposes. The images the comparison of the filtered point cloud's inliers (magenta) and outliers (yellow). The

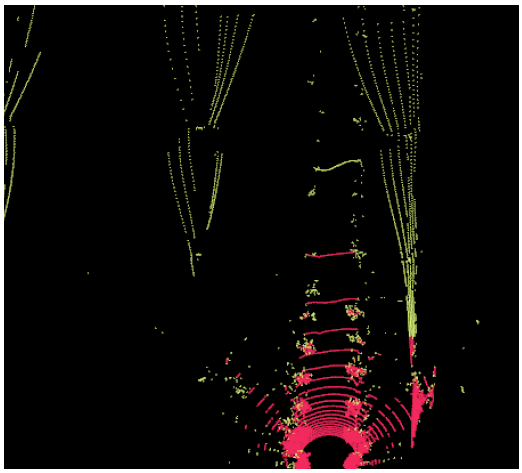


Figure 3.1: Image Showing the Comparison of Inliers and Outliers After Statistical Outlier Removal Filtering

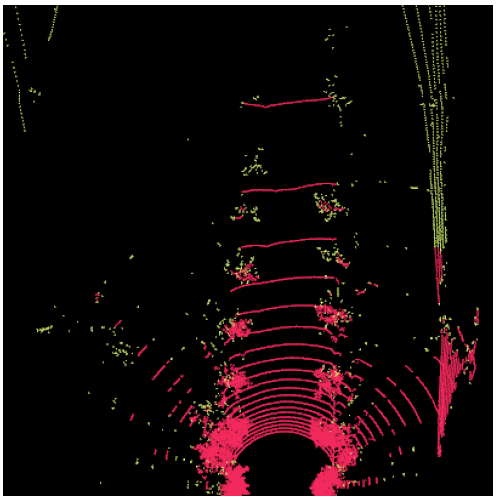


Figure 3.2: Image Showing a Closer View of the Application of Statistical Outlier Removal

figures above show the successful removal of points that are not within the dense regions of the point cloud. The filtering method is further investigated in Chapter 4.

Voxel Grid Downsampling

The whole premise on which downsampling is based is that of grouping small pieces of a whole into larger parts, which do no not change the way the whole looks, but rather just reduce the number of its constituent parts. This is a philosophical, psychological premise, called Gestalt Theory, (Gestalt, meaning 'shape' or 'form' in German): where humans perceive the individual parts of an element as a whole, and the shape of this part becomes an additional quality that we perceive [61]. One of the most well-known philosophers of Gestalt theory, Kurt Koffka, sums up the theory in the statement *'The whole is greater than the sum of its parts'* [62]. This means that the constituent parts of a whole are perceived independently from the whole itself - and applied to downsampling, if we make the constituent parts slightly larger without changing the fundamental makeup of the whole, the whole remains to be perceived as it was before [63].

Using a voxelised grid approach to downsampling is the method in which several points within a 3D voxel, or cuboid, are reduced to a single point [64]. In PCL, the points inside the voxel are represented by the centroid of this voxel [64]. In MATLAB, the points within each voxel are represented by the average of each contained point's X, Y, and Z coordinates, but this is essentially the same as the centroid - the only difference is that MATLAB takes the normals of the points into account for this average if they exist [65]. The parameter passed to each of these approaches is the size of the voxel grid - where the dimensions of each cuboid in the point cloud space can be defined. This is called either 'leaf size' in PCL or 'grid step'/'grid size' in MATLAB. For the implementation of this filter, the mathematics behind it is less important than the understanding of the approach, where the basic principle is grouping the points within a specific 3D box into the 'average point' to represent them. This is not to reduce the overall view of the point cloud, but rather simplify the processing of it.

3.1.2 Ground Plane Segmentation

Ground plane segmentation is a vital part of point cloud handling and can significantly reduce the amount of processing required [66]. By removing points not considered important for a task, such as localisation, the process can be simplified. For example, the ground plane in this project's specific application may lack important and identifiable features for localisation, so we can disregard it. By excluding a significant number of points collected in the ground plane, the data size for processing is minimised and computational resources can be saved: enhancing the processing pipeline to effectively become more efficient and less time-consuming. Some methods for ground plane segmentation are discussed in Chapter 2, however, the RANSAC and SMRF algorithms are the main focus of this project to be explored in this section.

RANSAC for Ground Plane Removal

Fischler and Bolles first introduced this algorithm in 1981 [53], and it has been used in many applications since it was published (see Section 2.4.1). The algorithm makes two assumptions (directly from [67]):

1. All the data being considered is made up of either inliers or outliers. '*Inliers can be explained by a model with a particular set of parameter values, while outliers do not fit that model in any circumstance.*'
2. '*A procedure which can optimally estimate the parameters of the chosen model from the data is available*'

The main workings of RANSAC are to iterate through subsets of the original data being passed into the algorithm as an input, and subsequently fit this input to a specific model. Points in the dataset are assumed inliers until proven otherwise: If points in this subset do not fit the model, they are defined as outliers [67]. The algorithm also calculates 'confidence parameters' [67]. The procedure for the testing of the subsets is as follows (directly from [67] [17]):

1. *Select a subset of the original data at random, and call it* ***hypothetical inliers****.*
2. *A model is fitted to the* ***hypothetical inliers****.*
3. *All of the original data points are tested against the fitted model. The points that fit the estimated model well (criteria defined by a model-specific loss function) are called the* ***consensus set****.*
4. *If sufficient points are classified as part of the* ***consensus set*** *then the estimated model is defined as reasonably good.*
5. *The model can be improved by re-estimating it using all members of the* ***consensus set****. The fitting quality (as a measure of how well the model fits the* ***consensus set****) is used to hone the model as the iterations continue. The model is evaluated according to the error of the inliers relative to the model.*

The pseudocode to illustrate the workings of the algorithm **??**, taken from [17] (and adapted for this text using [18]), is included in Appendix A.1 to elaborate on the above procedure and forms the base from which the code by PCL[15][67] used in Section 4 was built.

In the project-specific PCL implementation of RANSAC for ground plane segmentation and removal, the model used is planar, so the ground points can be fitted to a planar model. The number of maximum iterations as well as the distance threshold (which defines how close a point should be to the model to be considered an inlier) are parameters set in the code [14]. The planar segments of the point cloud are then presented, and the largest one (assumed as the ground) is removed from the point cloud data.

SMRF for Ground Plane Removal

The focus of this algorithm is on the MATLAB implementation function: '**segmentGroundSMRF**'. This algorithm consists of three stages: *Creation of a minimum elevation surface map from the point cloud data, segmentation of the surface map into ground and non-ground elements, and segmentation of the original point cloud data.*' [4]. The stages of the MATLAB implementation function are illustrated and outlined in the following graphic:

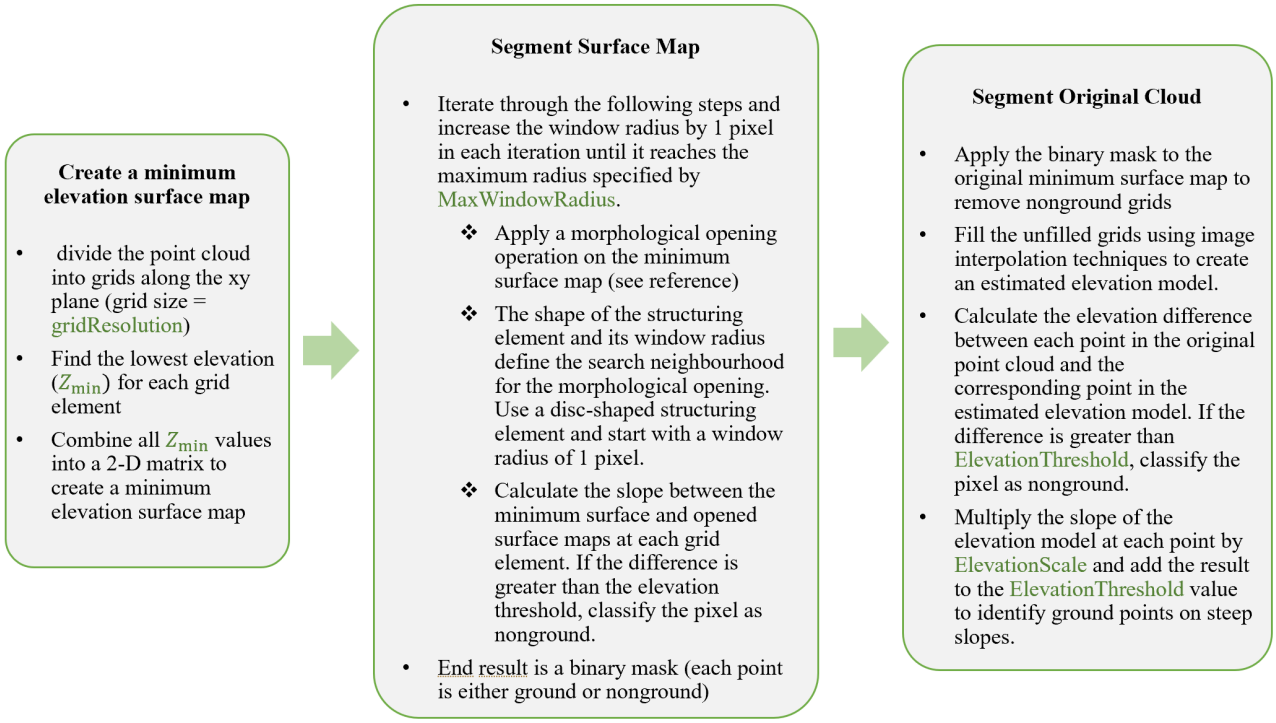


Figure 3.3: SMRF Algorithm Process (Created using the exact algorithm from [4], which cites [5])

The significant difference between PMFs and SMRFs is that SMRFs have been built on the same principle as PMFs, however, they have the ability to process more complex (and therefore more unstructured) terrains [5]. This is due to differences in the filtering techniques such as linearly changing the window size (window radius in the graphic) and controlling the elevation threshold with only one metric, rather than using multiple parameters to change the window size and elevation angle for each iteration [5][54]. Although Zhang's algorithm [54] uses a more complex approach with more variables, Pingel, Clarke, and McBride [5] found their SMRF algorithm to be more successful than it despite its comparative simplicity.

The improvement made on PMF in the implementation of SMRF was mainly in its simplicity, and in how it includes a 'slope dependent elevation threshold' to segment ground points from the provisional digital elevation model built by the filter [5]. The default '**MaxWindowRadius**' and '**ElevationThreshold**' parameters for the MATLAB implementation of the function are 18 and 0.5 respectively [4] - and the documentation specifies that the elevation threshold can be increased if more of an uneven ground terrain is to be included, and that increasing the maximum window radius of the function is likely to increase the computation time of the function.

3.1.3 Euclidean Clustering and Segmentation

The use of ES in this project is limited to the clustering of the point clouds. These point clouds needed to be clustered and segmented as an initial processing step for the implementation of the SegMatch algorithm (Section 3.2.2), to achieve localisation. Important topics being discussed in this section include crucial aspects of Euclidean Clustering and Segmentation such as the octree data structure, Kd-Tree structures for finding Nearest Neighbour (NN)s, and finally the algorithm itself.

It is also important to understand the difference between clustering and segmentation, as the words can be used interchangeably, but have subtleties that may infer otherwise. The difference understood through the implementation of each/both, is that clustering refers to the organisation of the point cloud into meaningful pieces that are classified as separate from one another, and segmentation refers to extracting those meaningful pieces. Both of them rely on Euclidean distance and end up with similar results, but their end goals slightly differ. In the context of this project, the terms will be used interchangeably (except when referring to ground plane segmentation in which the ground points are extracted from the original cloud).

Octree & Kd-Tree Structures

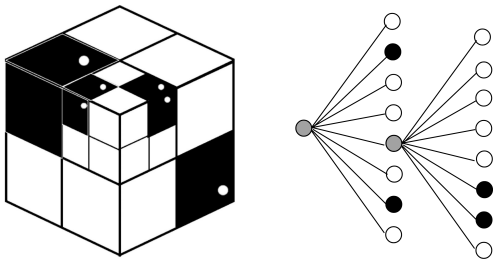


Figure 3.4: Image Showing the Graphical Representation of an Octree (Taken from [6])

The octree is a data structure which encapsulates and represents 3D data in a binary format, for the rendering of 3D images and scans on computers [68]. Each node in an octree is representative of a voxel (cuboid structure) in 3D space [68]. The constituents of an octree model are scene segmentation, encoding, and data representation [6]. The 3D image or object to be rendered is segmented into cuboids or voxels: similar to applying a voxel grid, but in the case of octrees, the goal is to represent the data and encode it, rather than downsample it.

With octree segmentation, a minimum cube size can be defined, so that the resolution of the data is adequate according to user/data specifications. This minimum cube length (size) is represented as d_{min} in the following equation, which shows the definition of an octree structure [6]:

$$\text{octree} = \begin{cases} v_i = \frac{1}{8}v_{i-1},\ d_i = \frac{1}{2}d_{i-1},\ h_i = h_{i-1}+1, & n > 2 \\ \text{end of separation}, & n \leq 2 \text{ or } d_i = d_{\min} \end{cases}$$

Where v_i, d_i, and h_i are the volume, side length and depth of the cube in the i^{th} division and n is the number of objects contained in the cube [6]. This structure can then be serialised into binary data (bytes) where 1 represents a filled node, and 0 represents an empty one [68].

The reason these structures are important for this application is to define how the data is structured for the NN search done when completing Euclidean clustering. The significance of the octree or Kd-Tree structures is in their efficiency for data storage and access, specifically in applications requiring a NN search or point cloud registration using ICP (Section 3.2.1) [68].

The octree's computational efficiency is comparable to the Kd-Tree structure, if not more efficient, due to its regular subdivisions - however, the octree is more susceptible to challenges when dealing with significant maximal distances and '[unfavourable] starting pose estimates' [68], making it the less suitable option for localisation in unstructured environments. It is nevertheless, still important to understand as the octree data structure is used in PCL, and for comparative purposes to the improved use of Kd-Tree structures. The octree is, therefore, the approach to take when volumetric descriptions of the data are needed [12], however, Kd-Tree structures are part of the more common and generalised approach.

In the applications considered for this project, the use of Kd-Tree structures is associated with NN searches and algorithms such as Euclidean clustering, feature extraction, and both ICP and NDT registration. Kd-Trees as a structure, and the associated algorithm, were first formally presented in 1977 [52]. Directly from the article [69]: '*A Kd-Tree is a binary tree that hierarchically subdivides k-dimensional space with hyperplanes orthogonal to the coordinate axes.*'. In other words, the Kd-Tree structure is organised by subdividing each level of the tree by drawing a hyperplane perpendicular to the axis of choice, and this axis changes for each iteration of the subdivision. For each iteration of the subdivision, the nodes are divided into two (binary). Explained in mathematical terms, a '*hyperplane* $\mathbf{V}$ *in a vector space* $\mathbf{W}$ *is any subspace such that* $\mathbf{W}/\mathbf{V}$ *is one-dimensional. Equivalently, a hyperplane is the linear transformation kernel of any nonzero linear map from the vector space to the underlying field.*' [70]. In simpler terms, a hyperplane is a plane or vector of one less dimension than the data which divides the data and is drawn perpendicular to the last division.

The algorithm defining the building of the Kd-Tree structure depicted to the right can be found in [7]. The images are sequential from left to right, showing each recursive subdivision using the hyperplanes, and ending with an image showing the tree structure for the example data. For this application, the idea that remains more of a focal point is that of the NN search using this structure.

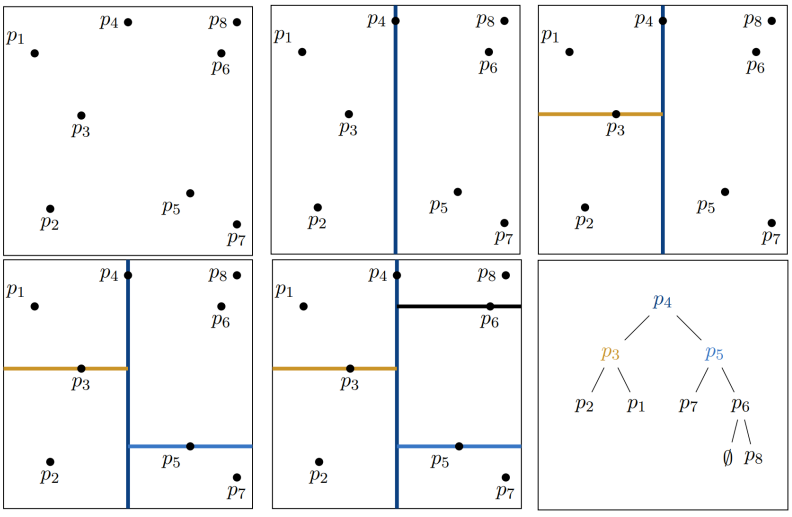


Figure 3.5: Image Showing the Graphical Representation of Recursive Kd-Tree Building (Taken from [7])

The key issue when designing the Kd-Tree building algorithm, according to [69] is how to choose where to place the dividing hyperplanes on each iteration. The original method proposed in [52] is to select a perpendicular plane so that the corresponding coordinate for that plane is the median of the largest data distribution [69].

Common alternatives for the hyperplane location selection are detailed in [69], but are not relevant for this application. The mathematical description, and algorithms for both building the structure around these medians and completing a NN search using Kd-Trees can be found in [7]. The more important algorithm concerning most applications in this project is the NN search using the Kd-Tree structure. This algorithm is used in the ICP registration, and importantly for the implementation of SegMatch, the ES procedure relies heavily on it. The algorithm (Figure **??**) [7] importantly defines how the search is conducted through the Kd-Tree and an informative animation of this can be found in [69].

The algorithm recursively moves through the branches, propagating the current best-found distance (to the point being searched) at the leaf nodes up through the branches [69][52]. The binary-search method allows the algorithm to quickly rule out points in space that are not needed (ie. all the distances are too far in that space, so it is ignored from a certain node, downwards) [69]. The psudocode for the algorithm can be found in Appendix A.2.

Clustering Methods in MATLAB and PCL

This Kd-Tree NN search algorithm is used in the ES methods in both PCL and MATLAB, and therefore is important to understand at a base level. In the PCL implementation, from the example [12], the method for ES followed is as follows [3]:

1. Create a Kd-Tree structure for the input point cloud dataset **P**, an empty list of clusters **C**, and a set of points that remain unchecked, and need to be: **Q**.
2. Set the Euclidean clustering parameters, and specify the search method as 'tree', which employs the above algorithm.
3. Cluster the points in the point clouds and store them in individual cloud files. For every point $\mathbf{p_i}$ in the point cloud dataset **P**:
   (a) search for the set of point neighbours in a specified radius (in 3D, so a sphere radius).
   (b) for every neighbour within the sphere radius, check if the point has been processed, and if not, add it the the queue **Q**.
   (c) When all points in **Q** have been processed using a Kd-Tree NN search, add **Q** to **C**, and empty **Q**.
   (d) Algorithm terminates when all points in **P** have been processed and are in **C**.

The parameters mentioned above include Cluster Tolerance, Minimum Cluster Size, and Maximum Cluster Size. The cluster tolerance needs to be adjusted according to the data being processed, and [12] specify that if it is too low, the result can be the clusters are too small - so an object can be represented by multiple clusters - and if it is too high, then multiple objects could become part of one cluster. In MATLAB, the algorithm differs slightly, and it takes in different parameters. The parameters used by

the ES function called '**segmentLidarData**', besides the input point cloud, are an angle threshold, and a distance threshold [9], and additionally can include a minimum cluster size specified as a number of points. The slightly different algorithm implemented by the '**segmentLidarData**' function is a relatively new one developed by Bogoslavskyi and Stachniss in 2017 [8]. They propose adding distance and angle thresholds when computing the ES. The following figures illustrate the purpose of the angle threshold: it allows the user to define the angle at which points are no longer part of a cluster.

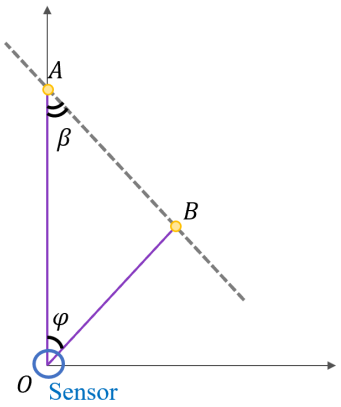


Figure 3.6: A Mathematical Diagram of the Workings of the Angle Threshold Parameter, Adapted from [8]

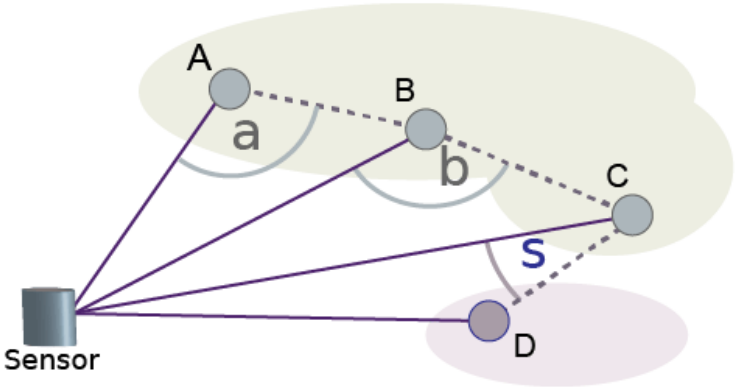


Figure 3.7: An Illustration of Using the Angle Threshold to Segment Cloud Data, taken directly from [9]

As can be seen in the above figures, the angle β (in the left-hand figure) is the angle to which the user-defined angle threshold, θ is compared. If β is greater than θ, then the points between the laser beams OA and OB are defined as belonging to the same neighbourhood, and in the right-hand figure, if $\beta < \theta$ (angle s in the figure $< \theta$), then a new neighbourhood is defined. The algorithm for clustering, called labelling in this context, after the neighbourhoods have been defined - is based on taking advantage of the structure of the laser range image (not the same as a point cloud, but synonymous for the purposes of this explanation) and the defined angle threshold [8]. The algorithm then loops through the points within the neighbourhood, and if it encounters an unlabelled point, it starts a NN search beginning with the breadth for points around the query point, within the specified (Euclidean) distance threshold [8]. If the points in the search satisfy both the threshold criteria as well as the minimum number of points meeting the criteria, the query point can be labelled and the algorithm can begin with a new query point [8]. The algorithm also processes the points in order from the top left to the bottom right and can be found in Appendix A.3.

3.1.4 Feature Estimation and Extraction

After clusters have been identified in a point cloud, there need to be identifiable features extracted from these clusters in order to do a recognition-based localisation. In other words, we need to be able to extract information from these clusters which makes them identifiable, so if they are seen by the sensor again, the vehicle needing localisation recognises that it has seen these features before, and therefore finds its location within a map containing the clusters and their unique features. These features, in PCL, can be described and summarised by a PFH, or a Fast PFH which is discussed in the Literature

Review, but not applicable to this project.

Rusu [3][15] shared how using points in point clouds, and their cartesian coordinates alone, are not enough to compare two point clouds together - because between the times the point clouds were recorded, the environment could have completely changed. While using Euclidean distances should suffice to measure the points and make comparisons, the points could be on different surfaces completely and so when analysed in the context of their environment, can represent totally different factors [3]. This ambiguity can be alleviated by including point feature descriptors [3], which describe the point in the context of its environment, its neighbours, and other geometric features. This is where Kd-Tree NN searches become important, where the features are inferred for a point by also accounting for the K NNs of the point. Feature descriptors are categorised as acceptably 'good' if they can record the same surface characteristics when there is noise, rigid transformations or varying sampling density.

Normal Estimation and Boundary Computation

Rusu [3] and [15] mention that normals are a sufficient feature when taking nearest neighbours into account, meaning if you have the normal estimation of a point and the information of the points around it for a geometric context, the features are sufficiently identifiable between point clouds. This normal estimation is done using a built-in function in PCL, which in the command line can be done using '**pcl_normal_estimation inputcloud.pcd outputcloud.pcd (params)**'. The parameters are either '**-k**' or **-radius** - but cannot be both. This specifies whether the normal estimation makes use of the NN search for K points, or if it uses a radius search which is not considered in this scope. For the MATLAB implementation, the focus is not on normals as features, so it is not necessary to understand them further.

Although the normals may not be important for the end product in this instance, they were still used as features when computing the boundary estimations. The boundary estimation is done similarly to the normal estimation in the command line and is explained further in Chapter 4.

Eigenvalue-Based Features

In MATLAB, Eigenvalue-based features encapsulate the feature description of points in point clouds, in a geometrical representation [71]. These eigenfeatures are explored in depth in the paper [16] referenced by [71] and it can be referred to for the mathematical Eigenvalue descriptions. These Eigenvalues are normalised ones e_1, e_2 and e_3, and form the base for the extraction of a feature set with 8 members - additionally, 6 members are added to the feature set for representing the local neighbourhood [16]. The Eigenvalue-based 3D features describe linearity, planarity, scattering, omnivariance, anisotropy, Eigenentropy, the sum of eigenvalues, and the change of curvature [16]. The Eigen-based features need not be elaborated on due to the nature of them being hidden within MATLAB functions, however their calculations are all done using the three eigenvaues mentioned earlier, and can be seen in the following table taken from [16]:

The MATLAB function to calculate and store these features in an 'eigenFeature' object is '**extractEigenFeatures**' [71] and extracts the features to use in feature matching, explored in later sections.

Feature	Expression
Linearity	$L_\lambda = \frac{e_1 - e_2}{e_1}$
Planarity	$P_\lambda = \frac{e_2 - e_3}{e_1}$
Scattering	$S_\lambda = \frac{e_3}{e_1}$
Omnivariance	$O_\lambda = \sqrt[3]{e_1 e_2 e_3}$
Anisotropy	$A_\lambda = \frac{e_1 - e_3}{e_1}$
Eigenentropy	$E_\lambda = -\sum_{i=1}^{3} e_i \ln(e_i)$
Sum of eigenvalues	$\Sigma_\lambda = e_1 + e_2 + e_3$
Change of curvature	$C_\lambda = \frac{e_3}{e_1 + e_2 + e_3}$

Table 3.1: Eigenvalue-based 3D features (Table adapted from [16])

3.2 Localisation Frameworks & their Foundational Theory

3.2.1 ICP Algorithm

The ICP algorithm proposed and described by Besl and McKay [72] in 1992 works on multiple different types of geometric data, which includes a collection of points [72] and therefore can be applied to the larger scheme of point cloud processing algorithms used to this day. It is primarily used in this project for localisation, but also provides a reliable method for point cloud registration [15], for which this project used a different algorithm. A significant amount of mathematical principles behind this algorithm are described in [72] and further referenced to [73], but excluded from this paper for brevity. In summary, Besl and McKay [72] present principles including:

1. Point to Parametric Entity Distance: A measure of how close a singular point of interest is to the nearest parametric entity (like a parametric surface, line, or curve). This utilises a Newtonian Minimisation.

2. Point to Implicit Entity Distance: This measures the smallest distance between the point of interest and the closest edge of a non-bounded implicit entity when the point is not within the entity.

3. Corresponding Point Set Registration: A procedure using a quaternion-based algorithmic approach to find 'the least squares rotation and translation' [72, p. 242]. Quaternions are numbers of a hypercomplex form that have 4 parts, encompassing both the angle and axis of rotation of an entity - and they are used extensively in computer graphics [74].

Algorithm Definition

The above principles are necessary to define the ICP algorithm, which is directly from the text [72], as follows: '*The distance metric* $\mathbf{d}$ *between an individual data point and a model shape* $\mathbf{X}$ *will be denoted*'

$$d(\vec{\mathbf{p}}, \mathbf{X}) = \min_{\vec{\mathbf{x}} \in \mathbf{X}} \|\vec{\mathbf{x}} - \vec{\mathbf{p}}\|. \tag{3.1}$$

'*The closest point in* $\mathbf{X}$ *that yields the minimum distance is denoted* $\mathbf{y}$ *such that* $d(\vec{\mathbf{p}}, \vec{\mathbf{y}}) = d(\vec{\mathbf{p}}, \mathbf{X})$*, where* $\vec{\mathbf{y}} \in \mathbf{X}$*. Note that computing the closest point is* $O(N_x)$ *worst case with expected cost* $\log(N_x)$*. When the closest point computation (from* $\vec{\mathbf{p}}$ *to* $\mathbf{X}$*) is performed for each point in* $\mathbf{P}$*, that process is worst case* $O(N_p N_x)$*. Let* $\mathbf{Y}$ *denote the resulting set of closest points, and let* $\mathbf{C}$ *be the closest point*

operator:'

$$\mathbf{Y} = \mathbf{C}(\mathbf{P}, \mathbf{X}). \tag{3.2}$$

'*Given the resultant corresponding point set* $\mathbf{Y}$*, the least squares registration is computed as described above:*'

$$(\vec{\mathbf{q}}, \mathbf{d}) = \mathbf{Q}(\mathbf{P}, \mathbf{Y}). \tag{3.3}$$

'*The positions of the data shape point set are then updated via*' $\mathbf{P} = \vec{\mathbf{q}}(\mathbf{P})$. [72, p. 243]

Again, directly from the text [72], the algorithm is described:

'*The point set* $\mathbf{P}$ *with* N_p *points* $\{\vec{p}_i\}$ *from the data shape and the model shape* $\mathbf{X}$ *(with* N_x *supporting geometric primitives: points, lines, or triangles) are given.*'

'*The iteration is initialized by setting* $\mathbf{P}_0 = \mathbf{P}$, $\vec{q}_0 = [1, 0, 0, 0, 0, 0]^t$ and $k = 0$. *The registration vectors are defined relative to the initial data set* $\mathbf{P}_0$ *so that the final registration represents the complete transformation. Steps 1, 2, 3, and 4 are applied until convergence within a tolerance* τ*. The computational cost of each operation is given in brackets.*'

a. *Compute the closest points:* $\mathbf{Y}_k = \mathbf{C}(\mathbf{P}_k, \mathbf{X})$ (cost: $O(N_p N_x)$ worst case, $O(N_p \log N_x)$ average).

b. *Compute the registration:* $(\vec{q}_k, d_k) = \mathbf{Q}(\mathbf{P}, \mathbf{Y}_k)$ (cost: $O(N_p)$).

c. *Apply the registration:* $\mathbf{P}_{k+1} = \vec{q}_k(\mathbf{P}_0)$ (cost: $O(N_p)$).

d. *Terminate the iteration when the change in mean-square error falls below a preset threshold* $\tau > 0$ *specifying the desired precision of the registration*': $d_k - d_{k+1} < \tau$.

The above was taken directly from [72].

3.2.2 SegMatch

Challenges are faced in the building of pose graphs or maps for localisation, such as accumulated drift error, and others more applicable to RGB scans. The authors of SegMatch [20] propose that their approach and algorithm, which uses feature descriptors and segment-based feature matching, can account for this drift error in map building, and produce a reliable localisation on the pre-built map. Due to the challenges faced in not having a Robot Operating System (ROS)-based system or environment, only applications in the context of MATLAB are explored and developed in this theory section.

Feature extraction is a primary process in SegMatch, which (in the MATLAB usage of it) uses the extraction of Eigen-based features, as elaborated on previously (3.1.4) to create feature vectors describing segments. These segments are clustered from a point cloud using the 'segmentLidarData' method, which was also previously expanded upon in Section 3.1.3. Importantly, the segments, their labels, and their corresponding feature vectors are then stored in a '*pcmapsegmatch*' object, which forms the map for localisation, and has properties that become assets in the SegMatch process:

1. *ViewIds*: read-only vector of View identifiers (M elements).
2. *Features*: read-only, N-element vector of '*eigenFeature*' objects.
3. *Segments*: read-only, N-element vector of '*pointCloud*' objects.
4. *SelectedSubmap*: read-only, with specified max and min coordinate limits.
5. *XLimits, YLimits, and ZLimits*: read-only, range of map along the respective axes based on centroid locations.
6. *CentroidDistance*: read-only, minimum distance between segment centroids for them to qualify as unique to be added to the map.

The object has many methods associated with it for map building, localisation, and visualisation of the map segments. The MATLAB terminal after typing 'help pcmapsegmatch' produces a list of the associated properties (in the list above, from MATLAB help [10]) and methods, and the screenshot in Figure 6.12b shows the methods and their descriptions [10]. The important methods are '**addView**',

```
pcmapsegmatch methods:
   Map Building:
      addView           - Add view with corresponding features and segments
      deleteView        - Delete views from the map
      findView          - Get indices that correspond to specific views for
                          features and segments selection
      hasView           - Check if views exist in the map
      deleteSegments    - Delete all segments in the map, but keep the
                          corresponding features
      findPose          - Find the absolute pose in the map that aligns
                          the segment matches
      updateMap         - Update the features' centroids and segment locations,
                          and remove possible duplicates in the map
   Localization:
      selectSubmap      - Select a submap within the map
      isInsideSubmap    - Check if a position is inside the current submap
      findPose          - Find the absolute pose in the map that aligns
                          the segment matches
   Visualization:
      show              - Visualize the map of segments
```

Figure 3.8: Screenshot of MATLAB 'pcmapsegmatch' Methods [10]

'**findPose**', and '**updateMap**'. '**findPose**' looks for segment matches between two consecutive views and returns a '*rigidtform3d*' object, specifying the alignment pose of the segment matches, which is an absolute pose within the map. It takes in a number of parameters, which were experimented with in the design phase but found to be inconclusive and irrelevant. '**addView**' adds a view, and its related features and segments to the map. This view is only added if the view is unique, which uses the '*CentroidDistance*' property to filter out features and segments that are duplicates.

Using loop closure detection is a reliable way to account for and remove the effects of accumulated drift error, and optimise the pose map [**?**]. Detection of loop closure is done by evaluating the '*rigidtform3d*' object returned by the '**findPose**' method in SegMatch map building - if it is empty, a loop closure is not detected, and if it contains a location, it means a loop closure was found.

NDT Registration

An important part of the SegMatch implementation is the use of the NDT Registration. It is used to find the first absolute pose before beginning with the SegMatch map building and to find the relative pose between successive scans fed into the map building loop, from which absolute poses can be computed using matrix manipulation of a previous absolute pose, combined with a current relative pose. The relative pose is found in relation to the previous absolute pose, and can then be used as a transformation to make a new absolute pose. The application of the NDT to point cloud registration is first described in [75].

The basic working behind this principle is explained in two dimensions, and when applied to 3D point clouds, is extended into a third dimension using a Voxel grid concept. For a full mathematical explanation behind the algorithm, refer to [75].

'The NDT models the distribution of all reconstructed 2D-Points of one laser scan by a collection of local normal distributions' [75, p. 2]. In 2D, the laser scan is first subdivided into cells of uniform size, but in 3D this would be equivalent to dividing the scan into uniform Voxels and each cell is then evaluated if it meets the criterion of consisting of more than 3 points [75]:

1. Collect all points: $x_{i,\ i=1,\dots,n}$ in the grid box (2D).
2. Calculate the mean: $q = \frac{1}{n}\sum_i x_i$.
3. Calculate the covariance matrix:

$$\Sigma = \frac{1}{n}\sum_i (x_i - q)(x_i - q)^T.$$

4. The probability of selecting or measuring a sample in the specified range (at the given x coordinate) is then given by a normal distribution function $N(q, \Sigma)$:

$$p(x) \sim \exp\left(-\frac{(x-q)^T\Sigma^{-1}(x-q)}{2}\right).$$

The above was adapted from [75]. The advantage of this method is there is now 'a piece-wise continuous

and differentiable description [of the plane (2D) or volume (3D)] in the form of a probability density' [75] function.

The use of NDT registration is primarily focused on scan alignment, which Biber and Straßer [75] outline meticulously as follows: The spatial mapping T between two consecutive scans or 'coordinate frames' is calculated as:
T :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

, where $(t_x,\, t_y)^t$ and ϕ are the translation and rotation between the scans/frames respectively.

To complete the alignment for recovering this spatial mapping, the algorithmic process is [75]:

1. Compute the NDT of the initial scan.
2. Initialise the first scan's parameters as estimations using odometry, or zero.
3. For the second scan, iterate through the samples and '*map the reconstructed 2D point into the coordinate frame of the first scan according to the parameters*'.
4. Find the normal distributions for each point.
5. Evaluate the distribution for each point and sum the result to obtain the score for the parameters.
6. Use one iteration of Newton's Algorithm to attempt to optimise the score, and the output is a new calculated parameter estimate.
7. Continue from step 3 until a predefined convergence criterion is met.

In the MATLAB function which does this registration, the Voxel size (for 3D data) can be passed as an input parameter alongside an initial rough estimate of the transformation ('*InitialPose*' as a name-value argument). Other parameters are possible to include, but not used within the scope of this project.

Chapter 4

Design & Implementation

The design for this project was iterative. Due to the nature of this project being developmental, in creating a data processing pipeline, the phases of design needed to be adjusted on each iteration of development. Firstly, the processing only occurred in C++, where the open-source PCL was used extensively. After encountering difficulties with the important open-source library for this project, 'SegMatch', the methodology had to be adapted to make use of the implementation of 'SegMatch' through MATLAB. Finally, the resulting pipeline was designed where the most efficient and applicable methods are combined and implemented.

This chapter is a guide through the exploration of the designs and implementations and concludes by outlining the steps taken to refine the design of the resulting processing pipeline.

4.1 Setup and Development Environment

4.1.1 Ubuntu

Ubuntu was used as the environment of choice for the integration of the external library, PCL [15], and its dependencies. The initial installation of these libraries proved a lot more cumbersome in Windows than in a Linux environment, so Ubuntu was installed as an additional environment on the Windows laptop.

4.1.2 Visual Studio Code and Use of C++ language

Visual Studio Code IDE was chosen for its ease of integration and use with the dual-platform choices (Ubuntu environment on a Windows machine). C++ was the language of choice because it is the language the PCL and its dependencies were originally written in, and it is a lot faster than many alternative languages/platforms such as Python or MATLAB because it is a compiled language, and Python/MATLAB are interpreted [76].

4.1.3 MATLAB

Later in the methodology, the use of MATLAB [10] was necessary for the completion of the processing pipeline, due to significant and time-consuming issues encountered with the compilation of C++ libraries and files. While MATLAB is conventionally slower than C++, its use saved a lot more time on the project by having all the tools needed for the project already accessible within the platform.

4.1.4 Combination of platforms

Towards the end of the project, both the Ubuntu environment with C++, as well as MATLAB on the Windows environment, were used to refine the LiDAR data processing pipeline. The majority of the pipeline exists in MATLAB, with PCL being used for only a few pre-processing steps.

4.2 Requirements and Specifications

4.2.1 Requirements

The processing pipeline is required:

1. To be fast and efficient.
2. To be robust and work for multiple input data sets.
3. To work for structured and unstructured environments.
4. To reliably localise a robot/UGV in unstructured environments using LiDAR point clouds.
5. To allow for simple integration into existing systems.

4.2.2 Specifications

1. The pipeline should complete localisation fast: as close to real-time as possible.
2. The processing should account for input point clouds of '.ply' and '.pcd' types, and the parameters for the processing should require minimal (if not zero) tuning.
3. The localisation should be qualitatively accurate relative to a built map for both indoor, structured environments and unstructured ones.
4. To be reliant on as few different pieces of software as possible.

4.3 Initial Design & PCL Investigations

4.3.1 Data Preparation

1. Load and convert: The data that needed processing within the designed pipeline was initially loaded in its original form as a polygon file ('.ply') and converted into the necessary point cloud data format ('.pcd') to enable further processing and analysis using PCL. The '.pcd' format was used for ease of integration and use with the rest of the PCL, especially when loading the clouds as objects into the C++ code. This was done using the PCL command line function: **pcl_ply2pcd input.ply output.pcd**. The function takes in an input cloud as a '.ply' and outputs a point cloud file with the '.pcd' extension. To make this possible for all files in a directory, the directory was looped through and the system terminal was interacted with using the 'filesystem' and 'system' standard C libraries respectively. The pseudocode for this is available in Appendix A.4.1, and because it holds such relevance to the project, an illustration of the flow of code is shown in Figure 4.3.

2. Verification: The first few data sets comprised of ones freely available on the internet, to ensure the PCL as an external library was installed correctly, and could be used as intended. Additionally, the viewing capability of this library was evaluated to ensure it was possible to load and view point clouds - with emphasis on '.pcd' files. The figures below show that the viewer was successful, when used to view point cloud data sourced from the internet, in '.pcd' format. The syntax to view the point clouds (which need to be in '.pcd' format) from the terminal is **pcl_viewer cloud.pcd**.

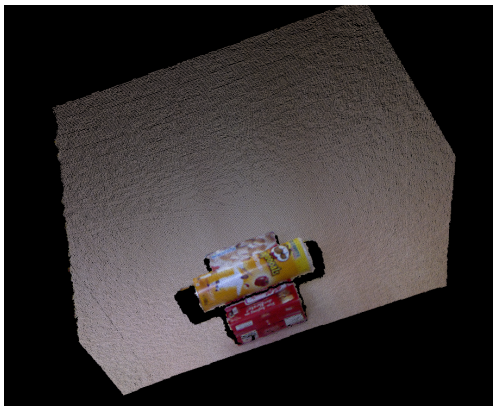


Figure 4.1: Test Point Cloud of Random Items (Data from [11])

Figure 4.2: A Different Test Point Cloud (Data Sourced from [12])

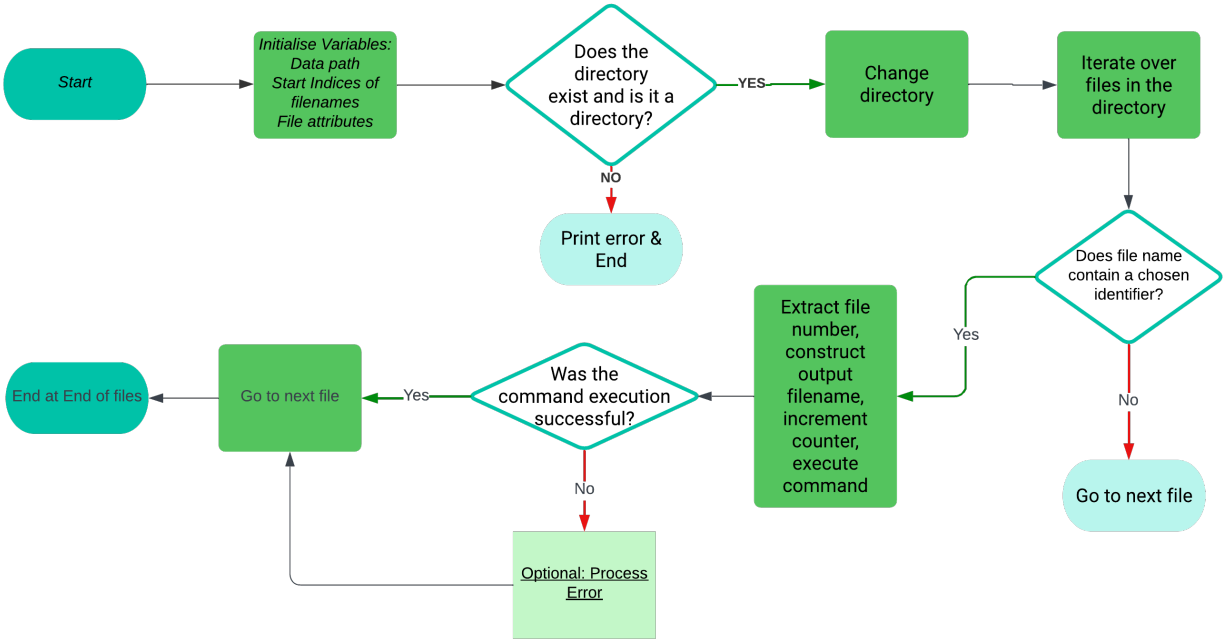


Figure 4.3: Figure Showing the Code Flow of Implementing the command 'pcl_ply2pcd' for Multiple Files in a Directory (figure created using [13])

4.3.2 Region of Interest

The dataset was cropped to focus on the defined ROI. To begin to define the ROI, a trial-and-error approach on the coordinate limits was conducted. The definition of the ROI in terms of requirements was it needed to include the front-facing view of the robot, restricted to a reasonable height above it. This statement is broad due to the nature of the data, in that the region of interest can differ for each data set or environment. The following process was adhered to:

1. Initially, a more analogue method for cropping a point cloud was investigated, where the PCL visualisation tool 'pcl_visualizer' was used to crop a singular cloud uniquely, and then that cropped cloud was used as a template to crop the rest of the clouds. Because precision in this discipline is so important, and the cropping method needs to be repeatable for different data sets, the more suitable method for using PCL to crop point clouds to a specified ROI is to be able to define the limits for each dimension.

2. The limits placed on the regions were defined by setting all lower limits to the smallest available number able to be defined, and the upper limits to the largest. These limits were set for each dimension: X, Y, and Z. The limits could then be changed to suit that of the data generated by the LiDAR sensor, or the needs of operators/navigation.

3. For the data set used in experimentation, the limits were defined (in metres) as:

$$X \in \mathbb{R},\ Y > 0,\ Z < 1.7$$

   It took a few attempts of cropping experimentally to find the correct cropping regions for this specific set of data. The intention was to find limits that can be applied to many data sets without the need to change them.

4. The C++ code to crop the point clouds was then made into an executable which iterated through all files in the specified directory and cropped them to the relevant ROI. This code can be seen in Appendix Item A.1, lines 30-38.

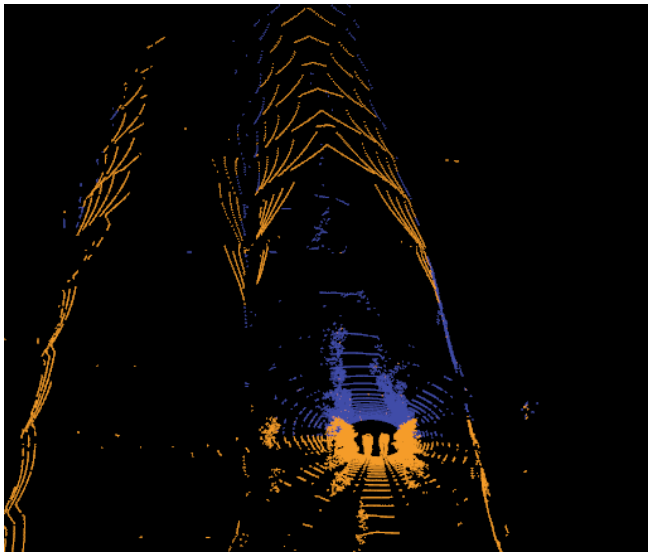


Figure 4.4: Figure Comparing the Cropped Point Cloud (Blue) to the Original (Orange)

4.3.3 Data Cleaning

Filtering techniques were applied to the data to make the further processing more efficient and robust. Part of the point clouds that could cause inefficiencies include outliers, noise, and the ground plane. This is because they do not form part of the important regions needing processing, for clustering, segmentation, feature extraction, or localisation, as mentioned in the Literature Review (Chapter 2) and the Theory Development (Chapter 3). Important filtering techniques used by PCL implementations include ground plane segmentation using RANSAC, and Voxel grid downsampling (refer to Sections 3.1.2 and 3.1.1 for a theoretical background to these methods).

Voxel Grid Downsampling

```
#include <pcl/filters/voxel_grid.h>
pcl::VoxelGrid<pcl::PointXYZ> vg;
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl::
↪ PointXYZ>);
vg.setInputCloud (cloud);
vg.setLeafSize (0.01f, 0.01f, 0.01f); //leaf size: 1cm
vg.filter (*cloud_filtered);
std::cout << "PointCloud after filtering has: " << cloud_filtered->size () << "
↪ data points." << std::endl;
```

Listing 4.1: Code to Downsample a Point Cloud

The code above (Listing 4.1) shows the implementation of the Voxel Grid downsampling within the PCL environment and was adapted from [64]. The leaf size can be determined by the user and needs to be adjusted to suit the sample data. The leaf size suggested by PCL is originally a $2.5cm$ cuboid, however, because the processing in this case relies heavily on details found in the unstructured regions, a smaller leaf size was considered to conserve more of the detail. The final adjusted leaf was a cuboid with an edge size of $1cm$.

Ground Plane Segmentation

The removal of the ground plane is important for removing a significant amount of points in the point cloud which are not necessary for processing, thereby saving time in not processing the points further. This is done in PCL using the RANSAC algorithm to identify a planar surface in the point cloud and remove the largest one. The code provided by PCL [67] [12] was integrated into a flow chart for ease of further implementation and understanding (Figure 4.5).

The technical background of the algorithm and ground plane segmentation can be found in Section 3.1.2. Important design considerations for this implementation are the choices of parameters. The maximum iterations and distance threshold were changed to 50 and larger values for the threshold, due to the unstructured, dense nature of the data, but more values were experimented with as seen in Figure 4.7 when the Euclidean Clustering was tested. Overall, the ground plane was always successfully segmented, and the ground plane segmentation parameters did not make a significant difference to the results, as the ground plane was always absent from the clustering results. The parameters were determined insignificant for this set of data. A smaller 'Max Iterations' was used to take up less

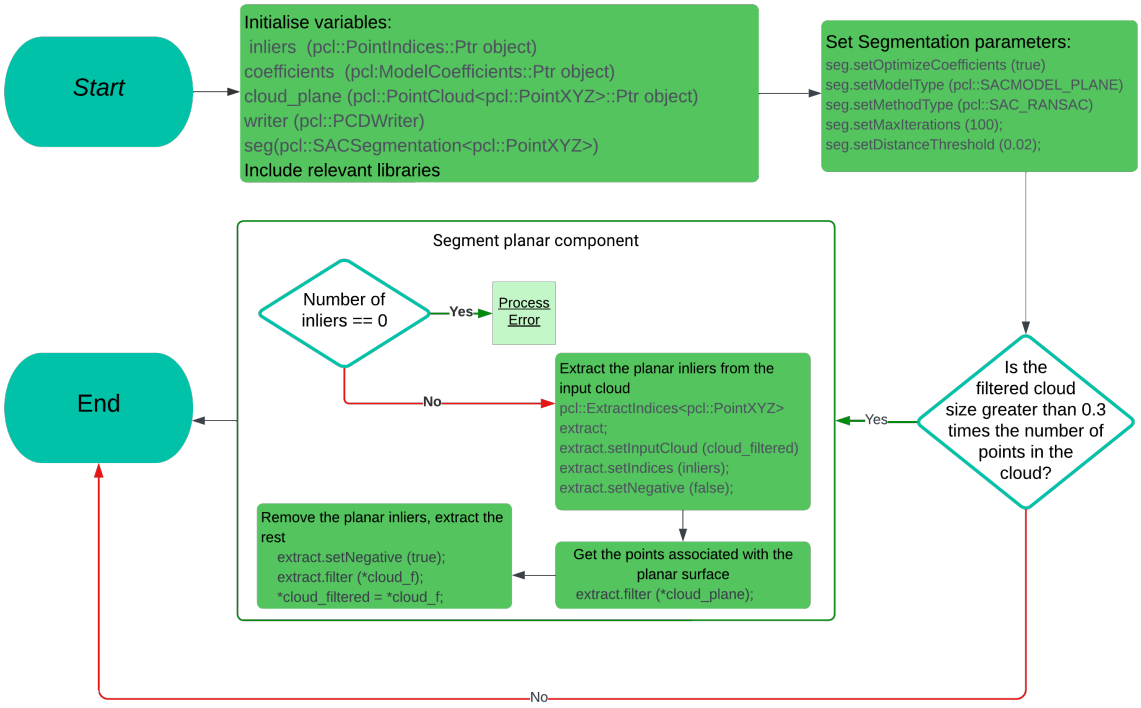


Figure 4.5: Figure Showing the Code [14] and Process of Segmenting the Ground Plane (figure created using [13])

computational effort in the tests, since the iterations made little to no difference.

4.3.4 Clustering and Segmentation

Euclidean Clustering

Euclidean clustering was enacted through the use of the built-in function of PCL. The functionality can be used on a singular file through the terminal, or it can be used within a C++ file on a number of files. It was also used to segment the ground plane from the rest of the clusters/points within the point cloud. The workings of the Euclidean clustering method from PCL are based on the mathematical principle of Euclidean distance, in combination with a Kd-tree structure and NN search principle. In this, the Euclidean distance is minimised for K points around another point, and if it satisfies a distance and minimum points criteria, it is attributed to a specific cluster - this is done for all points in a point cloud until all points have been processed.

The detailed workings of the algorithm used by PCL can be found in Section 3.1.3, and the implementation of the PCL code to accomplish this is tested with different parameters. These parameters include the Cluster Tolerance and minimum and maximum cluster sizes. The code for this segmentation can be found in Appendix A.4.3. The testing of the code against the sample data [1] shows how the clustering changes for the different parameters. The parameters used for each trial can be found in Table 4.1, and the visual, qualitative results can be seen in the 6-part Figure 4.7.

The first two images in the figure, (a) and (b), show two different views of the original point cloud being clustered (which was cropped to a smaller square region purely for experimental and demonstrative purposes). Subfigure (c) shows Trial 1, which has clearly defined clusters, however not enough detail

on the clusters and some of the unstructured objects, like the fruit trees, were not adequately captured. In Trial 2 (d), the change made was to decrease the minimum cluster size, to investigate if there were more clusters being formed that did not meet the size criterion. The image (d) shows how more clusters were represented, but the detail in the original clusters was not changed or improved. In Trial 3 (e) there was no change to the clustering parameters (the image shown is slightly zoomed outwards). The cloud remained the same even when the ground segmentation parameters changed - showing that the ground plane segmentation parameters have no effect on the cloud. For all the values tried up until this point the ground is successfully segmented from the clustered cloud. Trial 4 included an increase in the cluster tolerance, which added slight detail to the clusters but made a minimal difference, as most of the clusters remained unchanged and there was no improvement in the clustering of the fruit trees (unstructured clusters).

Trial	Max Iter.	Dist. Threshold	Cluster Tolerance	Min Cluster Size	Max Cluster Size
Default	100	0.02	0.2	100	25 000
1	50	0.1	0.2	100	25 000
2	50	0.5	0.2	50	25 000
3	50	0.05	0.2	50	25000
4	50	0.1	0.25	50	25 000

Table 4.1: Table Showing the Parameters Used to Trial the PCL Euclidean Clustering and RANSAC Segmentation, Demonstrated by Figure 4.7

Region Growing Investigation

Following the small success with the Euclidean Clustering, Region Growing was investigated, which showed minimal success with the clustering of the fruit trees. Region Growing required feature extraction using normal estimation, as it uses the normal of the points in the point clouds to be computed, but due to the lack of success with this implementation, the experimentation with it is not elaborated on further. The figure below shows how the raw point cloud has the ground plane effectively clustered, and the pots of the plants clustered correctly, however, the unstructured parts of the point cloud are not well-defined and clustered by region growing.

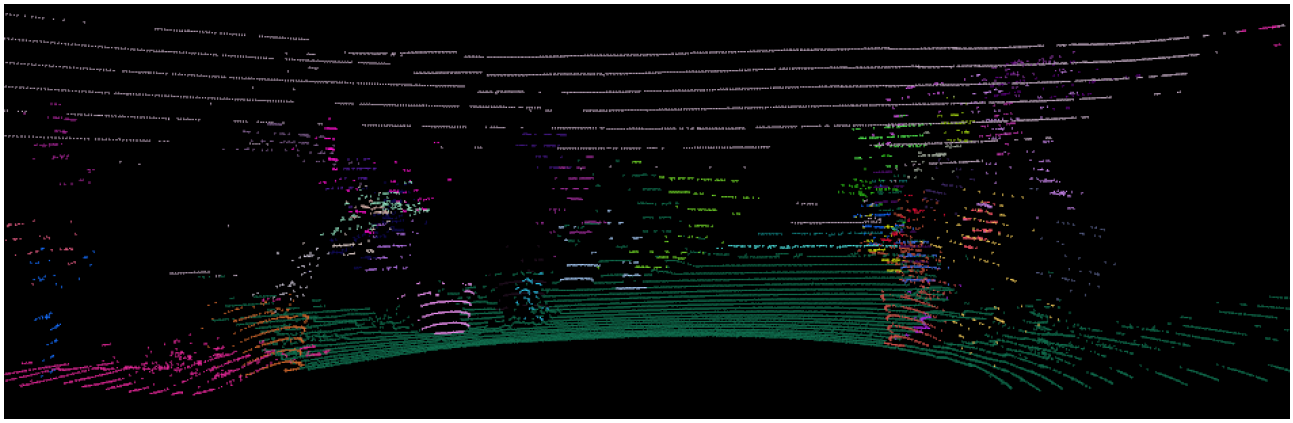
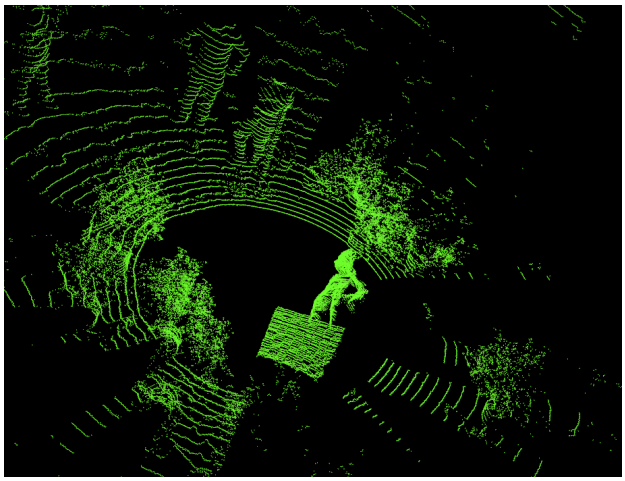
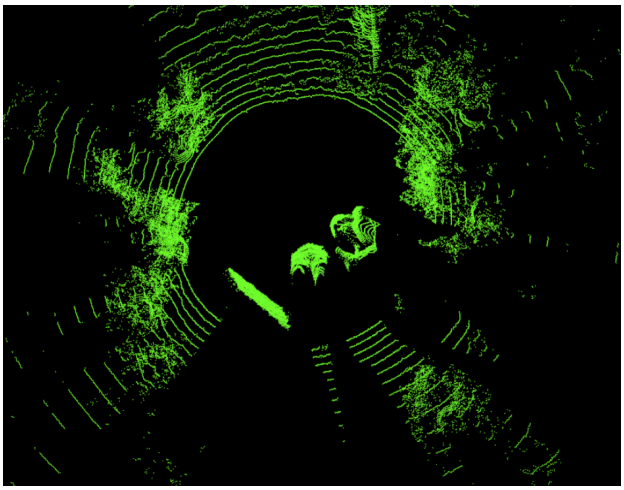


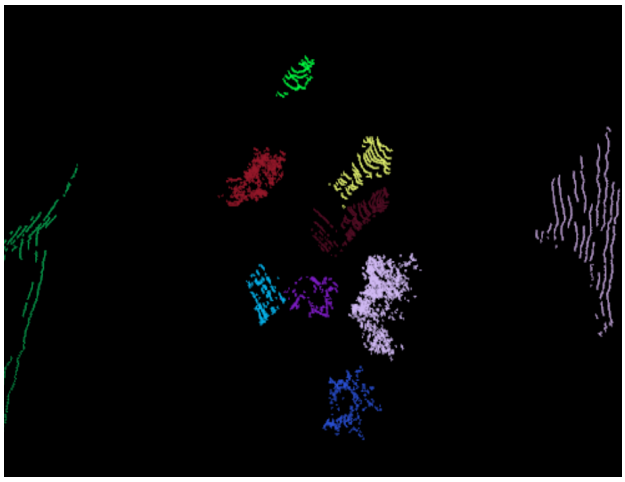
Figure 4.6: Figure Showing the Best Trial Implementation of the PCL Region Growing code [15]



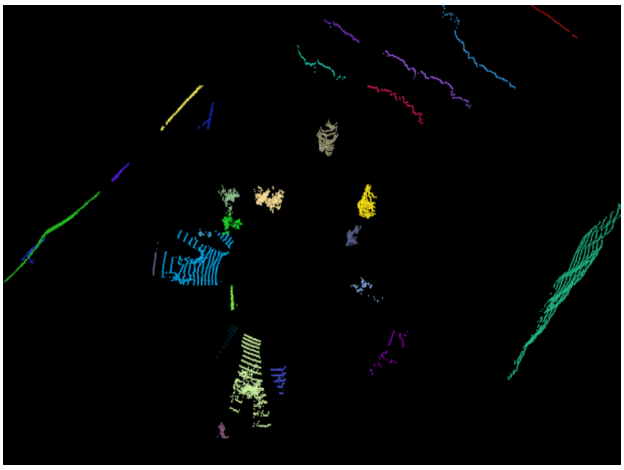
(a) Original



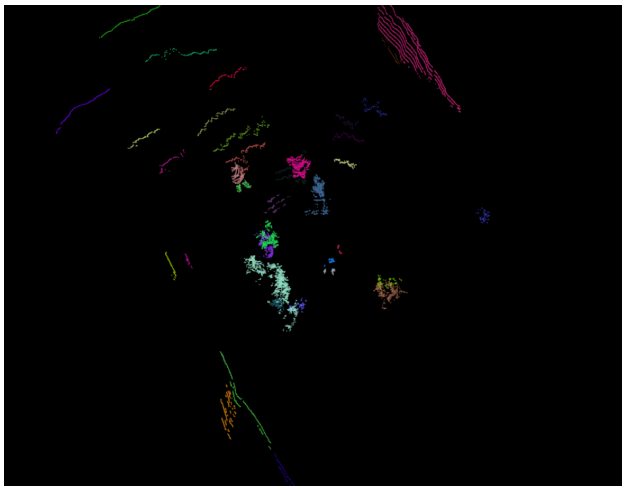
(b) Original, Top View



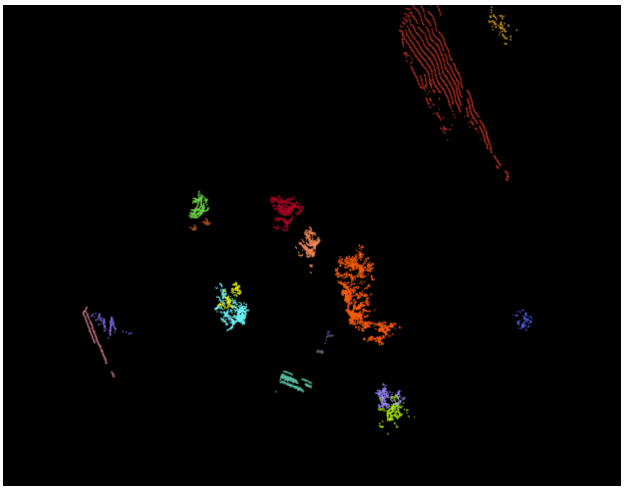
(c) Trial 1



(d) Trial 2



(e) Trial 3



(f) Trial 4

Figure 4.7: Figure Showing the Parameter Testing for Ground Plane Segmentation and Euclidean Clustering

4.3.5 Feature Extraction

Feature extraction was very much an experimental part of this implementation, where its effectiveness and use were evaluated to understand whether it made any difference to the localisation process. Surface normal estimation and boundary estimation were both done to visualise the feature extraction process and ensure it worked correctly, as intended. Feature extraction creates important information to feed into the SegMatch localisation algorithm, which can use the features to match two similar segments together, and identify them as 'matches' for the same location.

Normal Estimation

Normals are estimated as a way to extract geometric features from the point cloud data. The reason for estimating normals as features, and the process of doing so, is elaborated on in Chapter 3, Section 3.1.4. The above figure shows a representation of what happens to points in a point cloud that has its

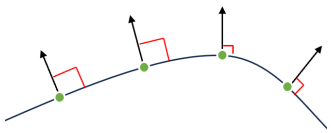


Figure 4.8: A Figurative Representation of the Normals

normals estimated, and an actual view of this can be seen in Figure 4.9.

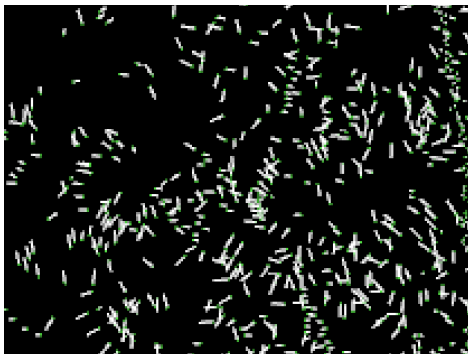


Figure 4.9: A Close-up View of the Estimated Normals for a Small Fruit Trees

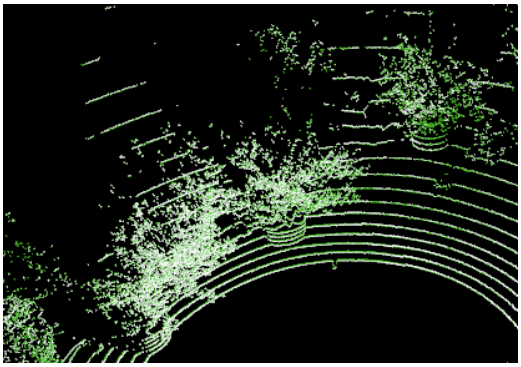


Figure 4.10: A Different View of the Normals Estimated

This estimation was done using PCL terminal commands, with a K parameter (Kd-Tree NN search) specified.

pcl_normal_estimation laserPCDcrop0.pcd fileWithNormals.pcd -k 10
pcl_viewer fileWithNormals.pcd – *Press 'L' in the viewer to view the boundaries*

Table 4.2: Terminal Commands for Normal Estimation and Viewing

Boundary Estimation

After Normal Estimation has taken place, the normals can be used to compute possible boundaries on objects. Boundary estimation relies on the normals having been computed, and since this is simply an investigation into its usefulness of feature extraction in PCL (and it was not used further), it is not elaborated on further, but has positive results. As can be seen in the below figures, the boundaries are effectively estimated for the sample data. This is representative of how extracting normals as features can be a powerful tool in the evaluation and processing of point cloud data. The figures below were created using the commands, with normals set to 1 to view, and a normal size (ns) of 0.1:

```
pcl_boundary_estimation fileWithNormals.pcd Output.pcd -k 200
pcl_viewer -normals 1 -ns 0.1
```

Table 4.3: Terminal Commands for Boundary Estimation and Viewing

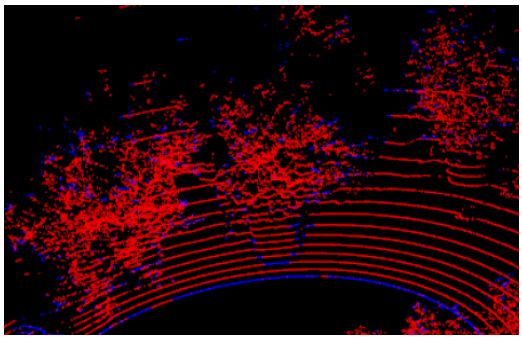


Figure 4.11: A View of the Boundaries (Blue) Estimated for Small Fruit Trees

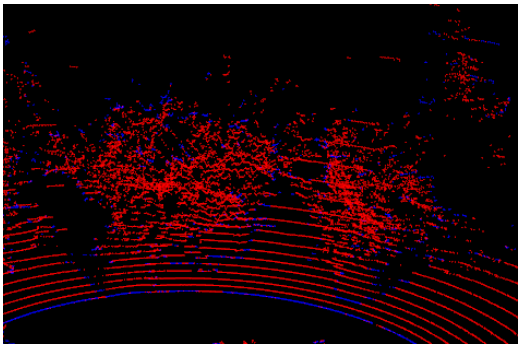


Figure 4.12: A Different Perspective of the Estimated Boundaries

4.3.6 Localisation

ICP Implementation

The successful implementation of ICP was dependent on ensuring all of the above functionality was correct, however, it did not rely on the clustering in Subsection 4.3.4. ICP relies on the surface normals computed, as well as its own iterative distance-based algorithm which computes differences between point clouds in the form of transformations to confirm if two point clouds match. The ICP algorithm was employed for localisation, with a focus on its efficiency and outcomes.

ICP is used extensively within this project and the first use of it was implemented using PCL. The testing of this enactment was done using code which can be found in Appendix A.4.4. The code uses ICP to find the transformation between an input cloud, and all other clouds within a specified directory. If the transformation matrix of a given file of the directory is closer to a 4x4 identity matrix (less of a transformation between the clouds) than the previous best, the file name for that point cloud is stored with the transformation matrix too.

To test the effectiveness of the algorithm and code, three different files within the directory were chosen at random, and set as the input clouds. The results of this test can be seen in Table 4.4.

For each run, the input cloud was different, and the matching cloud found and time to localise were recorded, as well as whether the localisation was successful (the input file name matched the cloud found with the smallest transformation by ICP). Figure 4.13 shows an example of the output of a run.

```
Locating your robot...
Best ICP Transformation Matrix:
           1  1.49012e-07 -7.45058e-09 -4.76837e-07
 1.78814e-07            1 -2.98023e-08  2.38419e-07
 -2.6077e-08 -4.47035e-08            1  1.49012e-07
           0            0            0            1
The estimated location is:
lasersMoving/lasers/croplaserPCD28.pcd
The actual location is:
croplaserPCD28.pcd
The location is correct.

>> Elapsed time: 28489.5 ms
```

Figure 4.13: An Example Output of a Single ICP Run

Original File	croplaserPCD28.pcd	croplaserPCD499.pcd	croplaserPCD658.pcd
ICP Match	croplaserPCD28.pcd	croplaserPCD499.pcd	croplaserPCD658.pcd
Correct?	Yes	Yes	Yes
Time (ms)	28489.5	30411.2	29318.5

Table 4.4: Table Showing the Results of ICP Localisation

Following this test, which had a 100% success rate, and an average localisation time of 29406.4ms, the hypothesis that ICP would be faster having features available was tested. This hypothesis could be tested by estimating the normals for the input point clouds, and evaluating if the ICP returned the correct point cloud as the 'location', and if it took less time to complete this localisation. The same point clouds used in the first test were used for this test, to account for any unexpected factors within other point clouds and to be able to compare the results to the benchmark created by the first test. The results of completing ICP localisation after estimating the normals (with a 'K' parameter of 10 - see Section 4.3.5) are presented in Table 4.5.

The results of this test show that the ICP localisation remained 100% correct for the point clouds with the estimated normals, however, the localisation was slightly slower, averaging a time of 29565.8ms to localise. This disproves the hypothesis that the estimation of normals would speed up the process, and shows that it has no effect on the localisation, besides making the process fractionally slower due to the algorithm having to compute more information: the normals add to the amount of information being processed. The negative result of these tests is that the localisation takes approximately 30 seconds for one location/input file, which is too long. The ideal localisation time is required to be closer to real-time, which is why SegMatch was investigated.

Original File	croplaserPCD28.pcd	croplaserPCD499.pcd	croplaserPCD658.pcd
Converted with Normals	trialFile1.pcd	trialFile2.pcd	trialFile3.pcd
ICP Match	croplaserPCD28.pcd	croplaserPCD499.pcd	croplaserPCD658.pcd
Correct?	Yes	Yes	Yes
Time (ms)	28567.0	30310.3	29820.0

Table 4.5: Table Showing the Results of ICP Localisation After Normal Estimation

Original (ms)	With Normals (ms)
29406.4	29565.8

Table 4.6: Table Showing the Average Time to Complete ICP Localisation, Before and After Normal Estimation

4.3.7 SegMatch

SegMatch is the important open-source library and algorithm for Segment Matching and localisation from which this project draws a lot of information and functionality, and was created by the authors of [20]. The implementation of this library was important for the project to work as intended, with the specified accuracy and time requirements. After a significant amount of time (approximately seven days) of trying to debug the installation of SegMatch, which did not recognise the installation of some dependencies on the local instance of Ubuntu, the decision was made to move to the MATLAB implementation [**?**][20] of the same library, by the original authors ([20]). The reason for the libraries not compiling correctly is because SegMatch was originally created for use on ROS based systems - and because the environment for this project was reliant on Ubuntu and C++, the integration was the opposite of seamless. This decision to change platforms was made due to the deadline for the project being mere weeks from this issue, the fact that the SegMatch implementation was not even working yet, and resource and time considerations for testing, however, future work should investigate the ROS-based realisation further.

The SegMatch library was still explored while the reasons for the compilation errors were investigated. The theory and algorithmic explanations are explained in-depth in Section 3.2.2. Key findings regarding the flow of code used in the library for localisation are demonstrated in Figure 4.14. The figure shows how the different parts of the process all mesh together to form a larger algorithm, from the input of point cloud files, to the localisation of an input cloud against a built map.

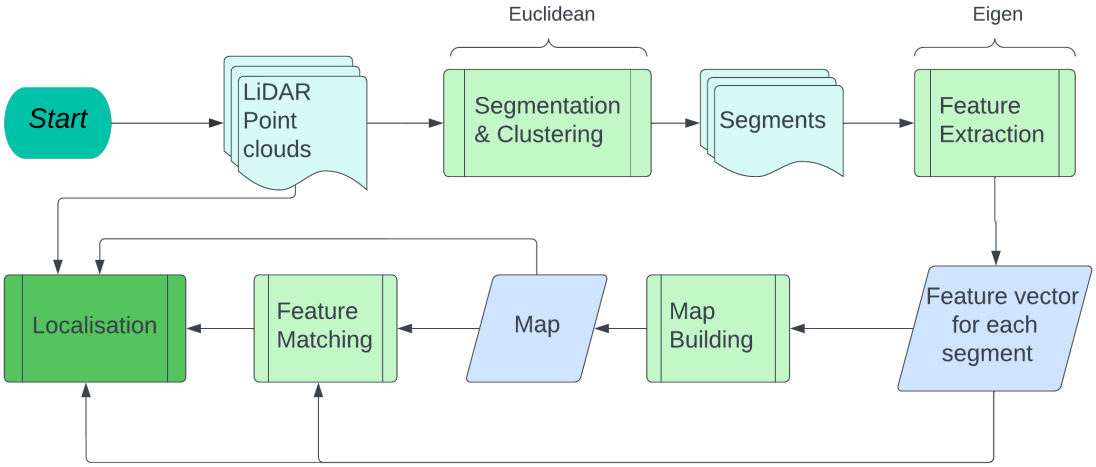


Figure 4.14: Figure Showing the Flow Outline of the SegMatch Process, made using [13]

Due to the extensive investigation into PCL, its methods, classes, and functionality - as well as that of SegMatch in the attempt to utilise it, the insights gained on the point cloud processing techniques were indispensable. The accumulation of in-depth knowledge of how the file conversions, filtering, clustering,

ICP, SegMatch, and the algorithms depended on (for all of the aforementioned functionality) worked and were integrated into a processing pipeline, was of utmost importance when moving through the next iteration of the methodology in Section 4.4 because it aided the coding process, and ensured that debugging any issues was not as time-consuming as that of the C++ issues faced in this method.

Additionally, in MATLAB, many of the methods can be used blindly without understanding their underlying structure or workings. This knowledge was necessary to understand the inner workings of the code in both MATLAB and C++ and gain insight on the real principles behind point cloud processing.

4.4 Adapted Design - MATLAB

Due to issues faced with the SegMatch library, being originally created for ROS-based systems (4.3.7), an article regarding the use of SegMatch within MATLAB inspired a shift in the approach. The two relevant examples found - from MATLAB [77][**?**] - were closely followed and implemented with the data, well known by ARU researchers as 'the Farm Data' [1]. Upon verifying that the examples worked as intended with data from the ARU, the examples were subsequently combined into the same MATLAB file, and major adjustments were made to suit the project: parameters were changed, code was added to account for specific differences in the data set provided by the example, and the set provided by the ARU, code was added to draw experimental metrics from the data processing pipeline, and the code was also adapted to suit the specific pipeline intended by the project.

4.4.1 Loading Files and Visualising the Data Set

Before any processing can be done on any of the point clouds, they need to be loaded into the MATLAB workspace and environment. This can be achieved for either '.pcd' or '.ply' files (unlike PCL which only accepts loading of '.pcd' files) through the use of the '**pcread**' function. An error was encountered when adapting the original code from [77] where the files in the target directory were being read into a cell array of point clouds in lexical order, not numerical order, so the files needed to be sorted into numerical order. The annotated code to do can be found in Appendix A.5.1, where the point cloud file directory and pattern are specified.

Viewing a Single Point Cloud

After the point clouds from the target directory have been loaded into the workspace, they can be easily accessed by indexing the '**pointClouds**' cell array. To view the properties of the point cloud at any index in the cell array, the code used was: **disp(pointClouds{k})**. To view the actual point cloud, two arrays of point clouds were created: a map set, 'ptCloudMap' and a localisation set, 'ptCloudLoc', using **vertcat** and any point cloud in the arrays could be indexed and shown using **pcshow(ptCloudMap(k))**, where k is a numeric index.

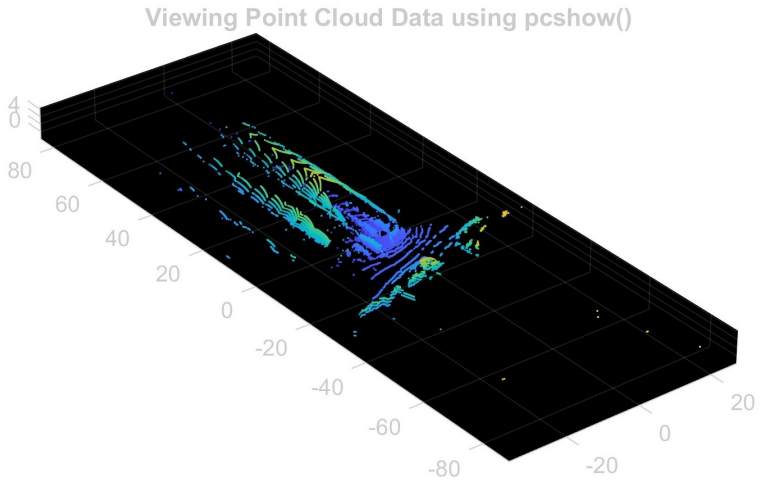


Figure 4.15: Figure Showing the Result of Using *pcshow* to View a Point Cloud in the Map Set

Viewing the Data Set in a Video

The important MATLAB functions used for this are pcplayer() and VideoWriter, but because this was not important for the end result, their workings are not further investigated, and the full code can be

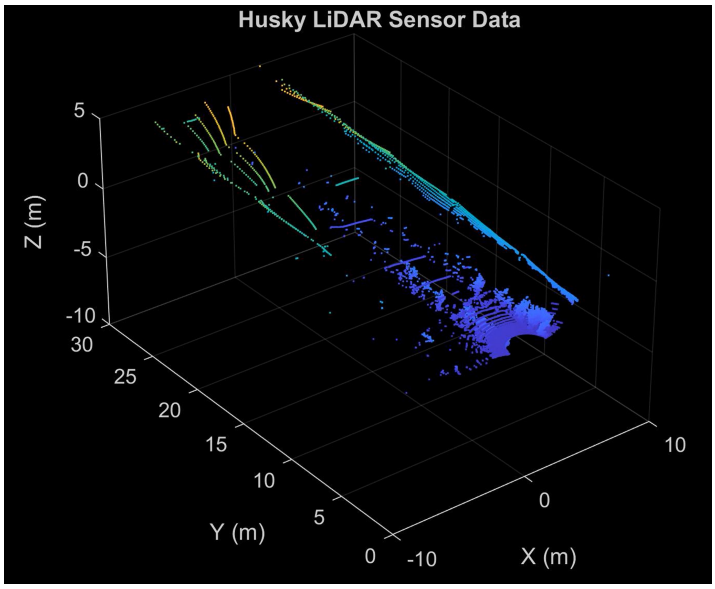


Figure 4.16: Figure Showing a Screenshot of Using pcplayer() and VideoWriter() to Create a Video of the Path Traversed

After proving that point clouds can be viewed, the entire data set of point clouds within the loaded directory, in numerical order, could be transformed into a video by treating each point cloud as a 'frame'. Limits could be specified to crop the view of the frames, either to zoom in or crop to a ROI manually for the video. In the fulfilment of creating a video, because there were many (1032) files in the 'Farm Data' [1], every second file was added to the video as a frame. The result of this was a video showing the view of the robot as it travelled along its path. This is useful to give the user a brief overview of (roughly) what the path of the UGV looked like from their desired perspective/ROI.

found in Appendix A.5.2, where the method to use pcplayer() to play the ordered point clouds was taken from [77], and the writing to a video and cropping the view was an original addition to the code.

4.4.2 Building a Pose Graph Using LiDAR Odometry

Pre-Processing

To begin the map-building process, all the LiDAR scans used to build this map needed to be pre-processed by undergoing ground plane segmentation and removal. This wass done using the 'PreProcess' method, which segments the ground plane using '**segmentGroundSMRF**', and removes the points outside the ego radius and within the cylinder radius (to account for the position of the sensor in the centre of the scan). The original method from the source [**?**] called 'helperProcessPointCloud' accomplishes the same task, however has additional code which checks if the point cloud is organised, and the majority of the code in the method was removed because none of the point cloud scans used in this scope were organised - and only need to be organised if they are being clustered. Appendix A.6 shows the function adjusted to pre-process the point cloud data.

The '**segmentGroundSMRF**' method uses a SMRF algorithm for ground plane removal, the detailed workings of which are described in Section 3.1.2. As part of this method, an elevation threshold can be passed as an input parameter. After some experimentation with the parameter at an original value of 0.05, it showed that the ground plane was not always completely excluded, especially when working with structured data which proved complicated. An increased final value of 0.5 was chosen for the elevation threshold and kept constant for the remainder of the tests, as uneven terrain needed to be factored in (refer to Section 3.1.2 for more detail and references).

Furthermore, the LiDAR scans needed to be downsampled *after* the ground plane had been removed. The reason it was done in this order is to not waste processing resources on downsampling ground plane points which will be removed. The algorithm imposed for downsampling makes use of a Voxel

grid. Voxel grid downsampling is a method which describes a 3-dimensional file by the centroids of cuboids (defined width) enclosing all data points. This method is explored thoroughly in Section 3.1.1, and is enacted using the '**pcdownsample**' method in MATLAB. For this method, a downsampling grid size was defined as $gridSize = 1.5$.

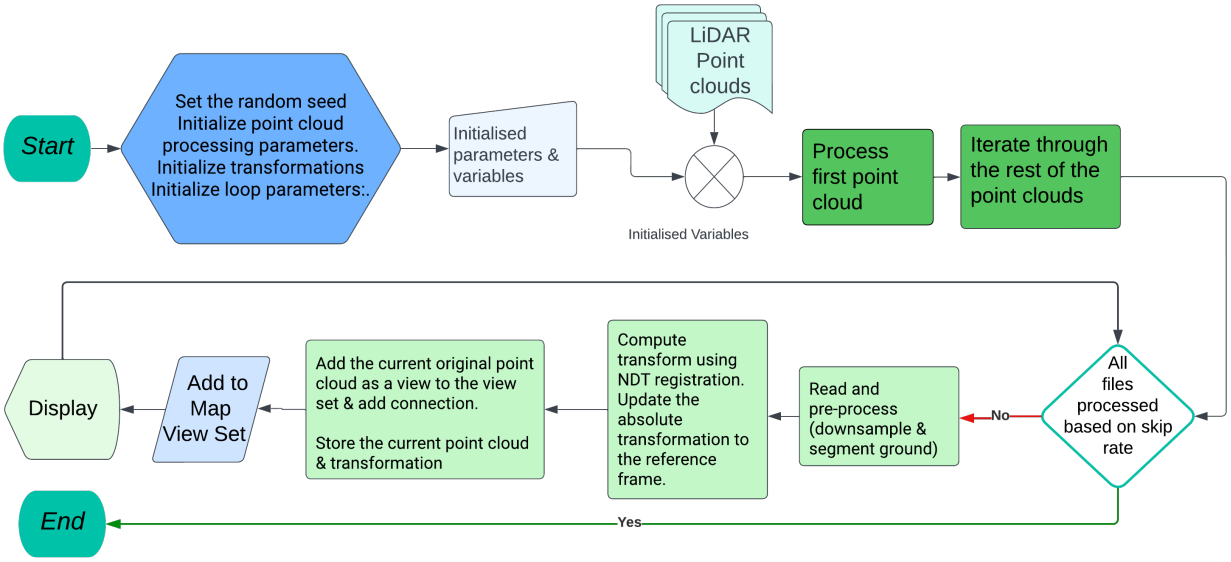
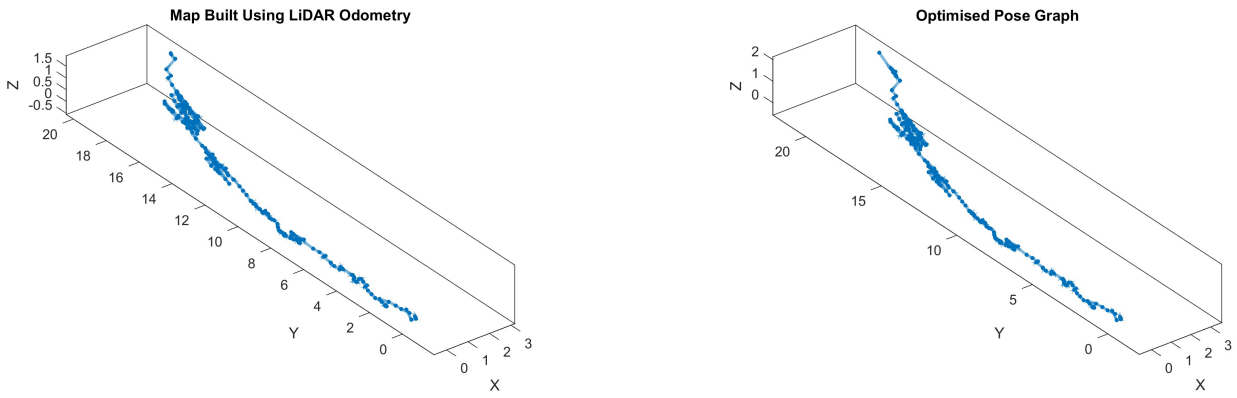


Figure 4.17: Flow Diagram of the Method to Build a Map Using LiDAR Odometry, made using [13]

Map Building

The method followed in the MATLAB Example [77] to build a map/pose graph using LiDAR data and SLAM was followed closely. The downsampling percentage was kept the same, but the Voxel size was decreased from 3 to 1.5 due to the data being unstructured, and needing to keep much of the detail intact. When the script was run with the grid size at 3, there were serious implications for the later stages of the processing pipeline which use feature extraction. The technique used in [77] is outlined in Figure 4.17, and the full code can be found in the source [77].

Following the building of the pose graph/map, loop closure detection (**detectLoop()**) could be added to the method, in conjunction with pose graph optimisation (**optimisePoseGraph()**), to account for accumulated drift error and poses which have noisy points. For this investigation purpose, the loop closure detection was left to investigate with SegMatch, and only the odometry building and optimisation were considered. Another excluded factor was the Inertial Navigation System (INS) data, as the point clouds in the data sets available did not include this, and it could not be further investigated whether this data makes a significant difference to map building using NDT registration, and is proposed for use in future work. When this code, excluding loop closure detection (found on the source web page [77]), was applied to the unstructured 'Farm Data', the resulting figures were produced - which shows the originally built pose graph (Figure 4.18a) using the method in Figure 4.17, alongside an optimised pose graph (Figure 4.18b). Upon comparing the two figures, the optimised pose graph shows how the nodes/poses were unchanged until the region of $Y \in [15; 20]m$ - which shows how the poses were unsuccessfully estimated in that region, as upon reviewing the video created in the steps from Section 4.4.1 the Husky only ever travelled in a positive Y direction, not back and forth as represented on the maps. After that region, the nodes are smoothed out by the optimisation. Overall, this method for map building shows errors in the Y direction and that the optimisation was not effective enough without the loop closure detection, and so using SegMatch methodologies such as segmentation, feature extraction, and matching to compute a map and loop closures is the ultimate goal, and is hypothesised to be more effective for pose graph building of unstructured environment



(a) (b)

Figure 4.18: Results of the Odometry Pose Graph Building and Optimisation

data.

4.4.3 SegMatch Implementation: Loop Closure Pose Graph & Map Building

This section details how the code from [**?**] was adapted and used for map building. The code made use of segmentation, feature extraction, and feature matching to complete loop closure detection and build a pose graph.

Pre-Processing

The same approach used in Section 4.4.2 to pre-process the data was adapted for use with the SegMatch flow. There were two key adaptations made:

1. The downsampling method changed from a Voxel grid method to a 'random' downsampling with a specified downsampling percentage (which specifies the percentage of the input that the function should return). This method is more efficient than the Voxel method if it is done before a registration task [78]. The downsampling percentage was tested in the experimentation phase and was settled at 50%, as the previous value of 10% held insufficient data and significantly reduced the accuracy of the map building and localisation.

```
ptCloud = PreProcess(ptCloud,egoRadius,cylinderRadius);
ptCloudDS = pcdownsample(ptCloud,'random',dsPercent);
```

Listing 4.2: Code Showing Function Implementations to Pre-Process and Downsample a Point Cloud

2. Prior to segmentation, the point cloud needed to be organised, meaning the data the point cloud file holds needed to be placed in a structure that is accepted by the segmentation function. This was done using a function which takes the type of sensor as an input parameter, as well as its horizontal resolution. The function is listed in Appendix A.7.

Segmentation & Feature Extraction

Segmentation and Feature Extraction, when isolated, are simple to do for a singular point cloud. To segment the data, the MATLAB function '**segmentLidarData**' is used, with three segmentation parameters: minimum number of clusters, distance threshold, and angle threshold. In the beginning stages, the thresholds were experimented with, however, they made minimal differences so the experimentation with those parameters need not be included in this scope. The original parameters as specified by the example code [**?**] are 150, 1 and 180 respectively. The final values settled on were 100, 1 and 180, to account for smaller clusters in the unstructured data, the same way this adjustment was made for the PCL ES (see Section 4.3.4). The code attributed to the implementation of this is in Appendix A.8.

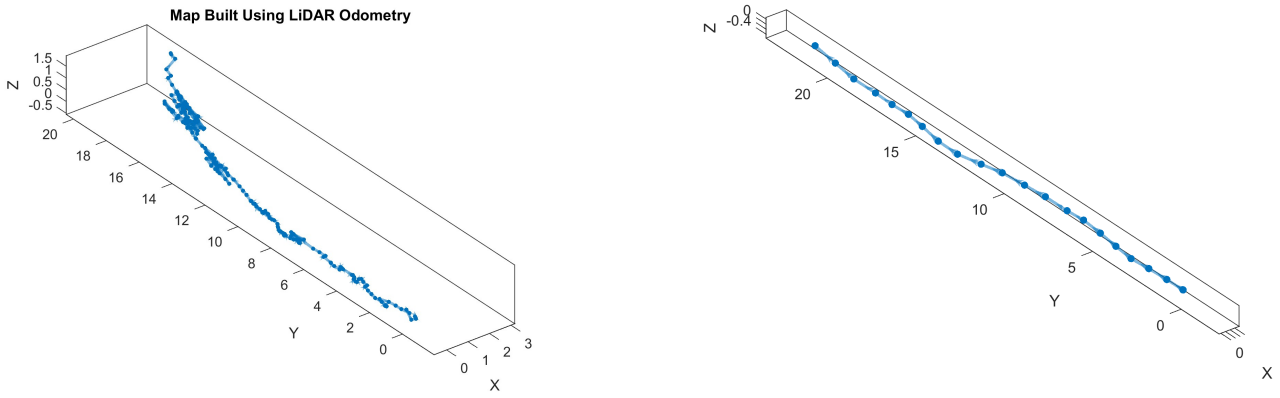
The feature extraction method in Appendix A.8, '**extractEigenFeatures**' is used to return an array of features for the segments labelled and stored in *labels1* and *labels2*. The method and the features it extracts are explored in more detail in the theoretical development, Section 3.1.4, but the basic working principle of it is to extract geometrical features from the point cloud segments based on operations of the points' Eigenvalues, and store those features in an '*eigenFeatures*' object for each cluster/segment.

Map Building, Segment Matching & Loop Closure Detection

Following the extraction of features and labels for the individual clusters/segments in a point cloud, the '*pcmapsegmatch*' object can be created and have these features and segments added to it as part of unique 'views'. The '*pcmapsegmatch*' object stores useful properties for the SegMatch process and forms the object used as a map. The underlying principles behind SegMatch are delved into in Section 3.2.2. The basic flow of the map building is:

1. Two consecutive point clouds are pre-processed and downsampled, and the NDT registration (see theoretical development, Section 3.2.2) between the two is used to find an initial relative pose estimate (the relative pose between the previous scan and the current one).
2. If the vehicle has moved a prescribed distance since the last map update, the segments and features of the point cloud (if unique) are computed and added to the SegMatch map using the SegMatch-specific '**addView**' and '**addConnection**' methods.
3. If there is a loop closure, the map can be optimised and updated using this loop closure, by matching the features to an existing pose/view within the map.

This is similar to building a map using LiDAR odometry, but the significant difference is that loop closure detection is used to eliminate drift error and maximise the benefits of using both local and global feature descriptors [20] - thereby creating a map of scans within the prescribed distance measure, rather than creating a map where each node is reliant on an un-verified transformation. Figure 4.19 shows the comparison between the two methods investigated and shows clearly how loop closure eliminates the errors encountered using purely registration for odometry.



(a) Map Built with Odometry

(b) Map Built with Segmatch Loop Closure Detection

Figure 4.19: Comparison of LiDAR Odometry and Loop Closure Map Building

4.4.4 SegMatch Implementation: Localisation of a Robot in a Known Map

SegMatch Localisation

For localisation, the position of a point cloud (or more specifically, its segments and features) on the pre-built map can be estimated using SegMatch's 'findPose' method, where the features and segments attributed to a specific point cloud are used as the input parameters, alongside other numerical parameters to optimise the process. The code for this is presented in Appendix A.9 as part of the final pipeline design. The important line of code is :

```
1    smLoc = findPose(sMap, features, 'MatchThreshold',30, 'MinNumInliers',3);
```

Listing 4.3: Code to Find the Absolute Pose Within a Pre-Built Map or submap (sMap)

The match threshold represents the percentage similarity between the feature matches, which was increased to 30 due to the data needing to account for unstructured environments, which may have features not matching within a small threshold. Additionally, the minimum number of inliers is reduced to 3, as increasing it increases the probability of returning a false positive location [**?**][10].

Validation Metrics

Initially, the validation metric used was to use a Euclidean distance-based minimisation on the distance between the estimated pose transformation, and the previous estimated pose transformation. This, after performing similar tests to the experiments done on the final pipeline, later proved to just be validating distances (with 50% accuracy), and there was no way to validate the actual position estimate.

The second validation attempt was done using a MATLAB version of ICP registration (see Section 3.2.1 for the theoretical background). The validation was done by minimising the root mean squared error of the transformation between the localised point and all other points on the map. Again, this returned a false representation of validation because the estimated pose was being validated against itself, and produced an accuracy of 88.96% when run on all 1032 files of the Farm data set. Additionally, this test took 14 hours to complete and was therefore ruled out due to its uselessness for validation and overly

computationally expensive implementation.

Lastly, the correction to the validation that was settled on was using a qualitative analysis of the map built using SegMatch, the distance from the map to the path found using NDT registration, and the result of 'findPose' as the SegMatch method for localisation. The map built using SegMatch is qualitatively and visually inspected to see how successful each was, and additionally, the poses found with both the SegMatch localisation and the NDT method were plotted against the map pose locations to see how effective each method was for localisation.

4.5 Streamlined Processing Pipeline

The full processing pipeline of code is included in Appendix A.9 due to the length of the code, however, the important design features are demonstrated in the flow chart in Figure 4.20, which is an updated flow from Figure 4.14. The majority of the pipeline was designed in Section 4.4, and makes use of all SegMatch methods described, from Section 4.4.3 up to this point. This is due to Segmatch map building using loop closure detection was more reliable and less susceptible to drift error, and the localisation being reliant on the more efficient NDT registration. The final pipeline ultimately includes the previously designed MATLAB SegMatch and NDT localisation, with the addition of analysis tools and data collection to plot graphs and informative metrics for performance evaluation.

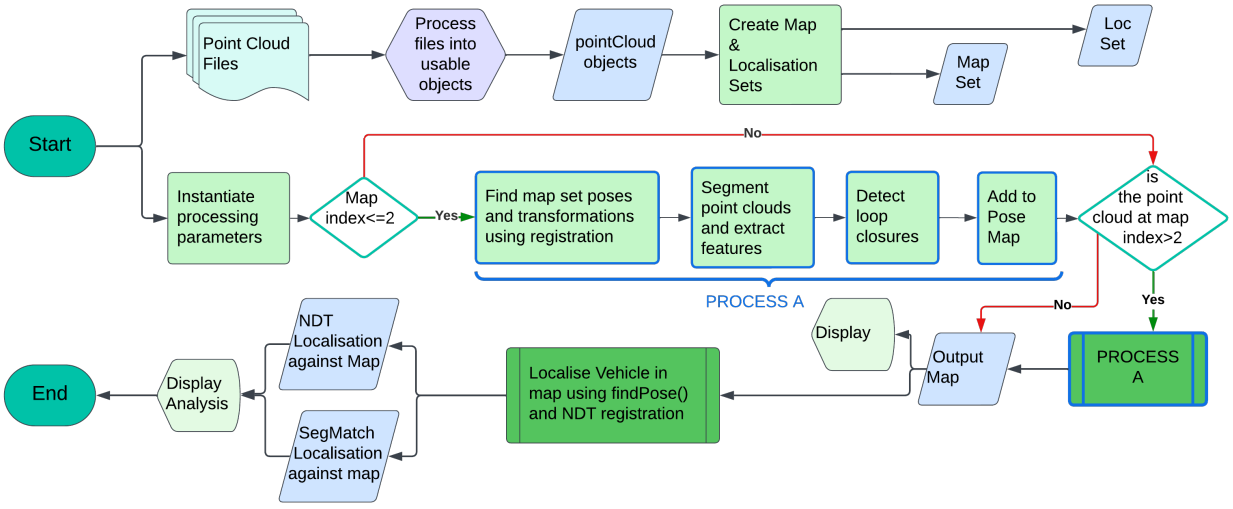


Figure 4.20: Final Streamlined Pipeline

Chapter 5

Experimentation

5.1 Experimental Outline

The experimentation conducted was to evaluate the effectiveness of SegMatch localisation on a map built using the SegMatch feature extraction, matching and loop closure detection, in comparison to the localisation on that map using NDT registration in diverse environments using LiDAR data. Additionally, the tests are completed on both cropped and uncropped data to identify any applicable differences.

Aim

To evaluate and compare the performance of SegMatch and NDT registration in localising a robot in structured and unstructured environments using LiDAR data.

Hypothesis

The processing pipeline can accurately localise the robot in both structured and unstructured paths, using both SegMatch and NDT registration techniques.

Variables

1. Independent: Path type or the mapping dataset.
2. Dependent: Accuracy of the localisation.
3. Controlled: Dataset size, environmental conditions, LiDAR settings and sensor, comparison metrics.

Data Collection

- 4 sets of LiDAR data:
  - 2 indoors (structured environment)
  - 1 outdoors (unstructured environment), a cropped version and a full version.
- Each dataset is split for map creation and localisation testing with different starting points to evaluate localisation accuracy.

- The outdoor set is evaluated as full point clouds, and cropped point clouds, to evaluate the difference in accuracy.
- The indoor data was collected in the same environment, but two different paths were traversed: one was a loop around the room's perimeter, and the second was a figure of 8, travelling from the starting point, to the left through the centre of the two tables, right around the far table, crossing back through the tables, and around the table closest to the camera - ending at the starting point. The room is depicted in the following image, where the camera is placed at the starting point of the paths:

Figure 5.1: Indoor Environment for Data Capture

Tools

- MATLAB for data processing and analysis.
- Tic&Toc for timestamp data.
- All hardware and software defined in Section 1.3, related to data collection and MATLAB processing, as well as using PCL to crop to a ROI.

Procedure

- The LiDAR data was pre-processed (including being cropped to a ROI using ?? if applicable).
- The pipeline was used to build a pose graph/map using the mapping set of point clouds.
- SegMatch and NDT registration localisation were executed on the map, using the localisation cloud sets on MATLAB, and results were collected - using the same number of point clouds each time (250 for Mapping, 250 for Localisation).

- Compare localisation accuracy, accumulated error, and processing time, between SegMatch and NDT registration.
- Investigate the feature matching error and the point at which features can no longer be matched between files.

5.1.1 Key Performance Indicators (KPIs)

- Localisation accuracy
- Accumulated rrror
- Processing time
- Path complexity (qualitative)

5.1.2 Success Criteria

- Accuracy threshold: localised position should be within 0.5m of the map position.
- Maximum acceptable processing time for localisation = 2 seconds.

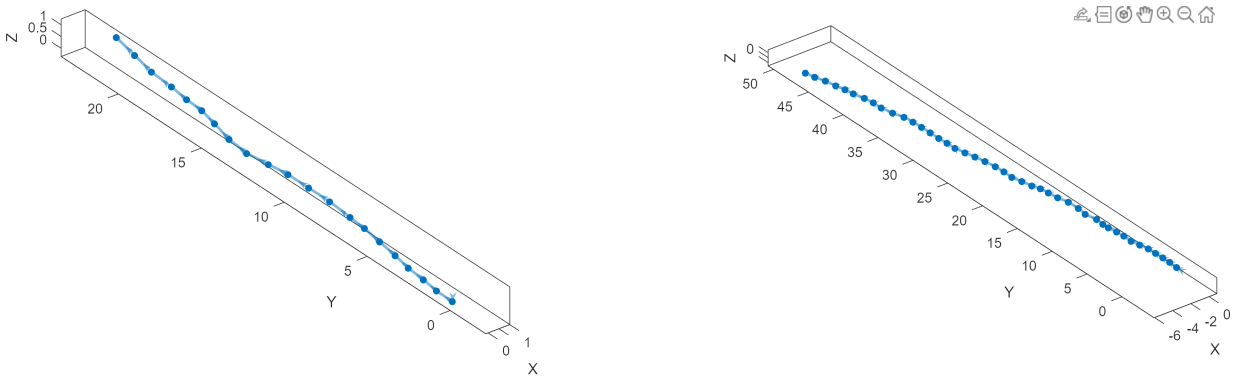
5.1.3 Data Analysis

- Use MATLAB for statistical analysis.
- Graphical representation of localisation accuracy and other KPIs.

Chapter 6

Results & Discussion

6.1 Unstructured Environments



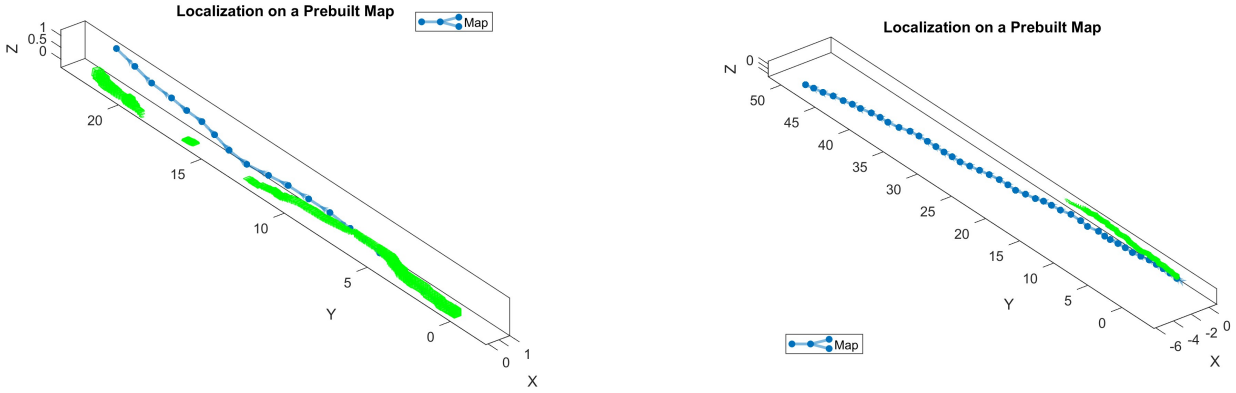
(a) Full Cloud

(b) Cropped

Figure 6.1: Figure Showing the Map Comparison between Cropped and Full Point Cloud Data

In the above figure, the Full Cloud data shows the vehicle travelling in mostly a straight line, with slight changes in elevation. The cropped point clouds result in the map being built much further into the Y dimension, with a drift into the negative X dimension, even though the data sets contained the same number of mapping point clouds.

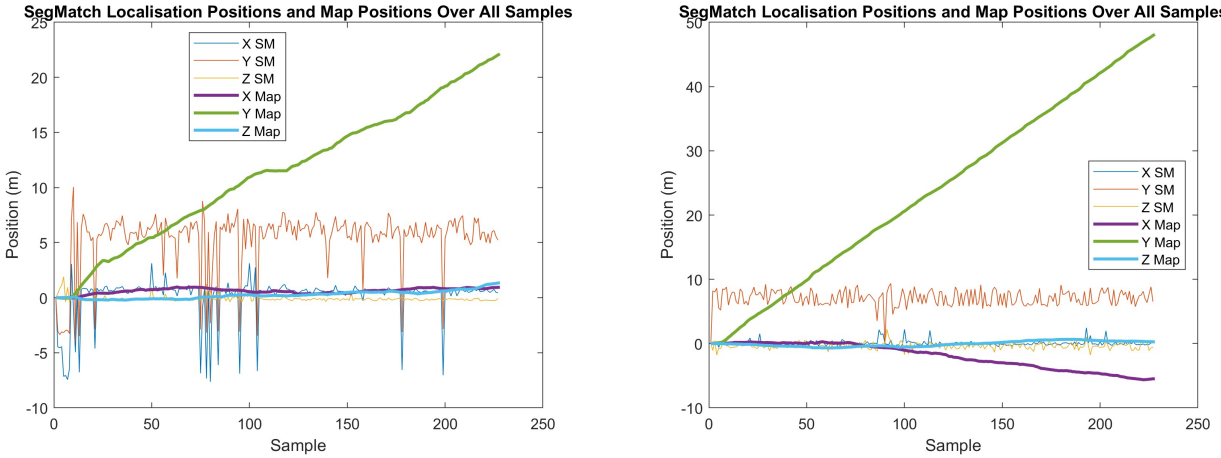
In Figure 6.2: The localisation using registration was plotted against the map built using SegMatch and loop closure detection. The cropped dataset shows that the registration moves in a straight line in the Y dimension, and does not strictly follow the map. This shows that the map built with the cropped data was inaccurate, as for the full cloud, the registration follows the map for the most part, with a smaller error at the end (the blank parts are where the registration moved off the map). Additionally, the dimensions presented by the registration for both the cropped and full clouds are similar, indicating that the registration is more reliable, and the map created with the cropped point clouds is where the fault lies in the cropped trial for the most part.



(a) Full Cloud (b) Cropped

Figure 6.2: Figure Showing the Localisation on the Maps for Full Cloud Data and Cropped Data

The following figures show the comparison of the map locations, to the locations produced by localisation using both NDT registration and SegMatch. The keys show 'X Map' and 'X Loc'/'X SM' for all X, Y, and Z coordinates, which shows a comparison of the coordinates returned by the map, to the coordinates returned by the localisation processes respectively.



(a) Full Cloud (b) Cropped

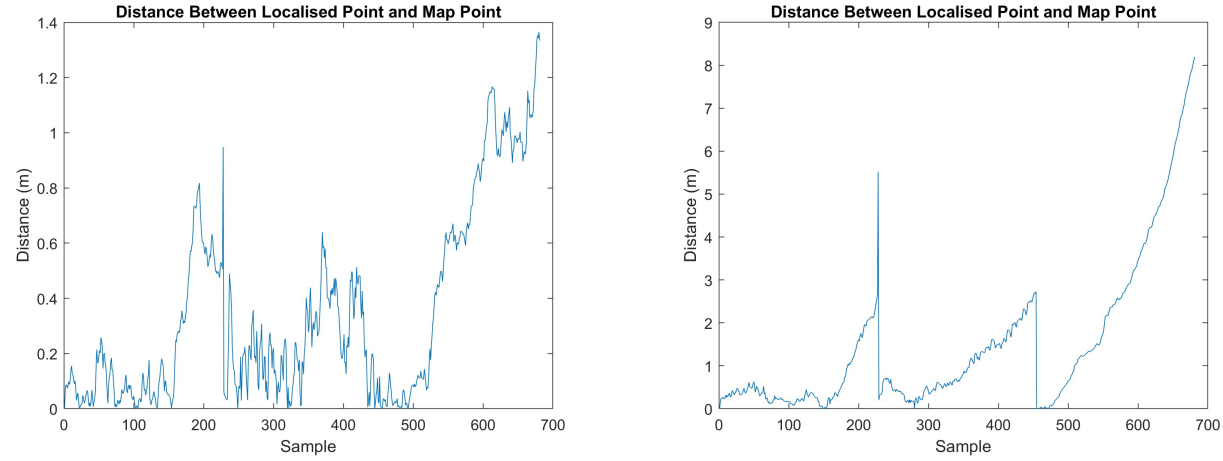
Figure 6.3: Figure Showing the Comparison between Map Locations and SegMatch Locations, for Cropped and Full Point Cloud Data

The results of the SegMatch localisation are anything but successful and show that for all points on the map, SegMatch fails to locate a corresponding position from the localisation data set.

(a) Full Cloud (b) Cropped

Figure 6.4: Figure Showing the Comparison between Map Locations and NDT Registration Locations, for Cropped and Full Point Cloud Data

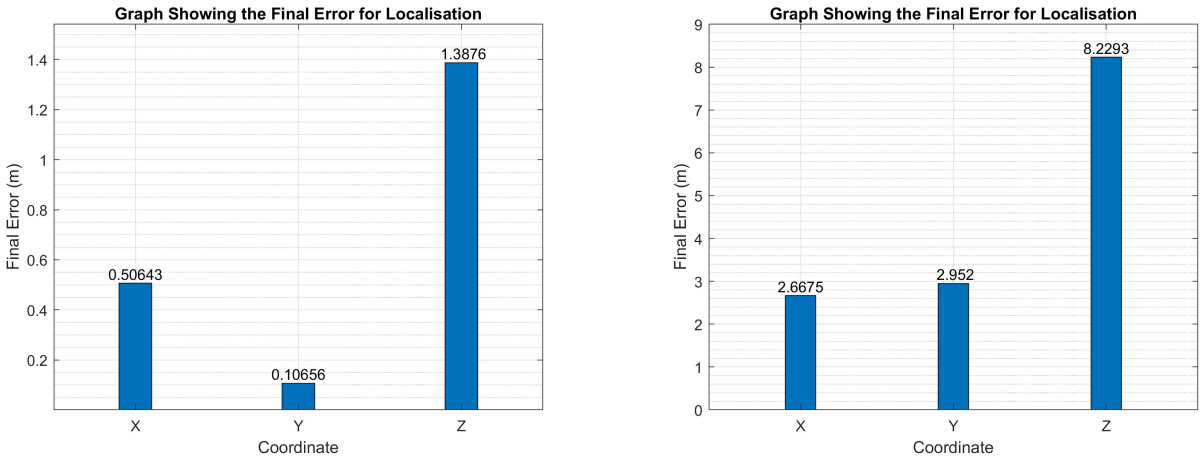
In Figure 6.4: the results show that for the full clouds, the NDT registration is wholly successful when localising against a map built with SegMatch methods. In contrast to this, for the cropped point clouds, there is a lot more error between the map and the registration localisation. This confirms the results shown by Figure 6.2.



(a) Full Cloud (b) Cropped

Figure 6.5: Figure Showing the Distance (error) between Map Locations and NDT Localisations, for Cropped and Full Point Cloud Data

These graphs in Figure 6.5 show how the accumulated error over the same interval is just over 6 times as much for the cropped clouds as it is for the full clouds. They also show a trend of three peaks in distance throughout the datasets at similar intervals, which if compared to Figure 6.2 (a) which shows the registration moving off the map in the Z dimension, can be mostly attributed to significant changes in the elevation, indicating that the elevation threshold for the segmentation or ground plane removal can be investigated further. The graphs in Figure 6.6 show the drastic differences in error, where the Z dimension is the cause for the majority of the localisation error by the end of the localisation process.



(a) Full Cloud

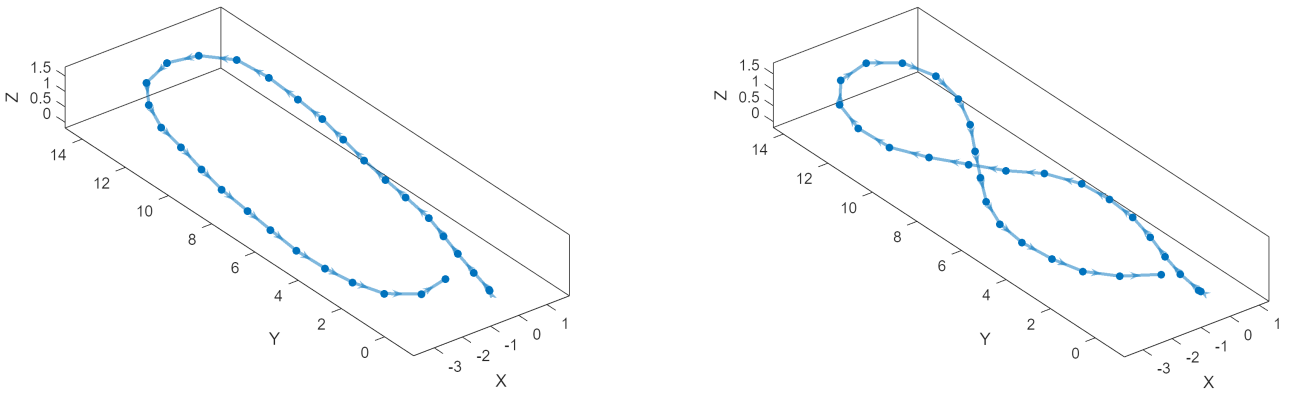
(b) Cropped

Figure 6.6: Figure Showing the Final Error (after 230 points) between Map Locations and Localised Points, for Cropped and Full Point Cloud Data

The error for the cropped clouds is also staggeringly high compared to when the full clouds are used. This could be due to the increased number of features present in the full clouds, for both building the map and localising against it. The error in the Z dimension could be attributed to the elevation threshold chosen when segmenting and removing the ground plane, but because different values were experimented with, the reason why there is drift is suggested to be because of the feature extraction and matching being unreliable, which is explored in the analysis of Figure 6.14

6.2 Structured Environments

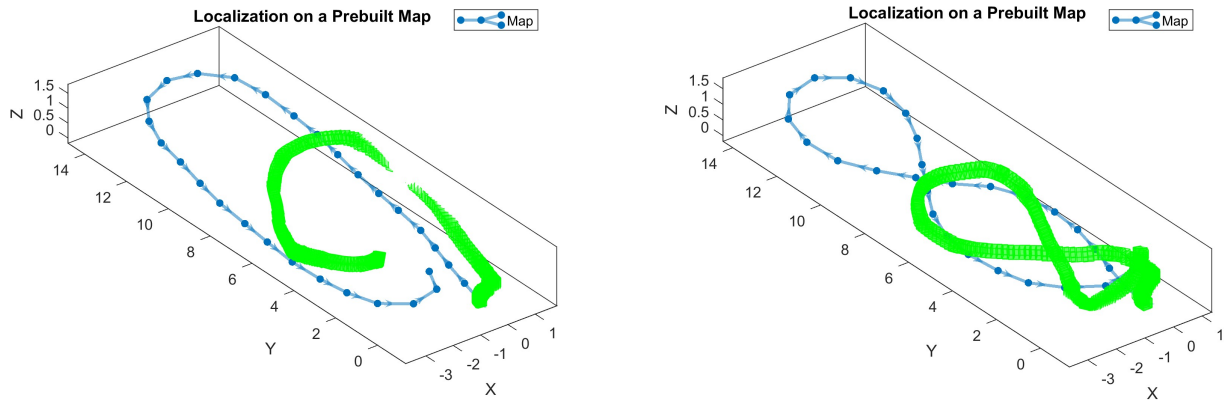
The maps created using the SegMatch process are depicted in Figure 6.7: for both the path around the indoor perimeter and the figure of 8 around the tables, as mentioned in the experimental outline 5. This shows that the map building had some success, with the dimensions of the map being close to realistic, besides the fact that the maps include elevation error in the Z dimension because the room was flat.



(a) Perimeter Path

(b) Figure-of-8 Path

Figure 6.7: Figure Showing the Map Comparison between the Two Indoor Environments



(a) Perimeter Path

(b) Figure-of-8 Path

Figure 6.8: Figure Showing the Localisation on the Maps for the Two Indoor Environments

The maps with the NDT-based localisation are shown in Figure 6.8, where the overall shapes of the registration are correct, but they do not accurately fit the maps. Because Figures 6.2 and 6.4 (a) show that NDT can successfully localise on a map built by SegMatch, the conclusion drawn is that the SegMatch Map is assumed to be correct, as it builds the map using a combination of the NDT registration and feature matching, whereas the localisation is purely NDT-based. This results in there being an error for both the indoor areas, where the pipeline was investigated. The outcome of the investigation was that there was an error in the data capturing with the sensor, as one of the driver updates for the Husky had affected its Velodyne LiDAR sensor, resulting in data corruption: the flat room looked curved in the LiDAR scans, and there was an unexplained radius of points around the sensor which were not present in the environment. This was for all the data collected in the indoor environment. Nevertheless, the tests were still continued, and the results were presented, but no further analyses were conducted due to them becoming irrelevant.

For Figures 6.10 to 6.12: Due to the data corruption, the results of the tests were rendered inconclusive,

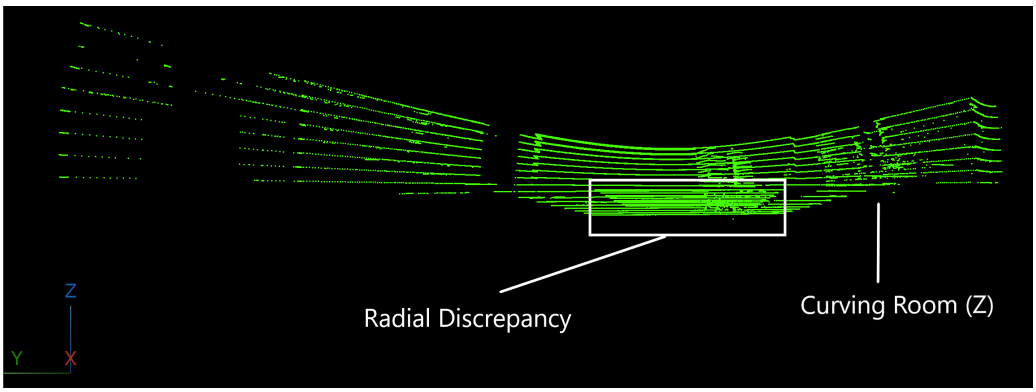
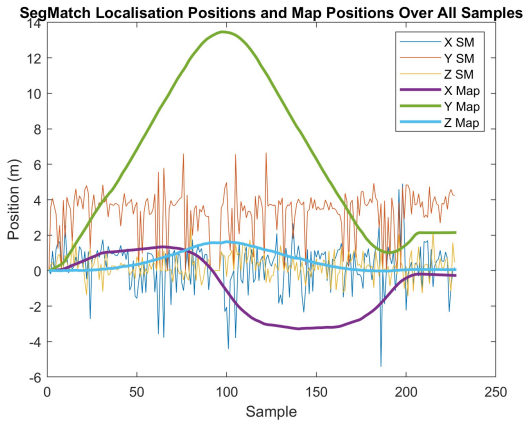


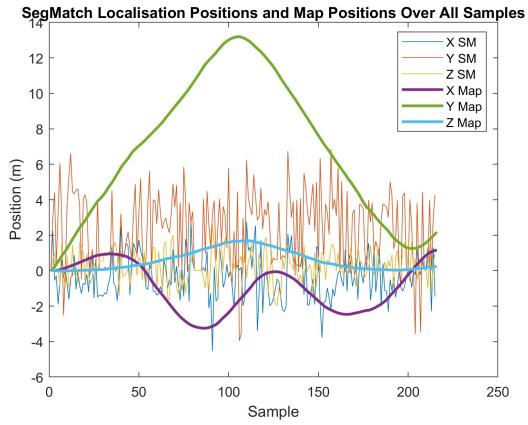
Figure 6.9: Figure Showing the Indoor Data's Corruption

but indicate trends on a qualitative basis. The trends identified was that most of the error in localisation for the NDT registration method happened furthest from the starting point, confirmed by the results in the unstructured environment, and also that the SegMatch localisation is just as ineffective in

the structured environments as it was in the unstructured one. An effort was made to use the indoor/structured data from the MATLAB examples [77][?] to do some testing, however, the data use was restricted to the examples because the functions to read the data into MATLAB were hidden from users.

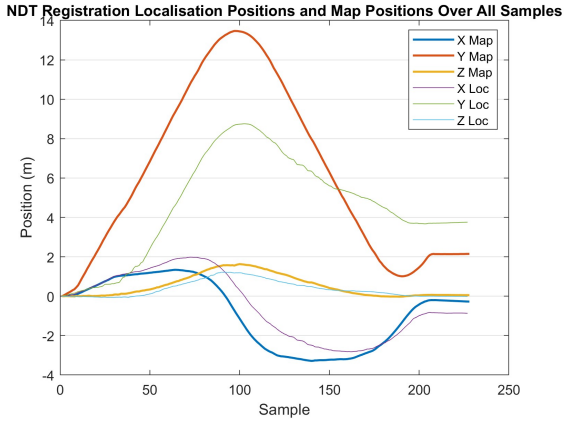


(a) Perimeter Path

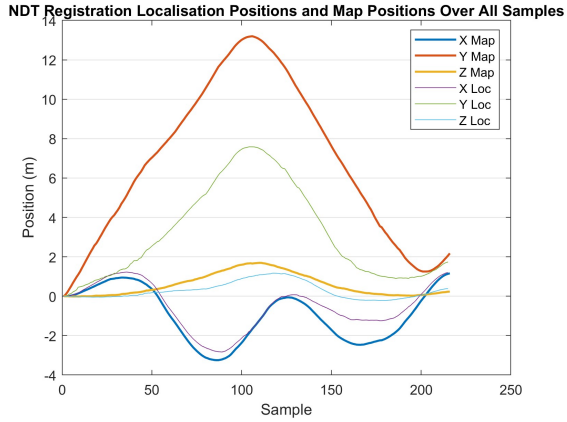


(b) Figure-of-8 Path

Figure 6.10: Figure Showing the Comparison between Map Locations and SegMatch Locations, for the Two Indoor Environments



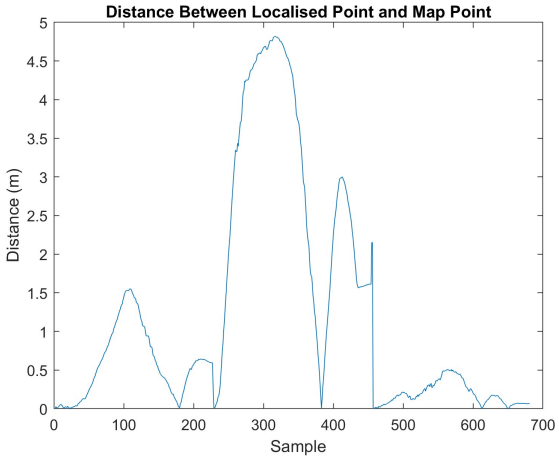
(a) Perimeter Path



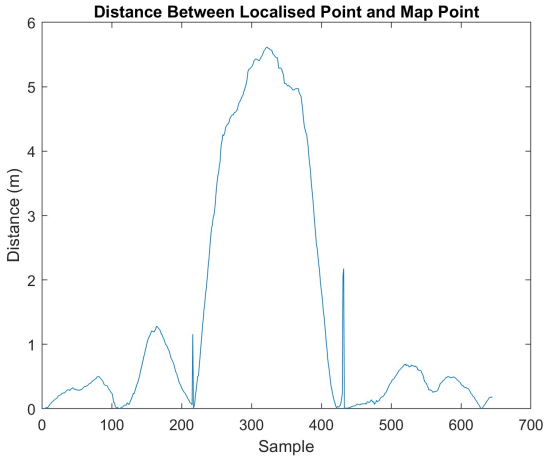
(b) Figure-of-8 Path

Figure 6.11: Figure Showing the Comparison between Map Locations and NDT Registration Locations, for the Two Indoor Environments

It was unnecessary to evaluate the final accumulated error at the end of the localisation because the paths for the indoor trial started and ended at the same position. The important metric to focus on is the distance of the localised points from the map (Figure 6.12), showing that approximately halfway through the trial is where the map and localisation differ in position the most, for both indoor trials. This is supported by the figures showing localisation on the maps, where the most difference is approximately halfway through the process. The large error in distance is attributed to the corruption of data.



(a) Perimeter Path



(b) Figure-of-8 Path

Figure 6.12: Figure Showing the Distance (error) between Map Locations and NDT Localisations, for Cropped and Full Point Cloud Data

	Scenario	Map Building (ms)	Registration (ms)	SegMatch (ms)
1	"Indoor, 8 shape"	106.5759	520.4070	0.3060
2	"Indoor, U shape"	76.6744	291.6747	0.2899
3	"Farm Data"	60.0281	657.3723	0.5570
4	"MapVMap Farm Data Crop"	41.4675	334.3984	0.0705
5	"Farm Data Cropped"	41.4675	330.0201	0.0668

Figure 6.13: Screenshot of MATLAB Table Constructed with Timing Data from Tests

K	Q	Difference
10	12	2
30	32	2
75	77	2
100	102	2
120	122	2
150	152	2
180	182	2
220	222	2
225	227	2

Figure 6.14: Screenshot of MATLAB Table Constructed with Feature Matching Error Test Results, where K is the Starting Point, and Q is the Point at which the Feature Matching Fails

The timing results accumulated by MATLAB using the tic & toc functions are shown in Figure 6.13. This shows a few interesting results:

1. The cropped point clouds underwent map building in a much shorter amount of time (41.4675 seconds) than the full clouds (60.0281 seconds) for the Farm data set. And the more complicated paths in the structured environments (including turns) took significantly longer to build the map for (106.5759 seconds for a complex shape - Figure-of-8, and 76.6744 for a perimeter path).

2. The registration localisation for the data sets was faster on cropped data (approximately half the time of the full cloud registration localisation), due to the cropped clouds being approximately half the size of the full ones. The registration localisation on the indoor data was faster than for the unstructured farm data in both cases, but as mentioned previously, was incorrect due to the data being corrupted.

3. Finding a singular pose using SegMatch, although the poses were incorrect (results from Figures 6.3 and 6.10 show no tracking of the map when Segmatch was used for localisation), took a significantly reduced amount of time for cropped clouds, when compared to the full clouds, and is also faster for structured data (but not as much of a difference is made as the cropping).

4. The localisation is slower in registration and SegMatch localisation when the cropped map set is localised against itself (MapVMap).

5. The average time for a singular registration of the Farm data (with a map set and a localisation set size of 217 clouds) is $\frac{657.3723}{217} = 3.0293$ seconds, which is a lot longer than for SegMatch, but considering its accuracy is an improvement on the ICP localisation in PCL which takes approximately 30 seconds to localise one cloud in a data set of 23 clouds.

Lastly, it was identified that throughout the design process, there were issues with feature matching if the point clouds being matched were too far away from one another, or too different. This means that SegMatch is not reliable when matching features in point clouds that are too far apart, and so if the data is split into more sets (so the consecutive scans in the mapping subset of clouds are too far apart), a reliable map and localisation using SegMatch will not be produced. The test conducted was feeding two point clouds into a feature matching sequence, looping through the set from the index of the first input cloud onward to see where an error is encountered. In Figure 6.14, K represents the index of the starting point cloud, from which the loop to match the features began, and Q is the last index at which feature matching was successful. There is a constant difference of 2 point clouds. Because this was done with a subset of point clouds (the map set of the Farm data), which takes every second point cloud of the larger data set, it means that feature matching is effective for twice the difference shown in the results: so it took 4 point clouds for the feature matching to become ineffective. This was not done for the structured data sets due to their corruption.

Chapter 7

Conclusions

> Look at how a single candle can both defy and define the darkness.
>
> —*Anne Frank*

The purpose of this project was to develop a processing pipeline using SegMatch for localisation in unstructured environments. This report began with an introduction to the research context. The literature review followed in Chapter 2, presenting an overview of existing techniques and technologies relevant to the scope of this project, with a focus on localisation methods and the underpinning concepts. The bulk of the work for this project was detailed in Chapter 4, where the design and development of the processing pipeline were elaborated. The chapter provided insights into the methodological approach adopted for data collection and preprocessing using MATLAB.

In Chapter 6, the data collected and processed from both structured and unstructured environments was presented and analysed. The chapter explained the findings from the comparative evaluation of SegMatch and NDT registration for localisation, based on the different data sets. The chapter also mentions the challenges encountered, particularly with data integrity for the structured environment tests and the limitations of SegMatch in feature matching when point clouds are distant from each other.

The analysis presented provided a comparative assessment of SegMatch and NDT registration for localising a robot in structured and unstructured environments using LiDAR data. The evaluation was conducted on both cropped and uncropped data to identify any significant variances that may affect the accuracy and processing time of the localisation. In the unstructured environment, it was observed that the cropped data resulted in a drift in the map construction, affecting the subsequent localisation accuracy. Particularly, the SegMatch localisation failed to provide successful localisation, as depicted in Figure 6.3. On the other hand, NDT registration showcased better localisation when utilising full cloud data, indicating a more robust performance in unstructured settings as seen in Figure 6.4.

The structured environment data, although affected by data corruption and sensor issues, revealed some level of success in map building using the SegMatch process. However, the NDT-based localisation did not accurately align with the maps, as shown in Figure 6.8.

The timing analysis revealed that cropped point clouds facilitated a faster map building and registration localisation process. However, the trade-off was evident in the reduced localisation accuracy, indicating a trade-off between processing speed and accuracy. Furthermore, the feature matching limitations of SegMatch, particularly when point clouds are too distant, were highlighted. This limitation could

potentially affect the reliability of SegMatch in diverse environments and needs further investigation to evaluate its effectiveness.

The hypothesis formulated at the beginning of the experiments was that the processing pipeline can accurately localise the robot in both structured and unstructured paths, using both SegMatch and NDT registration techniques, was partially validated, and partially disproven. The NDT registration showed robust performance in unstructured environments, while SegMatch faced challenges in localisation accuracy, particularly in unstructured settings. Additionally, limitations on the structured data prevented an in-depth analysis on the application of SegMatch localisation to it.

In summary, the investigations outlined the challenges and considerations necessary for accurate and reliable robotic localisation in varied environmental settings. The findings outline the importance of thorough data preprocessing and investigative approach in achieving this localisation. The comparative assessment between SegMatch and NDT registration provides a basis for further exploration and improvements in robotic localisation.

Chapter 8

Recommendations

> Our greatest human adventure is the evolution of consciousness. We are in this life to enlarge the soul, liberate the spirit, and light up the brain.
> —*Tom Robbins*

Future work should begin with the exploration of SegMatch within a ROS-based system and environment, for which SegMatch was originally designed in C++. The investigation could provide more insight into the real-time performance, adaptability to larger data sets, and the capabilities for integration into other systems. It could also enable a broader assessment of the processing abilities in more environments, providing a more streamlined approach than the adaptation provided by MATLAB.

A portion of the MATLAB code which was omitted due to differences in data sets was the incorporation of INS data. Including the INS data could potentially improve the map building and localisation process using SegMatch, especially in challenging and/or unstructured environments. An evaluation of the combination of point clouds with INS data within the redesigned pipeline, and a further investigated ROS-based pipeline could lead to more robust and reliable maps and localisation to enhance the processing results.

Due to the disappointing corruption of the collected structured data following driver issues with the Husky, a valuable future contribution would include the thorough investigation of the pipeline's performance with structured data. This could provide further insights to any necessary adjustments for the different environments' data and would definitely contribute to the understanding of SegMatch and NDT's performance with data from structured environments.

The exploration of larger data sets, with more point clouds, could bring new challenges and results to the surface of the exploration. This would require more computational resources than this project was limited to. By taking advantage of increased computational power, the processing pipeline could be more efficient for larger data sets and could therefore lead to the adjustment of the pipeline for less accumulated error, and a more thorough performance evaluation of the designed pipeline.

Each of the recommendations provide a unique new exploration which was not within the scope or time limitations of this project. Each of the unique explorations have the potential to contribute to enhancing the understanding and advancement of the use of SegMatch and NDT registration for the localisation of a UGV using LiDAR data in unstructured environments.

Bibliography

[1] D. P. Amayo, "Lidar point cloud data from velodyne sensor: Farm data," 2023, unpublished raw data. Provided personally.

[2] C. Jinming, "Obstacle detection based on 3d lidar euclidean clustering," *Applied Science and Innovative Research*, vol. 5, p. p39, 11 2021.

[3] R. B. Rusu, N. Blodow, and M. Beetz, "Fast point feature histograms (fpfh) for 3d registration." IEEE, 5 2009, pp. 3212–3217.

[4] MATLAB and MathWorks, "segmentgroundsmrf: Matlab documentation," 2023. [Online]. Available: https://www.mathworks.com/help/lidar/ref/segmentgroundsmrf.html?searchHighlight=segmentGroundSMRF&s_tid=srchtitle_support_results_1_segmentGroundSMRF

[5] T. J. Pingel, K. C. Clarke, and W. A. McBride, "An improved simple morphological filter for the terrain classification of airborne lidar data," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 77, pp. 21–30, 3 2013.

[6] X. Yu, H. Wang, H. Lv, and J. Fu, "An optimization technique of the 3d indoor map data based on an improved octree structure," *Mathematical Problems in Engineering*, vol. 2020, pp. 1–13, 7 2020.

[7] M. Skrodzki, "The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time," *arXiv Computer Science: Data Structures and Algorithms*, 3 2019. [Online]. Available: https://arxiv.org/abs/1903.04936

[8] I. Bogoslavskyi and C. Stachniss, "Efficient online segmentation for sparse 3d laser scans," *PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, vol. 85, pp. 41–52, 2 2017.

[9] MATLAB and MathWorks, "segmentlidardata," 2023. [Online]. Available: https://www.mathworks.com/help/vision/ref/segmentlidardata.html

[10] T. M. Inc., *MATLAB*, Natick, Massachusetts, 2023, version R2023a. [Online]. Available: https://www.mathworks.com/products/matlab.html

[11] P. C. L. Contributors, "Test data for segmentation: test24.pcd," 2014. [Online]. Available: https://github.com/PointCloudLibrary/data/blob/master/segmentation/mOSD/test/test24.pcd

[12] P. C. Library, "Euclidean cluster extraction," 2014. [Online]. Available: https://pcl.readthedocs.io/projects/tutorials/en/master/cluster_extraction.html

[13] L. S. Inc., "Lucidchart," 2023. [Online]. Available: https://lucid.co/

[14] P. C. Library, "Plane model segmentation," 2014. [Online]. Available: https://pcl.readthedocs.io/projects/tutorials/en/master/planar_segmentation.html#planar-segmentation

[15] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China: IEEE, May 9-13 2011.

[16] M. Weinmann, B. Jutzi, and C. Mallet, "Semantic 3d scene interpretation: A framework combining optimal neighborhood size selection with relevant features," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. II-3, pp. 181–188, 8 2014.

[17] Wikipedia, "Random sample consensus," 10 2023. [Online]. Available: https://en.wikipedia.org/wiki/Random_sample_consensus

[18] OpenAI, "Chatgpt (september 25 version)," https://chat.openai.com, 2023, large language model.

[19] MATLAB and MathWorks, "Build map and localize using segment matching," 2023. [Online]. Available: https://www.mathworks.com/help/lidar/ug/build-a-map-and-localize-using-segment-matching.html

[20] R. Dubé, D. Dugas, E. Stumm, J. Nieto, R. Siegwart, and C. Cadena, "Segmatch: Segment based place recognition in 3d point clouds," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 5266–5272.

[21] S. S. Iyengar, C. C. Jorgensen, S. V. N. Rao, and C. R. Weisbin, "Robot navigation algorithms using learned spatial graphs," *Robotica*, vol. 4, pp. 93–100, 4 1986. [Online]. Available: https://www.cambridge.org/core/journals/robotica/article/abs/robot-navigation-algorithms-using-learned-spatial-graphs/1BF548D1833DCB858C5CAE8F13812EA6

[22] T. Guan, D. Kothandaraman, R. Chandra, A. J. Sathyamoorthy, K. Weerakoon, and D. Manocha, "Ga-nav: Efficient terrain segmentation for robot navigation in unstructured outdoor environments," *IEEE Robotics and Automation Letters*, vol. 7, pp. 8138–8145, 7 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9810192?casa_token=Q7ZlZ4snqHAAAAAA:Gk5pQpzZJUOiWPdOMnwt9qzkr7zXV265SBwzEUIrO3g54fsuNyU-2Q5zjxSZrYoJgPYVTn4SMg

[23] D. C. Guastella and G. Muscato, "Learning-based methods of perception and navigation for ground vehicles in unstructured environments: A review," *Sensors*, vol. 21, p. 73, 12 2020. [Online]. Available: https://www.mdpi.com/1424-8220/21/1/73

[24] C. Wang, L. Meng, S. She, I. M. Mitchell, T. Li, F. Tung, W. Wan, M. Q.-H. Meng, and C. W. de Silva, "Autonomous mobile robot navigation in uneven and unstructured indoor environments," *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 109–116, 9 2017. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8202145?casa_token=njtNcNttQlMAAAAA:LYba5CbHUbQHVyrFxeaiaQMDPmYUSEEIt25ouPZ4XTSSh2IVfFD8ct2sPrRhnaHEGoPGmPjyWw

[25] T. Bailey, "Mobile robot localisation and mapping in extensive outdoor environments," 8 2002. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?repid=rep1&type=pdf&doi=10.1.1.4.1707

[26] G. Tinchev, S. Nobili, and M. Fallon, "Seeing the wood for the trees: Reliable localization in urban and natural environments," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 8239–8246, 10 2018. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8594042&casa_token=CaI863WCA0wAAAAA:xLMdf8cEZzECb2AyZtEXuusOGX8OD4giJMr-k9EJ4At_oSGcDPG11fnVjzkuX7COLD0_ix0kHw

[27] M. Pierzchała, P. Giguère, and R. Astrup, "Mapping forests using an unmanned ground vehicle with 3d lidar and graph-slam," *Computers and Electronics in Agriculture*, vol. 145, pp. 217–225, 2 2018.

[28] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, and R. Vincent, "Efficient sparse pose adjustment for 2d mapping," *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 22–29, 2010. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5649043

[29] G. Grisetti, C. Stachniss, and W. Burgard, "Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling," *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pp. 2432–2437, 2005. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1570477

[30] H. Peel, S. Luo, A. Cohn, and R. Fuentes, "Localisation of a mobile robot for bridge bearing inspection," *Automation in Construction*, vol. 94, pp. 244–256, 10 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0926580517303916

[31] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d lidar slam," *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1271–1278, 2016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7487258?casa_token=QuT2oNpqlOcAAAAA:pes9othTcT8DwyThLZt3jKHeXyWHXXYpcKOlbWRffpJMw7YqwBxXy6cB6l_rvLwI6yW0kt8u6g

[32] G. Tinchev, A. Penate-Sanchez, and M. Fallon, "Learning to see the wood for the trees: Deep laser localization in urban and natural environments on a cpu," *IEEE Robotics and Automation Letters*, vol. 4, pp. 1327–1334, 4 2019.

[33] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on Robotics*, vol. 32, pp. 1309–1332, 2016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7747236?casa_token=vAs7IIf0oiEAAAAA:Wzc57sRsfDGyB-MkafV9YQl2xbpoxOKnp3cVH905IXojCEB2xorNYOJFje3yKUU1j4Tt6Xog3A

[34] Q. Zou, Q. Sun, L. Chen, B. Nie, and Q. Li, "A comparative analysis of lidar slam-based indoor navigation for autonomous vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, pp. 6907–6921, 7 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9381521?casa_token=KiWmptcsNvgAAAAA:Rudhe82AyX0ChItpRVm4Z_HcXbfJIrLPYb8HTujq4EVu9okJPFUz556LRm7WFmwchj0y0yrQZ1s

[35] A. S. Aguiar, F. N. dos Santos, J. B. Cunha, H. Sobreira, and A. J. Sousa, "Localization and mapping for robots in agriculture and forestry: A survey," *Robotics*, vol. 9, p. 97, 11 2020.

[36] C. Pang, Y. Tan, S. Li, Y. Li, B. Ji, and R. Song, "Low-cost and high-accuracy lidar slam for large outdoor scenarios," *2019 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pp. 868–873, 8 2019.

[37] R. O. Dubayah and J. B. Drake, "Lidar remote sensing for forestry," *Journal of Forestry*, vol. 98, pp. 44–46, 6 2000. [Online]. Available: https://doi.org/10.1093/jof/98.6.44

[38] M. A. Lefsky, W. B. Cohen, S. A. Acker, T. A. Spies, G. G. Parker, and D. Harding, "Lidar remote sensing of forest canopy structure and related biophysical parameters at h.j. andrews experimental forest, oregon, usa," *IGARSS '98. Sensing and Managing the Environment. 1998 IEEE International Geoscience and Remote Sensing. Symposium Proceedings. (Cat. No.98CH36174)*, vol. 3, pp. 1252–1254 vol.3, 1998.

[39] M. U. Khan, S. A. A. Zaidi, A. Ishtiaq, S. U. R. Bukhari, S. Samer, and A. Farman, "A comparative survey of lidar-slam and lidar based sensor technologies," *2021 Mohammad Ali Jinnah University International Conference on Computing (MAJICC)*, pp. 1–8, 7 2021.

[40] D. V. Nam and K. Gon-Woo, "Solid-state lidar based-slam: A concise review and application," *2021 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 302–305, 1 2021.

[41] G. Ren, C. Ai, Q. Xu, Z. Wang, Z. Wang, and D. Geng, "Research on indoor and outdoor navigation technology based on the combination of differential gnss and lidar slam," *2020 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pp. 134–139, 9 2020.

[42] S. Thrun, "Probabilistic robotics," *Communications of the ACM*, vol. 45, pp. 52–57, 2002. [Online]. Available: https://dl.acm.org/doi/fullHtml/10.1145/504729.504754?casa_token=IHtUkZy16wEAAAAA:ojeZ4170xEfsZHObepHCfS9nqjt3voyv1kaLXXyX_Ep5i60wWeMcOFz-mSm9Gqdh5SAJrpStO3sH

[43] S. T. O'Callaghan, F. T. Ramos, and H. Durrant-Whyte, "Contextual occupancy maps incorporating sensor and location uncertainty," *2010 IEEE International Conference on Robotics and Automation*, pp. 3478–3485, 5 2010.

[44] Y. Li and J. Ibanez-Guzman, "Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems," *IEEE Signal Processing Magazine*, vol. 37, pp. 50–61, 7 2020.

[45] H. Taheri and Z. C. Xia, "Slam; definition and evolution," *Engineering Applications of Artificial Intelligence*, vol. 97, p. 104032, 1 2021.

[46] M. J. Procopio, J. Mulligan, and G. Grudic, "Learning terrain segmentation with classifier ensembles for autonomous robot navigation in unstructured environments," *Journal of Field Robotics*, vol. 26, pp. 145–175, 2 2009. [Online]. Available: https://onlinelibrary.wiley.com/doi/epdf/10.1002/rob.20279

[47] H. Dong, X. Chen, S. Särkkä, and C. Stachniss, "Online pole segmentation on range images for long-term lidar localization in urban environments," *Robotics and Autonomous Systems*, vol. 159, p. 104283, 1 2023.

[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[49] Y. Cao, Y. Wang, Y. Xue, H. Zhang, and Y. Lao, "Fec: Fast euclidean clustering for point cloud segmentation," *Drones*, vol. 6, p. 325, 10 2022.

[50] A. M. Ramiya, R. R. Nidamanuri, and R. Krishnan, "Segmentation based building detection approach from lidar point cloud," *The Egyptian Journal of Remote Sensing and Space Science*, vol. 20, pp. 71–77, 6 2017.

[51] G. Kulathunga, R. Fedorenko, and A. Klimchik, "Regions of interest segmentation from lidar point cloud for multirotor aerial vehicles." IEEE, 9 2020, pp. 1213–1220.

[52] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[53] M. A. Fischler and R. C. Bolles, "Random sample consensus," *Communications of the ACM*, vol. 24, pp. 381–395, 6 1981.

[54] K. Zhang, S.-C. Chen, D. Whitman, M.-L. Shyu, J. Yan, and C. Zhang, "A progressive morphological filter for removing nonground measurements from airborne lidar data," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 41, pp. 872–882, 4 2003.

[55] P. C. Library, "Removing outliers using a statisticaloutlierremoval filter," 2023. [Online]. Available: https://pcl.readthedocs.io/projects/tutorials/en/latest/statistical_outlier.html

[56] J. Qu, S. Li, Y. Li, and L. Liu, "Research on railway obstacle detection method based on developed euclidean clustering," *Electronics*, vol. 12, p. 1175, 2 2023.

[57] H. Ni, X. Lin, X. Ning, and J. Zhang, "Edge detection and feature line tracing in 3d-point clouds by analyzing geometric properties of neighborhoods," *Remote Sensing*, vol. 8, p. 710, 9 2016. [Online]. Available: https://www.mdpi.com/2072-4292/8/9/710

[58] B. O'Neill, *Global Structure of Surfaces*. Elsevier, 2006, pp. 388–450.

[59] M. Bosse and R. Zlot, "Place recognition using keypoint voting in large 3d lidar datasets." IEEE, 5 2013, pp. 2677–2684.

[60] E. R. Boroson and N. Ayanian, "3d keypoint repeatability for heterogeneous multi-robot slam." IEEE, 5 2019, pp. 6337–6343.

[61] B. Smith, Ed., *Foundations of Gestalt Theory*. Philosophia, 1988, pp. 11–81.

[62] K. Koffka, *Principles of Gestalt psychology*. Routledge, 2013, vol. 44.

[63] A. Hacinecipoglu, E. Konukseven, and A. Koku, "Pose invariant people detection in point clouds for mobile robots," *International Journal of Mechanical Engineering and Robotics Research*, pp. 709–715, 01 2020.

[64] P. C. Library, "Downsampling a point cloud using a voxelgrid filter," 2014. [Online]. Available: https://pcl.readthedocs.io/projects/tutorials/en/latest/voxel_grid.html

[65] MATLAB and MathWorks, "3-d point cloud registration and stitching," 2023. [Online]. Available: https://www.mathworks.com/help/vision/ug/3-d-point-cloud-registration-and-stitching.html

[66] S. N. M. Isa, S. A. A. Shukor, N. A. Rahim, I. Maarof, Z. R. Yahya, A. Zakaria, A. H. Abdullah, and R. Wong, "Point cloud data segmentation using ransac and localization," *IOP Conference Series: Materials Science and Engineering*, vol. 705, p. 012004, 11 2019.

[67] P. C. Library, "How to use random sample consensus model," 2014. [Online]. Available: https://pcl.readthedocs.io/projects/tutorials/en/latest/random_sample_consensus.html

[68] J. Elseberg, D. Borrmann, and A. Nüchter, "One billion points in the cloud – an octree for efficient processing of 3d laser scans," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 76, pp. 76–88, 2 2013.

[69] N. Wroblewski, "Nearest neighbor search by k-dimensional tree traversal," 2020. [Online]. Available: https://www.nathaniel.ai/kd-tree/

[70] W. R. Inc and MathWorld, "Hyperplanes," 10 2023. [Online]. Available: https://mathworld.wolfram.com/Hyperplane.html

[71] MATLAB and MathWorks, "extracteigenfeatures," 2023. [Online]. Available: https://www.mathworks.com/help/lidar/ref/extracteigenfeatures.html

[72] P. Besl and N. D. McKay, "A method for registration of 3-d shapes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, pp. 239–256, 2 1992.

[73] M. Mortenson, *Geometric Modeling*, 1st ed. Wiley, 7 1985, vol. 1.

[74] MathWorks, "Rotations, orientation, and quaternions," 2023. [Online]. Available: https://www.mathworks.com/help/fusion/ug/rotations-orientation-and-quaternions.html

[75] "The normal distributions transform: A new approach to laser scan matching," in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, vol. 3. IEEE, 2003, pp. 2743–2748.

[76] J. Kinlay, "A comparison of programming languages," 10 2018. [Online]. Available: http://jonathankinlay.com/2018/10/comparison-programming-languages/

[77] MATLAB and MathWorks, "Build a map from lidar data using slam," 2023. [Online]. Available: https://www.mathworks.com/help/vision/ug/build-a-map-from-lidar-data-using-slam.html

[78] ——, "pcdownsample," 2023. [Online]. Available: https://www.mathworks.com/help/vision/ref/pcdownsample.html

Appendix A

Appendix

A.1 RANSAC Pseudocode

Click here to go back to the referring section.

Algorithm 1 Pseudocode for RANSAC taken from [17] (and adapted for this text using [18])

Given:
data - A set of observations.
model - A model to explain the observed data points.
n - The minimum number of data points required to estimate the model parameters.
k - The maximum number of iterations allowed by the algorithm.
t - A threshold value to determine data points that are fit well by the model (inlier).
d - The number of close data points (inliers) required to assert that the model fits well to the data.
Return:
bestFit - The model parameters which may best fit the data (or null if no good model is found).
$iterations \leftarrow 0$
$bestFit \leftarrow$ null
$bestErr \leftarrow$ something really large
while $iterations < k$ **do**
    $maybeInliers \leftarrow n$ randomly selected values from *data*
    $maybeModel \leftarrow model$ parameters fitted to $maybeInliers$
    $confirmedInliers \leftarrow$ empty set
    for every point in *data* **do**
        if point fits $maybeModel$ to an error smaller than t **then**
            add point to $confirmedInliers$
        end if
    end for
    if the number of elements in $confirmedInliers$ is $> d$ **then**
        $betterModel \leftarrow model$ parameters fitted to all the points in $confirmedInliers$
        $thisErr \leftarrow$ a measure of how well $betterModel$ fits these points
        if $thisErr < bestErr$ **then**
            $bestFit \leftarrow betterModel$
            $bestErr \leftarrow thisErr$
        end if
    end if
    increment *iterations*
end while
return *bestFit*

A.2 NN Search with Kd-tree Pseudocode

Click here to go back to the referring section.

Algorithm 2 Nearest Neighbor Search in k-d trees from [7], Adapted using [18]

procedure NNK-DTREE(point p, k-d tree T, distance ε, number k)
    $L \leftarrow$ empty list
    return NNK-DTREEREC(p, root of T, ε, k, L)
end procedure

procedure NNK-DTREEREC(p, k-d tree T, distance ε, number k, list L)
    if $T = \emptyset$ **then**
        return L
    end if
    Extract point p_j from T.root and store it in L if $\|p_j - p\| \leq \varepsilon$.
    if L is larger than k **then**
        Delete the point with largest distance to p from L.
    end if
    if T is just a leaf **then**
        return L
    end if
    if T.root.leftSubtree contains p **then**
        $T_1 = T$.root.leftSubtree
        $T_2 = T$.root.rightSubtree
    else
        $T_2 = T$.root.leftSubtree
        $T_1 = T$.root.rightSubtree
    end if
    NNK-DTREEREC($p, T_1, \varepsilon, k, L$)
    if $|L| < k$ and $\|p - T$.root.hyperplane $\| \leq \varepsilon$ **then**
        NNK-DTREEREC($p, T_2, \varepsilon, k, L$)
    else if $\|L$.farthest $-p\| > \|p - T$.root.hyperplane$\|$ and $\|p - T$.root.hyperplane$\| \leq \varepsilon$ **then**
        NNK-DTREEREC($p, T_2, \varepsilon, k, L$)
    end if
    return L
end procedure

A.3 Range Image Labelling (MATLAB Euclidean Clustering) Pseudocode

Click here to go back to the referring section.

Algorithm 3 Range Image Labelling Algorithm [8], Adapted using [18]

procedure LABELRANGEIMAGE(R)
    Label $\leftarrow 1$
    $L \leftarrow$ zeros(Rows $\times$ Cols)
    for $r = 1$ to Rows **do**
        for $c = 1$ to Cols **do**
            if $L(r, c) = 0$ **then** LABELCOMPONENTBFS($r, c,$ Label)
                Label $\leftarrow$ Label $+ 1$
            end if
        end for
    end for
end procedure
procedure LABELCOMPONENTBFS($r, c,$ Label)
    queue.push(r, c)
    while queue is not empty **do**
        $(r, c) \leftarrow$ queue.top()
        $L(r, c) \leftarrow$ Label
        for each $(r_\mathrm{n}, c_\mathrm{n}) \in$ Neighbourhood(r, c) **do**
            $d_1 \leftarrow \max(R(r, c), R(r_\mathrm{n}, c_\mathrm{n}))$
            $d_2 \leftarrow \min(R(r, c), R(r_\mathrm{n}, c_\mathrm{n}))$
            if $\arctan^2 d_1 - d_2 \sin\psi > \theta$ **then**
                queue.push$(r_\mathrm{n}, c_\mathrm{n})$
            end if
        end for
        queue.pop()
    end while
end procedure

A.4 Code: PCL Implementation

A.4.1 Change File Format: PLY to PCD

Click here to go back to the referring section.

Algorithm 4 Pseudocode to Convert PLY to PCD

```
pathToShow ← 'HuskyLiDAR_Data/lasers'
j ← 0
execute ← 'pcl_ply2pcd -format 1 '
newWorkingDirectory ← 'HuskyLiDAR_Data/lasers'
start_index ← 10
start_index2 ← 13
filenum ← ''
filenumInt ← 0
if exists(newWorkingDirectory)&is_directory(newWorkingDirectory) then
    current_path ← newWorkingDirectory
    print('Changed working directory to: ' + newWorkingDirectory)
else
    error('Failed to change working directory.')
end if
for each entry in directory_iterator(current_path) do
    inputfile ← filename(entry.path)
    if find(inputfile, 'mesh') ≠ −1 then
        filenum ← substr(inputfile, start_index)
        filenumInt ← toInt(filenum)
        outputfile ← 'laserPCDtrial' + format(filenumInt, '01d') + '.pcd'
        j ← j + 1
        outfinal ← execute + inputfile + ' ' + outputfile
        result ← system(outfinal)
    end if
end for
```

A.4.2 Crop to a ROI

Click here to go back to the referring section.

```
//Note; this code is intended to be placed in a main method, and does not show the header file includes
const filesystem::path pathToShow = "HuskyLiDAR_Data/lasers";
int j =0;
const std::filesystem::path newWorkingDirectory = "HuskyLiDAR_Data/lasers";
int start_index =10;
int start_index2 = 13;
std::string filenum = "";
int filenumInt = 0;

if (std::filesystem::exists(newWorkingDirectory) &&
    std::filesystem::is_directory(newWorkingDirectory)) {
    std::filesystem::current_path(newWorkingDirectory);
    std::cout << "Changed working directory to: " << newWorkingDirectory << std::endl;
} else {
    std::cerr << "Failed to change working directory." << std::endl;
}


for (const auto &entry : filesystem::directory_iterator(filesystem::current_path())){
    const auto extension = entry.path().extension().string();
    if (extension == ".pcd") {
        const auto inputfile = entry.path().filename().string(); //get the current filename
        //ensure the file name matches the name of a file to be cropped
            if(inputfile.find("trial") != std::string::npos){
            //load pcd file in
                pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);
                pcl::io::loadPCDFile<pcl::PointXYZ>(inputfile, *cloud);

                // Define a crop box to filter points where X > 0, Z<2.5 (according to dimensions)
                pcl::CropBox<pcl::PointXYZ> cropFilter;
                cropFilter.setInputCloud(cloud);
                cropFilter.setMin(Eigen::Vector4f(-std::numeric_limits<float>::max(), 0.0, -std::numeric_limits<float>::max(), 1.0));
                cropFilter.setMax(Eigen::Vector4f(std::numeric_limits<float>::max(), std::numeric_limits<float>::max(), 1.7, 1.0));

                // Filter the point cloud
                pcl::PointCloud<pcl::PointXYZ>::Ptr croppedCloud(new pcl::PointCloud<pcl::PointXYZ>);
                cropFilter.filter(*croppedCloud);

                //get naming convention
                std::stringstream ss;
                filenum = inputfile.substr(start_index2); //revised to keep the numbering convention
```

```
42                  filenumInt = std::stoi(filenum);
43                  ss << std::setw(1) << std::setfill('0') << filenumInt;
44                  std::string outputfile = "laserPCDcropped" + ss.str() + ".pcd";
45                  j++;
46
47                  //save file
48                  pcl::io::savePCDFileASCII(outputfile, *croppedCloud);
49                  //if (result!=0) {std::cout <<"Failed command"; }}
50              }
51          }
52      }
53
```

Listing A.1: Code to Crop to a Region of Interest for All Files in a Directory

A.4.3 Euclidean Clustering in PCL

Click here to go back to the referring section.

```
// Creating the KdTree object for the search method of the extraction
  pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>)
    ↪ ;
  tree->setInputCloud (cloud_filtered);
//Parameters & Clustering object
  std::vector<pcl::PointIndices> cluster_indices;
  pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
  ec.setClusterTolerance (0.3);
  ec.setMinClusterSize (50);
  ec.setMaxClusterSize (25000);
  ec.setSearchMethod (tree);
  ec.setInputCloud (cloud_filtered);
  ec.extract (cluster_indices);

  int j = 0;
  for (const auto& cluster : cluster_indices)
  {
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new pcl::PointCloud<pcl::
    ↪ PointXYZ>);
    for (const auto& idx : cluster.indices) {
      cloud_cluster->push_back((*cloud_filtered)[idx]);
    } //*
    cloud_cluster->width = cloud_cluster->size ();
    cloud_cluster->height = 1;
    cloud_cluster->is_dense = true;

    std::cout << "PointCloud representing the Cluster: " << cloud_cluster->size () << "
    ↪  data points." << std::endl;
    std::stringstream ss;
    ss << std::setw(1) << std::setfill('0') << j;
    writer.write<pcl::PointXYZ> ("cloud_cluster_" + ss.str () + ".pcd", *cloud_cluster,
    ↪  false); //*
    j++;
  }

```

Listing A.2: PCL Code for Euclidean Clustering [12][15]

A.4.4 ICP in PCL

Click here to go back to the referring section.

```
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/registration/icp.h>
#include <boost/filesystem.hpp>
#include <pcl/console/time.h> //for timing

int main(int argc, char** argv) {
    pcl::console::TicToc tt; //timing
    std::cerr << "Locating your robot...\n", tt.tic ();
    std::string source_dir = "lasersMoving/lasers";//argv[1];
    std::string target_file = "croplaserPCD28.pcd";//argv[2];

    // Load the target point cloud (must be in current directory)
    pcl::PointCloud<pcl::PointXYZ>::Ptr target_cloud(new pcl::PointCloud<pcl::PointXYZ
    ↪ >);
    if (pcl::io::loadPCDFile<pcl::PointXYZ>(target_file, *target_cloud) == -1) {
        std::cerr << "Could not read the target PCD file: " << target_file << std::endl
    ↪ ;
        return 1;
    }

    // Initialize variables to keep track of the best ICP transformation
    double best_score = std::numeric_limits<double>::max();
    Eigen::Matrix4f best_transformation = Eigen::Matrix4f::Identity();
    std::string source_file_best = " ";

    // Iterate through all PCD files in the source directory
    for (const auto& entry : boost::filesystem::directory_iterator(source_dir)) {
        if (entry.path().extension() == ".pcd") {
            std::string source_file = entry.path().string();

            // Load the source point cloud
            pcl::PointCloud<pcl::PointXYZ>::Ptr source_cloud(new pcl::PointCloud<pcl::
    ↪ PointXYZ>);
            if (pcl::io::loadPCDFile<pcl::PointXYZ>(source_file, *source_cloud) == -1)
    ↪ {
                std::cerr << "Could not read the source PCD file: " << source_file <<
    ↪ std::endl;
                continue;
            }

            // Initialise the ICP object
            pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
            icp.setInputSource(source_cloud);
            icp.setInputTarget(target_cloud);

            // Align the source cloud to the target cloud
            pcl::PointCloud<pcl::PointXYZ> aligned_cloud;
```

```
45              icp.align(aligned_cloud);
46
47              // Check the ICP transformation & if it is the best so far
48              if (icp.hasConverged()) {
49                  double score = icp.getFitnessScore();
50                  if (score < best_score) {
51                      best_score = score;
52                      best_transformation = icp.getFinalTransformation();
53                      source_file_best = source_file;
54                  }
55              }
56          }
57      }
58
59      // Output the best transformation matrix
60      std::cout << "Best ICP Transformation Matrix:\n" << best_transformation << std::
    ↪ endl;
61      std::cout << "The estimated location is: \n" << source_file_best<< std::endl;
62      std::cout << "The actual location is: \n" << target_file<< std::endl;
63      if (source_file_best.find(target_file)>0)
64      {
65          std::cerr << "The location is correct.\n" << std::endl;
66      }
67      else {
68          std::cerr << "The localisation failed."<<std::endl;
69      }
70      std::cerr << ">> Elapsed time: " << tt.toc () << " ms\n";
71      return 0;
72  }
73
```

Listing A.3: Code for Implementing ICP Using PCL [15]

A.5 MATLAB Design Code

A.5.1 Load Point Cloud Files into MATLAB

Click here to go back to the referring section.

```
dataFolder = "HuskyLiDAR_Data\";
pointCloudFilePattern = fullfile(dataFolder, 'ply', '*.ply');
plyFiles = dir(pointCloudFilePattern); %get list of directory
%%Error encountered: files were in lexical order
% Extract the numeric part of the file names
fileNumbers = cellfun(@(x) sscanf(x, '%d.ply'), {plyFiles.name});
% Sort the indices based on the extracted numbers
[~, sortedIndices] = sort(fileNumbers);
% Apply the sorted indices to plyFiles to get them in the correct order
plyFiles = plyFiles(sortedIndices);

%Instatiates cell arrays to store data for the table containing .cld data
numFiles = length(plyFiles);
fileNames = cell(numFiles, 1);
pointClouds = cell(numFiles, 1);

%Loops through the ply files and read them
for k = 1:length(plyFiles)
    plyFilename = fullfile(plyFiles(k).folder, plyFiles(k).name);
    pc = pcread(plyFilename);
    % Store the file name and point cloud in the cell arrays
    fileNames{k} = plyFiles(k).name;
    pointClouds{k} = pc;
end
cloudDataTable = table(fileNames, pointClouds, 'VariableNames', {'FileName', '
    ↪ PointCloud'})
```

Listing A.4: Code to Import Point Cloud Files into MATLAB

A.5.2 Capturing the Point Clouds in a Video

Click here to go back to the referring section.

```
%Limits of the player in meters: use to crop to ROI
xlimits = [-7.5 7.5];
ylimits = [0 15];
zlimits = [-10 5];

%Code to record the stream:
videoFile = 'huskyEnviroRecord.avi';
v = VideoWriter(videoFile); %obj to record the video
open(v); %initiate the video recording

%progress bar = why not, we are coding anyways
bar = waitbar(0, 'Processing frames...');

%Create streaming display
lidarPlayer = pcplayer(xlimits,ylimits,zlimits) %option to add arguments for
    ↪ customisation
%player axes labels
xlabel(lidarPlayer.Axes, 'X (m)')
ylabel(lidarPlayer.Axes, 'Y (m)')
zlabel(lidarPlayer.Axes, 'Z (m)')
title(lidarPlayer.Axes, 'Husky LiDAR Sensor Data')

% Skip evey other frame since this is a long sequence
skipFrames  = 2;
numFrames   = height(cloudDataTable);
for n = 2 : skipFrames : numFrames

    % Read a point cloud
    fileName = fileNames{n};
    ptCloud = pointClouds{n};
    % Visualize point cloud
    view(lidarPlayer, ptCloud);

    %add a frame to video file
    frame = getframe(lidarPlayer.Axes.Parent);  % see chatGPT note on Parent
    writeVideo(v, frame);

    % Update waitbar
    waitbar(n/numFrames, bar, sprintf('Processing Frame %d of %d', n, numFrames));
    pause(0.01)
end
close(v); %stop recording video
close(bar);
hide(lidarPlayer)
```

Listing A.5: Code to Create a Video of the Path Traversed by the UGV Using Ordered LiDAR Scans

A.6 Function to Pre-Process Point Clouds in MATLAB

Click here to go back to the referring section.

```
function ptCloudNoGround = PreProcess(ptCloud,egoRadius,cylinderRadius)

    % Select the points inside the cylinder radius and outside the ego radius.
     cylinderIdx = findPointsInCylinder(ptCloud,[egoRadius cylinderRadius]);
     ptCloud = select(ptCloud,cylinderIdx,'OutputSize','full');

    % Remove ground.
     [~,ptCloudNoGround] = segmentGroundSMRF(ptCloud,'ElevationThreshold',0.5);

      %return ptCloudNoGround;
end
```

Listing A.6: MATLAB Function Created to Remove the Ground Plane and Points Associated with the Vehicle

A.7 Function to Organise Point Cloud Data

Click here to go back to the referring section.

```
function OrganisedPCloud = organise(ptCloud)
%to organise the point cloud for the segmentation
     horizontalRes = 512; %channels in horizontal direc, 512
     params = lidarParameters('VLP32C', horizontalRes); %Specific Velodyne Sensor
     OrganisedPCloud = pcorganize(ptCloud,params); %uses MATLAB function
end
```

Listing A.7: Code Showing the Function to Organise a Point Cloud

A.8 Segmentation and Feature Extraction

Click here to go back to the referring section.

```
minNumPoints = 100;
distThreshold = 1;
angleThreshold = 180;

ptCloud1 = ptCloudMap(30);
ptCloud2 = ptCloudMap(31);

ptCloud1 = PreProcess(ptCloud1,egoRadius,cylinderRadius);
ptCloud2 = PreProcess(ptCloud2,egoRadius,cylinderRadius);

ptCloud1 = organise(ptCloud1);
ptCloud2 = organise(ptCloud2);

```

```
labels1 = segmentLidarData(ptCloud1,distThreshold, ...
    angleThreshold,'NumClusterPoints',minNumPoints);
labels2 = segmentLidarData(ptCloud2,distThreshold, ...
    angleThreshold,'NumClusterPoints',minNumPoints);
Feature extraction
[features1,segments1] = extractEigenFeatures(ptCloud1,labels1);
[features2,segments2] = extractEigenFeatures(ptCloud2,labels2);

featureMatrix1 = vertcat(features1.Feature);
featureMatrix2 = vertcat(features2.Feature);
indexPairs = pcmatchfeatures(featureMatrix1,featureMatrix2);

centroids1 = vertcat(features1(indexPairs(:,1)).Centroid);
centroids2 = vertcat(features2(indexPairs(:,2)).Centroid);
```

Listing A.8: Code to Segment Two Point Clouds and Extract their Features [19]

A.9 Full Final Pipeline Code

See the following pages.

Husky Localisation using LiDAR in Unstructured Environments

Kamryn Norton

References: https://www.mathworks.com/help/vision/ug/build-a-map-from-lidar-data-using-slam.html

https://www.mathworks.com/help/lidar/ug/build-a-map-and-localise-using-segment-matching.html

Process files in the directory into pointCloud objects and put them in a table with their file names

```
dataFolder = "HuskyLiDAR_Data";
%can be '.ply' or '.pcd' files
pointCloudFilePattern = fullfile(dataFolder, 'lasers', 'laser_mesh*.ply');
plyFiles = dir(pointCloudFilePattern); %get list of directory
```

Error encountered: files were in lexical order. Extract the numeric part of the file names:

```
fileNumbers = cellfun(@(x) sscanf(x, 'laser_mesh%d.ply'), {plyFiles.name});
% Sort the indices based on the extracted numbers
[~, sortedIndices] = sort(fileNumbers);
% Apply the sorted indices to plyFiles to get them in the correct order
plyFiles = plyFiles(sortedIndices);
```

Instatiate & populate cell arrays to store data for the table containing point cloud data.

```
numFiles = length(plyFiles);
fileNames = cell(numFiles, 1);
pointClouds = cell(numFiles, 1);

%Loops through the ply files and read them
for k = 1:length(plyFiles)
    plyFilename = fullfile(plyFiles(k).folder, plyFiles(k).name);
    pc = pcread(plyFilename);
    % Store the file name and point cloud in the cell arrays
    fileNames{k} = plyFiles(k).name;
    pointClouds{k} = pc;
end
cloudDataTable = table(fileNames, pointClouds, 'VariableNames', {'FileName',
'PointCloud'})
```

Instantiate Processing Parameters

```
dsPercent = 0.5; %downsamplePercent
regGridSize = 1.5; %changed from 3 to 1.5

% Set the cylinder radius and ego radius
cylinderRadius = 40;
egoRadius = 1;
numExpectedFiles =475; %one data set has 451 clouds (indoor3)
```

```
%segmentation params
minNumPoints = 100;
distThreshold = 1;
angleThreshold = 180;
```

SegMatch Map Building

Create Map & Localisation Sets

```
% Select a subset of point cloud scans, and split the data to use for map building
and for localisation.
ptCloudMap = vertcat(pointClouds{1:2:numExpectedFiles-20});
ptCloudLoc = vertcat(pointClouds{2:2:numExpectedFiles-20}); %have to use vertcat
because it is pcd data
%every 2nd odd cloud is map, every 2nd even cloud is localisation
ptCloudMapLabels = fileNames(1:2:numExpectedFiles-10);
ptCloudLocLabels = fileNames(2:2:numExpectedFiles-10);
```

Initial Transform

```
currentViewId = 2;

%Pre-processing
prevPtCloud = PreProcess(ptCloudMap(currentViewId-1), ...
    egoRadius,cylinderRadius);
ptCloud = PreProcess(ptCloudMap(currentViewId), ...
    egoRadius,cylinderRadius);

prevPtCloudDS = pcdownsample(prevPtCloud,'random',dsPercent);
ptCloudDS = pcdownsample(ptCloud,'random',dsPercent);
```

Register & find inital transformation

```
gridStep = 1.5; %size of voxels
relPose = pcregisterndt(ptCloudDS,prevPtCloudDS,gridStep);
locations =[0 0 0]; %to store map locations for analysis

vSet2 = pcviewset;
%initialise pose to an identity grid transformation
initAbsPose = rigidtform3d;

vSet2 = addView(vSet2,currentViewId-1,initAbsPose); %add to view
% absolute pose of the second point cloud using the relative pose estimated during
registration
absPose2 = rigidtform3d(initAbsPose.A * relPose.A);
locations = vertcat(locations,absPose2.Translation); %for analysis
vSet2 = addView(vSet2,currentViewId,absPose2); %add it to the view set.
%connect the 2 views
vSet2 = addConnection(vSet2,currentViewId-1,currentViewId,relPose);
```

```
%transform current pcloud to align to global map
ptCloud = pctransform(ptCloud,absPose2);
```

Segment point cloud and extract features: straight from example

```
prevPtCloud = organise(prevPtCloud);
ptCloud = organise(ptCloud);

%Segment the previous and current point clouds using segmentLidarData.
labels1 = segmentLidarData(prevPtCloud,distThreshold,angleThreshold, ...
    'NumClusterPoints',minNumPoints);
labels2 = segmentLidarData(ptCloud,distThreshold,angleThreshold, ...
    'NumClusterPoints',minNumPoints);
%Extract features from the previous and current point cloud segments using
extractEigenFeatures.
[prevFeatures,prevSegments] = extractEigenFeatures(prevPtCloud,labels1);
[features,segments] = extractEigenFeatures(ptCloud,labels2);
%Track the segments and features using a pcmapsegmatch object. Create an empty
pcmapsegmatch.
segMap = pcmapsegmatch;
segMap = addView(segMap,currentViewId-1,prevFeatures,prevSegments); %previous
features and segments added to pcmapsegmatch
segMap = addView(segMap,currentViewId,features,segments); %current features and
segments added
```

Detect loop closures using findPose

```
[absPoseMap,loopClosureViewId] = findPose(segMap,absPose2);
isLoopClosure = ~isempty(absPoseMap);
%set the absolute pose of the current view without the accumulated drift,
absPoseMap, and the absolute pose of the loop
%closure view, absPoseLoop, to compute the relative pose between the loop closure
poses without the drift.
if isLoopClosure
    absPoseLoop = poses(vSet2,loopClosureViewId).AbsolutePose;
    relPoseLoopToCurrent = rigidtform3d(invert(absPoseLoop).A * absPoseMap.A);
    %Add the loop closure relative to the pose as a connection using addConnection.
    vSet2 = addConnection(vSet2,loopClosureViewId,currentViewId, ...
    relPoseLoopToCurrent);

    prevPoses = vSet2.Views.AbsolutePose;

    %create a posegraph from the view & optimise
    G2 = createPoseGraph(vSet2);
    optimG2 = optimizePoseGraph(G2,'g2o-levenberg-marquardt');
    vSet2 = updateView(vSet2,optimG2.Nodes);

    %find the transformations from the poses before + after drift correction
    optimizedPoses2 = vSet2.Views.AbsolutePose;
```

```
    relPoseOpt = rigidtform3d.empty;
    for k = 1:numel(prevPoses)
        relPoseOpt(k) = rigidtform3d(optimizedPoses2(k).A * invert(prevPoses(k)).A);
    end

    segMap = updateMap(segMap,relPoseOpt); %updating the map segments & centroid
locations
end
```

Build map and correct for accumulated drift error - applied to the rest of the scans

```
% Set the random seed for example reproducibility.
rng(0)

% Update display every 2 scans
figure
updateRate = 2;

% Initialize variables for registration.
prevPtCloud = ptCloudDS;
prevPose = rigidtform3d;

% Keep track of the loop closures to optimize the poses once enough loop closures
are detected.
totalLoopClosures = 0;
tic; %timing for analysis

for i = 3:numel(ptCloudMap)
    ptCloud = ptCloudMap(i);
    % Preprocess and register the point cloud.
    ptCloud = PreProcess(ptCloud,egoRadius,cylinderRadius);
    ptCloudDS = pcdownsample(ptCloud,'random',dsPercent); %changed from Voxel
downsampling
    relPose = pcregisterndt(ptCloudDS,prevPtCloud,gridStep, ... %gridStep defined
earlier =1.5
        'InitialTransform',relPose);
    ptCloud = pctransform(ptCloud,absPose2);

    % Store the current point cloud to register the next point cloud.
    prevPtCloud = ptCloudDS;

    % Compute the absolute pose of the current point cloud.
    absPose2 = rigidtform3d(absPose2.A * relPose.A);
    locations = vertcat(locations,absPose2.Translation); %add to array of locations
    % If the vehicle has moved at least 1 meter since last time, continue through
steps to loop closure detection.
    if norm(absPose2.Translation-prevPose.Translation) >= 1 %Changed to suit
smaller data set, example was every 2m

        ptCloud = organise(ptCloud);
```

```
        % Segment the point cloud and extract features.
        labels = segmentLidarData(ptCloud,distThreshold,angleThreshold, ...
            'NumClusterPoints',minNumPoints);
        [features,segments] = extractEigenFeatures(ptCloud,labels);

        % Keep track of the current view id.
        currentViewId = currentViewId+1;

        % Fixing eigenvalue error (sometimes empty features)
        if ~isa(features, 'eigenFeature') && ~isempty(features)
                features = eig(features);
        end
        if isempty(features)
            [features,segments] = extractEigenFeatures(ptCloudMap(i-1));
        end

        % Add the view to the point cloud view set and map representation.
        vSet2 = addView(vSet2,currentViewId,absPose2);
        vSet2 = addConnection(vSet2,currentViewId-1,currentViewId, ...
            rigidtform3d(invert(prevPose).A * absPose2.A));
        segMap = addView(segMap,currentViewId,features,segments);

        % Update the view set display.
        if mod(currentViewId,updateRate) == 0
            plot(vSet2)
            drawnow
        end

        % Check if there is a loop closure.
        [absPoseMap,loopClosureViewId] =
findPose(segMap,absPose2,'MatchThreshold',1, ...
            'MinNumInliers',5,'NumSelectedClusters',4,'NumExcludedViews',150);
        isLoopClosure = ~isempty(absPoseMap);

        if isLoopClosure
            totalLoopClosures = totalLoopClosures+1;

            % Find the relative pose between the loop closure poses.
            absPoseLoop = poses(vSet2,loopClosureViewId).AbsolutePose;
            relPoseLoopToCurrent = rigidtform3d(invert(absPoseLoop).A *
absPoseMap.A);
            vSet2 = addConnection(vSet2,loopClosureViewId,currentViewId, ...
                relPoseLoopToCurrent);

            % Optimize the graph of poses and update the map every time 3
            % loop closures are detected.
            if mod(totalLoopClosures,3) == 0
                prevPoses = vSet2.Views.AbsolutePose;

                % Correct for accumulated drift: create and optimise pose graph
```

```
                G = createPoseGraph(vSet2);
                optimG = optimizePoseGraph(G,'g2o-levenberg-marquardt');
                vSet2 = updateView(vSet2,optimG.Nodes);

                % Find the transformations from the poses before and after
correcting for drift
                %and use them to update the map segments and centroid locations
using updateMap.
                optimizedPoses2 = vSet2.Views.AbsolutePose;
                relPoseOpt = rigidtform3d.empty;
                for k = 1:numel(prevPoses)
                    relPoseOpt(k) = rigidtform3d(optimizedPoses2(k).A *
invert(prevPoses(k)).A);
                end
                segMap = updateMap(segMap,relPoseOpt);

                % Update the absolute pose after pose graph optimization.
                absPose2 = optimizedPoses2(end);
            end
        end
        prevPose = absPose2; %for iterative registration through each scan
    end
end
timetobuildmap =toc; %for analysis
```

Localise Vehicle in Known Map

Same preprocessing and segmentation for consistency.

```
ptCloud = ptCloudMap(200);

% Preprocess the point cloud.
ptCloud = PreProcess(ptCloud,egoRadius,cylinderRadius);
ptCloud = organise(ptCloud);
% Segment the point cloud and extract features.
labels = segmentLidarData(ptCloud,distThreshold,angleThreshold, ...
    'NumClusterPoints',minNumPoints);
features = extractEigenFeatures(ptCloud,labels);
 segMap = selectSubmap(segMap,[segMap.XLimits segMap.YLimits segMap.ZLimits]);
```

Use the `findPose` object function of `pcmapsegmatch` to localise the vehicle on the prebuilt map.

```
tic; %analysis
 absPoseMap = findPose(segMap,features,'MatchThreshold',30,'MinNumInliers',3)
timeforsegmatch = toc %analysis
```

Visualize the map. Visualize the vehicle on the map as a cuboid.

```
mapColor = [0 0.4 0.7];
mapSegments = pccat(segMap.Segments);
```

```
figure;
hAxLoc = pcshow(mapSegments.Location,mapColor); %plot(vSet2)
title('Localisation on a Prebuilt Map')
view(2)

poseTranslation = absPoseMap.Translation;
quat = quaternion(absPoseMap.Rotation','rotmat','point');
theta = eulerd(quat,'ZYX','point');
pos = [poseTranslation 1 1 1 theta(2) theta(3) theta(1)];
showShape('cuboid',pos,'Color','green',...%'Parent',hAxLoc, ...
    'Opacity',0.8,'LineWidth',0.5)
```

Removed the selection of a submap due to the small nature of the environment traversed.

```
% Visualize the map.
figure('Visible','on')
hAx = plot(vSet2);%pcshow(mapSegments.Location,mapColor);

title('Localisation on a Prebuilt Map')

% Set parameter to update submap.
%submapThreshold = 20;
tic;
% Initialize the poses and previous point cloud for registration.
prevPtCloud = ptCloud;
relPose = rigidtform3d;
prevAbsPose = rigidtform3d;
localisedPoses = [0 0 0];
segMatchLocations = [0 0 0];
% Segment each point cloud and localise by finding segment matches in the map.
for n = 2:numel(ptCloudLoc)
    ptCloud = ptCloudLoc(n);

    % Preprocess the point cloud.
    ptCloud = PreProcess(ptCloud,egoRadius,cylinderRadius);
    ptCloud = organise(ptCloud);
    % Segment the point cloud and extract features.
    labels = segmentLidarData(ptCloud,distThreshold,angleThreshold, ...
        'NumClusterPoints',minNumPoints);
    features = extractEigenFeatures(ptCloud,labels);

    % localise the point cloud using SegMatch.
    smLoc = findPose(sMap,features,'MatchThreshold',30,'MinNumInliers',3);
    if isempty(smLoc)
        smLoc = [0 0 0];
        segMatchLocations = vertcat(segMatchLocations,smLoc);
    else
        segMatchLocations = vertcat(segMatchLocations,smLoc.Translation);
    end
```

```
    % Do registration localisation and analyse against SegMatch Locations above.
        relPose = pcregisterndt(ptCloud,prevPtCloud,gridStep, ...
            'InitialTransform',relPose);
        absPoseMap = rigidtform3d(prevAbsPose.A * relPose.A);

    localisedPoses = vertcat(localisedPoses, absPoseMap.Translation); %for analysis
    hold on; %to build the registration map on the pre-built map
    % Display position estimate in the map.
    poseTranslation = absPoseMap.Translation;
    quat = quaternion(absPoseMap.Rotation','rotmat','point');
    theta = eulerd(quat,'ZYX','point');
    pos = [poseTranslation 0.3 0.3 0.3 theta(2) theta(3) theta(1)];
    showShape('cuboid',pos,'Color','green',...%'Parent',hAx,
        'Opacity',0.4,'LineWidth',0.5)
    legend('Map', 'Location', 'best')

    %for iterative purposes:
    prevAbsPose = absPoseMap;
    prevPtCloud = ptCloud;
end
%hold off;
timeforregistration =toc;
```

ANALYSIS TOOLS

```
figure;
plot(locations,'LineWidth', 1.5)
hold on
plot(localisedPoses)
legend("X Map", "Y Map", "Z Map", "X Loc", "Y Loc", "Z Loc", 'Location','best')
title('NDT Registration Localisation Positions and Map Positions Over All Samples')
xlabel("Sample")
ylabel("Position (m)")
hold off
set(gca,"XGrid","off","YGrid","on")


%SegMatch Localisation vs Map
figure;
plot(segMatchLocations)
hold on
plot(locations, 'LineWidth', 2)
legend("X SM", "Y SM", "Z SM","X Map", "Y Map", "Z Map", 'Location','best')
title('SegMatch Localisation Positions and Map Positions Over All Samples')
xlabel("Sample")
ylabel("Position (m)")
hold off

%Final Error
```

```
xError = abs(locations(end-1,1)-localisedPoses(end,1))
yError = abs(locations(end-1,2)-localisedPoses(end,2))
zError = abs(locations(end-1,3)-localisedPoses(end,3))
barLabels= categorical({'X','Y','Z'});
barData = [xError yError zError];
clear bar;
figure;
bar(barLabels, barData, 0.2)
set(gca,"XGrid","off","YGrid","on")
grid minor
text(1:length(barData),barData,num2str(barData'),'vert','bottom','horiz','center');
title("Graph Showing the Final Error for Localisation")
xlabel("Coordinate")
ylabel("Final Error (m)")

%Distances
store = [];
outliers = [];
for i=1:numel(localisedPoses)
     temp = vecnorm(locations(i)- localisedPoses(i));
     if temp<10
        store(i) = temp;
     else
         outliers(i) = temp; %remove large outliers
     end
end
figure;
plot(store)
hold on;
hold off;
title("Distance Between Localised Point and Map Point")
xlabel("Sample")
ylabel("Distance (m)")
```

```
%Write times to spreadsheet to not lose them
filename = 'testData3.xlsx';
times = table(timetobuildmap, timeforregistration,timeforsegmatch);
writetable(times, filename, 'Sheet', 5, 'Range', 'A1');
```

Helper Functions

```
function ptCloudNoGround = PreProcess(ptCloud,egoRadius,cylinderRadius)

    % Select the points inside the cylinder radius and outside the ego radius.
    cylinderIdx = findPointsInCylinder(ptCloud,[egoRadius cylinderRadius]);
    ptCloud = select(ptCloud,cylinderIdx,'OutputSize','full');

    % Remove ground.
    [~,ptCloudNoGround] = segmentGroundSMRF(ptCloud,'ElevationThreshold',0.5);
```

```
    %return ptCloudNoGround;
end

function OrganisedPCloud = organise(ptCloud)
    %to organise the point cloud for the segmentations - WERE ISSUES HERE
    horizontalRes = 512; %channels in horizontal direc, 512 or 1024
    params = lidarParameters('VLP32C', horizontalRes);
    OrganisedPCloud = pcorganize(ptCloud,params);
end
```

A.10 Ethics Clearance

PRE-SCREENING QUESTIONNAIRE OUTCOME LETTER

STU-EBE-2023-PSQ000521

2023/09/21

Dear Kamryn Norton,

Your Ethics pre-screening questionnaire (PSQ) has been evaluated by your departmental ethics representative. Based on the information supplied in your PSQ, it has been determined that you do not need to make a full ethics application for the research project in question.

You may proceed with your research project titled:

LiDAR Localisation in Unstructured Environments

Please note that should aspect(s) of your current project change, you should submit a new PSQ in order to determine whether the changed aspects increase the ethical risks of your project. It may be the case that project changes could require a full ethics application and review process.

Regards,

Faculty Research Ethics Committee