LiDAR SLAM Pre-processing in ROS2 environment

Thanondrak Arunsangsirinak

June 2024

Abstract

In this work, we present an implementation of the first step of Light Detection and Ranging (LiDAR) based for Simultaneous Localization and Mapping (SLAM)[1], this involves pre-processing the raw data from the LiDAR sensor and manipulating it in a way that it can be processed efficiently in the next step of SLAM. This project utilizes Robot Operating System 2 (ROS 2)[3] with Point Cloud Library (PCL)[5].

However, the project's main goal is to build a building block to develop SLAM or understand it. Since various resources and documentation for SLAM technology are scarce. This project also aims to discuss the algorithms to process the data from scratch while ensuring that the implementation can be done without reinventing the wheel, which is the main idea of ROS.

Overall, we are balancing the deep understanding of SLAM and the quick implementation methodology with ROS 2 and PCL.

Keywords

Light Detection and Ranging, LiDAR, Simultaneous Localization and Mapping, SLAM, LiDAR SLAM, SLAM data acquisition, SLAM pre-processing, SLAM mapping, mapping, autonomous vehicle, AV, unmanned aerial vehicle, UAV, object detection, Computer Vision, CV, Robot Operating System, ROS, Point Cloud Library, PCL

Motivation

I had a dream of an autonomous drone that can independently do 3D scanning and mapping, for unexplored, inhabitable, or dangerous areas.

Around 6 years ago, when I was a student of Physics and Mathematics and not familiar with engineering or robotics. While watching documentation about cave exploration, I thought, what if we can send a robot to explore those caves and generate a 3D map of it, how would one do that? From that moment onwards, I've always had this urge to try to understand how could I make one and what would it take.

Those dreams and questions had always been inside my head, an urge to, one day build it by myself. Today, I have researched and conducted a practical way to reach my dream, hoping that one day, I will be able to make it. Of course, this project is a very small step towards it and I have not been able to keep my focus over the years.

However, the moment I completed this project, I regained my passion and was able to see a glimpse of hope to build my dream with my own hands again. I hope that this work is useful to my successors or whoever trying to research this topic since the resources are very scarce. And I hope it if it does not give you any insights, I should give you hope, an idea, something that will spark the passion inside you.

Before proceeding any further, I would like to thank you for reading this Motivation section.

Streszczenie

W niniejszej pracy przedstawiamy implementację pierwszego kroku w technologii detekcji światła i pomiaru odległości (LiDAR) stosowanej do jednoczesnej lokalizacji i mapowania (SLAM)[1]. Obejmuje to wstępne przetwarzanie surowych danych z czujnika LiDAR oraz manipulację nimi w taki sposób, aby mogły być efektywnie przetwarzane w następnym kroku SLAM. Projekt wykorzystuje system operacyjny dla robotów Robot Operating System 2 (ROS 2)[3] oraz bibliotekę Point Cloud Library (PCL)[5].

Jednak głównym celem projektu jest stworzenie elementu budulcowego do rozwijania technologii SLAM lub jej zrozumienia, ponieważ różne zasoby i dokumentacja dotycząca technologii SLAM są skąpe. Projekt ten ma również na celu omówienie algorytmów do przetwarzania danych od podstaw, jednocześnie zapewniając, że implementacja może być przeprowadzona bez wymyślania koła na nowo, co jest główną ideą ROS.

Ogólnie rzecz biorąc, dążymy do zrównoważenia głębokiego zrozumienia technologii SLAM z szybką metodologią implementacji za pomocą ROS 2 i PCL.

Słowa kluczowe

Detekcja światła i pomiar odległości, LiDAR, jednoczesna lokalizacja i mapowanie, SLAM, LiDAR SLAM, akwizycja danych SLAM, wstępne przetwarzanie SLAM, mapowanie SLAM, mapowanie, pojazd autonomiczny, AV, bezzałogowy pojazd latający, UAV, detekcja obiektów, wizja komputerowa, CV, system operacyjny dla robotów, ROS, biblioteka Point Cloud Library, PCL

Motivation

Miałem marzenie o autonomicznym dronie, który mógłby samodzielnie wykonywać skanowanie 3D i mapowanie nieodkrytych, niezamieszkałych lub niebezpiecznych obszarów.

Około 6 lat temu, kiedy byłem studentem fizyki i matematyki i nie znałem się na inżynierii ani robotyce, oglądając dokumentację na temat eksploracji jaskiń, pomyślałem, co by było, gdybyśmy mogli wysłać robota do eksploracji tych jaskiń i wygenerować ich mapę 3D, jak można by to zrobić? Od tego momentu zawsze miałem to pragnienie, aby spróbować zrozumieć, jak mógłbym to zrobić i co by to wymagało.

Te marzenia i pytania zawsze były w mojej głowie, pragnienie, aby pewnego dnia zbudować to samemu. Dziś przeprowadziłem badania i opracowałem praktyczny sposób na realizację mojego marzenia, mając nadzieję, że pewnego dnia będę w stanie to zrobić. Oczywiście, ten projekt to bardzo mały krok w tym kierunku i nie byłem w stanie utrzymać skupienia przez te lata.

Jednak w momencie, gdy ukończyłem ten projekt, odzyskałem swoją pasję i zobaczyłem iskierkę nadziei, że znowu będę mógł zrealizować swoje marzenie własnymi rękami. Mam nadzieję, że ta praca będzie przydatna dla moich następców lub dla każdego, kto próbuje badać ten temat, ponieważ zasoby są bardzo skąpe. I mam nadzieję, że jeśli nie dostarczy wam żadnych wglądów, to przynajmniej da wam nadzieję, pomysł, coś, co rozpali w was pasję.

Zanim przejdę dalej, chciałbym podziękować za przeczytanie tej sekcji Motywacji.

Contents

1 LiDAR

LiDAR or LADAR, an acronym of "light detection and ranging" or "laser imaging, detection, and ranging") is a method for determining ranges by targeting an object or a surface with a laser and measuring the time for the reflected light to return to the receiver. Lidar may operate in a fixed direction (e.g., vertical) or it may scan multiple directions, in which case it is known as lidar scanning or 3D laser scanning, a special combination of 3-D scanning and laser scanning.[3] Lidar has terrestrial, airborne, and mobile applications.

1.1 LiDAR Sensor

A LiDAR sensor is a sophisticated device made up of various electrical components that work together to measure distances and create detailed 3D maps of environments.

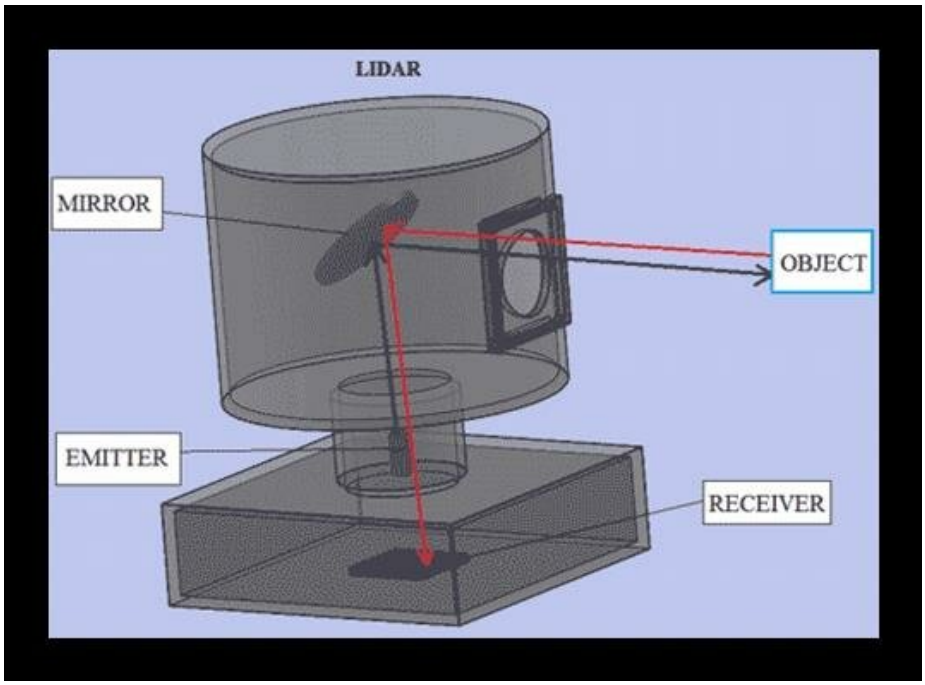

Figure 1: Key components of LiDAR

Below is a detailed breakdown of the key electrical components of a LiDAR sensor and their respective purposes:

- **Emitter:** The main goal of the emitter in a LiDAR system is to produce and transmit laser signals that are employed for ranging. These laser

pulses are aimed at a target and the amount of time it takes the pulse to cover the distance to the target and back is used to determine the distance. Emitter in the LiDAR system is an important element in defining the accuracy, up to which range LiDAR will cover, and the extent of the division of the range.

- **Receiver:** Technically the receiver is accountable for the identification of the Laser pulses that make impacts on the targets and then come back to the sensor. It converts all the optical signals back to electrical signals and these electrical signals are processed to yield the distance of the target. Its major function is to obtain proper and adequate information, which directly determines the efficiency and accuracy of LiDAR.

- **Mirror:** In general, a LiDAR system has employing the mirrors for the management of both transmitted and reflected laser beams in order to realize an appropriate scanning and reflecting of the context. The mirrors are very essential in channeling of the laser pulses to the required area of interest and in focusing the reflected pulses on to the receiver.

1.1.1 Emitter

The emitter normally comprises of a laser diode that is a semiconductor device which gives out light when an electric current is applied in it. This light is in the form of short, bright flashes or steady ones. Exactly, by producing laser pulses and measuring the amount of time the pulse takes to reflect off an object and come back, LiDAR system can measure distance. This is quite crucial in the construction of 3D models and maps of the environment.

1.1.2 Receiver

The essential part of the receiver is photodetector, the role of which is to convert the received light (laser pulses reflected in the to electrical signals). Specific photodetector types that are used in LiDAR systems include the Avalanche photodiodes (APD) and the photomultiplier tube (PMT). APDs are generally used due to their sensitivity and Recovery time. According to the time when the pulses reflected from the targets reach the sensor, the receiver enables computation of distances to one or another target. This is very essential in developing accurate models and maps in the 3D environment.

1.1.3 Mirror

A mirror in LiDAR primarily has a function of directing the laser beam coming out of the laser diode. This can be done by means of handling mirrors which rotates/oscillates to focus the laser pulses on a specific region of interest. Mirrors can guide the emitted laser pulses over the target area so that the LiDAR system can survey vast areas and volumes. It is quite crucial for Notation and Enumeration possessions especially in Surveying measurement capabilities.

1.2 Purpose

1. Topographic Mapping:
   - Purpose: Create detailed 3D maps of the Earth's surface.
   - Applications: Geography, cartography, and urban planning.
   - Benefits: High-resolution elevation data, accurate contour mapping.
2. Forestry and Agriculture:
   - Purpose: Assess vegetation structure, biomass, and land cover.
   - Applications: Forest management, crop monitoring.
   - Benefits: Detailed canopy height models, biomass estimation, and precision agriculture.
3. Autonomous Vehicles:
   - Purpose: Provide real-time 3D perception of the environment for navigation and obstacle detection.
   - Applications: Self-driving cars, drones.
   - Benefits: Accurate distance measurement, reliable object detection, and tracking.

1.3 Alternatives to LiDAR

- Radar (Radio Detection and Ranging): Uses radio waves to detect objects and measure their distance, speed, and other characteristics.
- Sonar (Sound Navigation and Ranging): Uses sound waves to detect objects and measure distances underwater.
- Photogrammetry: Uses photography to measure distances and create 3D models by analyzing the displacement between multiple images.

Technology	Range	Resolution	Environment	Advantages	Limitations
LiDAR	Medium-High	High	Terrestrial, Aerial	High accuracy, detailed 3D maps, works in low light	Affected by weather (rain, fog), higher cost
Radar	High	Medium	All-weather, Terrestrial	Works in poor visibility, long-range	Lower resolution, larger detectable object size
Sonar	Medium-High	Medium	Underwater	Effective underwater, good range	Limited to aquatic environments, lower resolution
Photogrammetry	Low-Medium	High	Aerial, Terrestrial	High-resolution images, cost-effective	Requires clear line of sight, affected by lighting

2 SLAM

This section will discuss our approach to SLAM methodology and how to implement it in a practical approach.

Firstly, we will need to understand what a SLAM is, its purpose, and its general approach to it.

SLAM stands for Simultaneous Localization and Mapping. It's a technology used in robotics and computer vision to help robots or devices understand and navigate their environment in real-time, without prior knowledge of their surroundings.

This project focuses on LiDAR SLAM. A comprehensive definition has been provided in a single sentence to give readers a clear understanding. SLAM addresses the challenges of localization and map-building simultaneously. Specifically, it involves estimating the location of a sensor while simultaneously modeling the environment. Achieving this requires a thorough understanding of sensor information. Sensors observe the external world in specific ways, and the methods for utilizing these observations can vary.

Why is this topic deserving of an entire project, even when only a portion of it is discussed? The complexity lies in performing SLAM in real-time and without prior knowledge. When discussing LiDAR SLAM, involves estimating both the trajectory and the map based on continuous sensor data.

This concept is quite intuitive. When humans enter an unfamiliar environment, they engage in similar activities. The question then becomes whether we can develop programs to enable computers to do the same. At the inception of computer vision, there was a vision that computers might one day emulate human capabilities, observing and understanding the world around them. The ability to explore unknown areas is a compelling and romantic notion, attracting numerous researchers dedicated to this challenge. Initially, it was believed that this task would be manageable, but progress has proven to be more challenging than anticipated.

In computers, elements such as flowers, trees, insects, birds, and animals are recorded as numerical matrices composed of numbers. Enabling computers to interpret the content of images is as challenging as it is for humans to understand these numerical representations. Our understanding of how humans perceive images is limited, and we are still learning how to enable computers to do the same. However, after decades of effort, we are beginning to see signs of success. Advances in Artificial Intelligence (AI) and Machine Learning (ML) technologies are gradually enabling computers to recognize objects, faces, voices, and texts, albeit through probabilistic modeling methods that differ significantly from human processes.

After nearly three decades of development in SLAM, our cameras and sensors have begun to accurately capture their movements and determine their positions. However, there remains a significant gap between the capabilities of computers and humans. Researchers have successfully developed a variety of real-time SLAM systems. Some can efficiently track locations, while others can perform three-dimensional reconstructions in real-time. This progress, though challenging, is remarkable.

Moreover, recent years have seen the emergence of numerous SLAM-related applications. Sensor location capabilities are highly beneficial in various fields: indoor sweeping machines and mobile robots, self-driving cars, Unmanned Aerial Vehicles (UAVs), Virtual Reality (VR), and Augmented Reality (AR). SLAM is critical; without it, sweeping machines would wander blindly, unable to navigate a room autonomously; domestic robots would fail to follow instructions to reach specific locations accurately; and virtual reality devices would be confined to a stationary experience. The absence of these innovations in real life would be a significant loss.

Today, researchers and developers increasingly recognize the importance of SLAM technology. With over 30 years of research history, SLAM has become a prominent topic in both robotics and computer vision communities. Since the beginning of the 21st century, visual SLAM technology has undergone significant theoretical and practical advancements, transitioning from laboratories to real-world applications. Although the theoretical framework of SLAM is now quite mature, implementing a complete SLAM system remains highly challenging and requires substantial technical expertise. New researchers in the field must invest considerable time learning a vast array of scattered knowledge, often encountering several obstacles before grasping the core concepts.

So, as discussed, SLAM generally introduces a few problems to solve the problem, as the name implies, how do we map and localize simultaneously?

1. Localization: Imagine you're in a new place blindfolded. Localization is figuring out where you are relative to your surroundings. In SLAM, this involves the robot determining its own position (like GPS coordinates but indoors or in places without GPS).

2. Mapping: While trying to figure out where it is, the robot also creates a map of the environment. This map could include walls, obstacles, objects, and other features that help the robot navigate.

3. Simultaneous: SLAM does these two tasks simultaneously. As the robot moves through the environment, it updates both its position and the map it's creating in real time. This is crucial because the environment might

change (like people moving around or doors opening), and the robot needs to account for these changes.

2.1 The classical LiDAR SLAM Framework

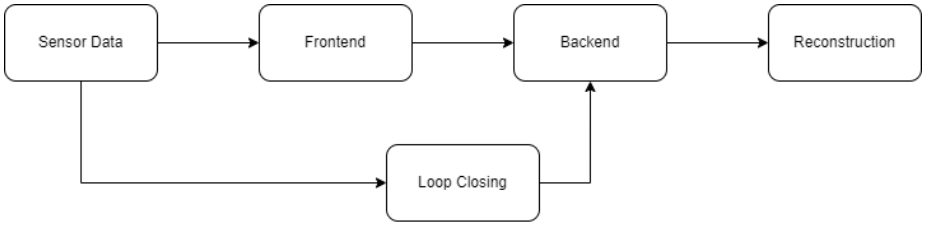

Figure 2: The classical LiDAR SLAM Framework

A typical SLAM workflow includes the following stages:

1. Sensor data acquisition. In LiDAR SLAM, this mainly refers to for acquisition and capturing the raw data from LiDAR. For a mobile robot, this will also include the acquisition and synchronization with motor encoders, IMU sensors, etc.

2. Frontend. In LiDAR SLAM, the frontend refers to the SLAM system where the task is to estimate the sensor's pose (position and orientation) relative to its surroundings and generate a rough local map.

3. Backend. In LiDAR SLAM, The backend is responsible for refining and optimizing the estimated trajectory (pose) of the sensor and constructing a consistent map of the environment based on the processed data from the frontend and loop closing. Because it is connected after the frontend, it is also known as the backend.

4. Loop Closing. Loop closing determines whether the robot has returned to its previous position in order to reduce the accumulated drift. If a loop is detected, it will provide information to the backend for further optimization.

5. Reconstruction. It constructs a task-specific map based on the estimated sensor trajectory.

Since this project will be discussing only methodology and practical implementation for the Pre-processing part of the Sensor data acquisition stage, we will assume that we do not have any problem with the physical sensors and the hardware. We will not be discussing any other steps, however, they will be referred to and described as needed.

2.2 Purpose

SLAM allows robots and devices to explore and understand their world autonomously, enabling them to navigate safely and efficiently without human intervention.

SLAM is used in various applications, from self-driving cars and delivery robots to drones and augmented reality. It relies on sensors like sensors, LiDAR (laser-based radar), and sometimes radar to gather information about the surroundings. Algorithms then process this sensor data to estimate the robot's position and create a map of the environment.

Here are more examples showcasing the diverse applications of SLAM across various fields:

1. Autonomous Navigation:
    - Purpose: SLAM allows robots to move on their own in environments that are complex and unstructured and also helps in the updation of the robots' position (localization) and construction of maps of the environment.
    - Example: AGVs moving on roads or in specific facilities require SLAM for object avoidance and determination of the best route or objective path for delivery.
2. Mapping:
    - Purpose: SLAM builds and updates maps of the environment and they contain information about the landmarks, obstacles, and spatial relation.
    - Example: An example of the use of SLAM is a mobile robot that operates in the unknown building; the robot has to form a detailed map of the building for the subsequent tasks such as inspection, security monitoring, and inventory management.
3. Augmented Reality (AR) and Virtual Reality (VR):
    - Purpose: SLAM can augment the real environment with real time AR and VR devices to place virtual objects or infos over the user's view of reality with a good level of precision.
    - Example: Due to the SLAM technology, applications such as gaming, education/training, and remote assistance are improved as it allows directly positioning of objects as experienced in Microsoft HoloLens AR glasses.
4. Precision Agriculture:
    - Purpose: SLAM is used in agricultural robots allowing them to not only navigate the field and generate maps but also determine the best practices to apply in the field.

- Example: SLAM enabled tractors or drones can cover fields for health inspection, detect disease or nutrient deficiency, spray pesticides or fertilizers selectively and generate maps of the fields for better yield.

5. Indoor Navigation and Smart Spaces:
   - Purpose: SLAM can be used in the environments where GPS cannot work or works very ineffectively, this means that SLAM helps in navigation and mapping of the indoors environment.
   - Example: SLAM is used by robots or smart devices in large buildings, airports, hospitals or shopping malls to move through the corridors, find certain rooms or points of interest and help the visitors and/ or personnel.

2.3 Proposed Approach

Although there are many classifications of SLAM based on various factors such as sensor modality, computational requirements, and specific application scenarios.

In this project, we will be using LiDAR SLAM which utilizes LiDAR (Light Detection and Ranging) sensors, which emit laser pulses to measure distances to objects in the environment. LiDAR sensors provide precise 3D point cloud data, which is used for mapping and localization tasks.

Advantages:

1. Accuracy: LiDAR sensors provide accurate distance measurements, which leads to precise mapping and localization.
2. Robustness: Works well in various lighting conditions, including low light and complete darkness, as LiDAR is not affected by ambient light.
3. Sparse Data Handling: LiDAR directly provides sparse 3D point clouds, making it easier to process and extract features for SLAM.
4. 3D Mapping: Enables the creation of detailed 3D maps of the environment, which can be crucial for navigation in complex terrains.
5. Range: Can detect objects and features at longer distances compared to sensors, enhancing the detection range and mapping capabilities.

Disadvantages:

1. Cost: LiDAR sensors are generally more expensive compared to sensors, which can increase the overall system cost.
2. Limited Color Information: Provides only geometric information and lacks color data, which may be useful for certain applications (e.g., object recognition).
3. Power Consumption: LiDAR sensors can consume more power compared to visual sensors, impacting battery life in mobile applications.

Usage:

1. Autonomous vehicles and drones operating in various outdoor environments.

2. Robotics applications requiring precise 3D mapping and localization, such as industrial automation and warehouse navigation.

2.4 Alternative Approach

We can also discuss another possible selection of SLAM, Visual SLAM, which uses sensors to capture images or video sequences of the environment. It extracts visual features (e.g., key points) from these images to estimate the sensor's pose and construct a map of the surroundings. Although it is possible to combine many types of SLAM for one use case, we will not be discussing it in this project.
Pros:

1. Cost-Effective: sensors are generally more affordable than LiDAR sensors, reducing overall system costs.

2. Rich Information: Provides rich visual information, including colors and textures, which can aid in object recognition and scene understanding.

3. Lightweight: sensors are often lighter and consume less power compared to LiDAR sensors, making them suitable for mobile platforms.

4. Indoor Navigation: Works well in structured environments with sufficient lighting conditions, such as indoor spaces.

Cons:

1. Sensitivity to Lighting: Visual SLAM performance can degrade in low-light conditions or when there are significant changes in lighting.

2. Feature Extraction Challenges: Extracting reliable features from images can be challenging in featureless or dynamically changing environments.

3. Scale Ambiguity: Visual SLAM may suffer from scale ambiguity, where it's difficult to determine absolute scale without additional sensor inputs.

Usage:

1. Mobile robotics applications, such as indoor navigation for robots and drones.

2. Augmented reality (AR) and virtual reality (VR) applications requiring real-time localization and mapping.

3 Pre-processing

Pre-processing refers to the initial stages of data processing where raw sensor data from LiDAR sensors is cleaned, filtered, and prepared for further analysis and use in the SLAM algorithm.

3.1 Purpose

Pre-processing in LiDAR SLAM is essential for preparing raw sensor data to meet the requirements of the SLAM algorithm, ensuring accurate environmental mapping, reliable localization, and efficient real-time operation of robotic and autonomous systems.

It is vital to SLAM that the sensor data is not too large and has useful information for the next step, the frontend. The frontend typically involves feature matching which is responsible for mapping and rough local or global scan matching which is responsible for estimating pose (localization).

For this, we need to make sure that only relevant data is being sent to the frontend since the smaller the data the more efficient SLAM becomes. However this raises another problem, how do we make sure that the data is small enough while containing essential data?

3.2 Proposed Approach

There are many algorithms to help process the raw input from the sensor. Since this project works around autonomous vehicles, we will be utilizing these concepts.

- Noise Removal: To eliminate erroneous points from the LiDAR data that can be caused by sensor inaccuracies or environmental factors.
- Downsampling: To reduce the density of the point cloud data, thereby lowering computational load while retaining important structural information.
- Ground Plane Segmentation: To identify and separate ground points from non-ground points in the point cloud, which helps in focusing on relevant objects and structures.
- Clustering: To group nearby points into clusters representing distinct objects or regions, aiding in object detection and feature extraction.

3.3 Noise Removal and Clustering

Noise removal in LiDAR SLAM involves filtering out erroneous or outlier points from the raw point cloud data collected by LiDAR sensors. These noise points can arise due to various factors, such as sensor inaccuracies, environmental conditions, or reflectivity issues.

Clustering in LiDAR SLAM involves grouping points in a point cloud into clusters based on their spatial proximity. This process helps in identifying and organizing points that belong to the same object or region.

In this project, we will try to combine Noise Removal and Cluster using one method

3.3.1 Proposed Approach

Euclidean Clustering: involves segmenting a point cloud into clusters where each cluster consists of points that are close to each other in Euclidean space. The method relies on a distance threshold to determine whether points belong to the same cluster.

3.3.2 Alternative Approach

DBSCAN is a density-based clustering algorithm that can discover clusters of arbitrary shape and is robust to noise (outliers) but Euclidean Clustering is a much simpler approach to our solution, we will be using it

3.4 Downsampling

Downsampling in LiDAR SLAM involves reducing the number of points in a point cloud while preserving the essential structure and features of the environment. This process is crucial for managing large datasets efficiently and ensuring real-time performance in SLAM systems.

3.4.1 Proposed Approach

In this step, we will be using 2 algorithms to solve this problem.

- Voxel Grid Filter: This method divides the point cloud space into a 3D grid of voxels (cubic cells) and replaces all points within each voxel with a single representative point, typically the centroid.

- Passthrough Filter: A passthrough filter is a simple and effective downsampling method used to limit the point cloud data to a specific range along one or more axes. By filtering out points that fall outside of a defined range, the passthrough filter can reduce the size of the point cloud while focusing on the region of interest.

3.4.2 Alternative Approach

There are however some alternative algorithms, Uniform Sampling, which selects points at regular intervals in the point cloud based on a specified grid size.

However, this algorithm does not preserve the structure of the point cloud as well as the Voxel Grid.

3.5 Ground Plane Segmentation

Ground plane segmentation involves identifying and separating the ground plane (flat surface) from other objects and structures in a LiDAR point cloud. This process is essential for various SLAM tasks, including mapping, obstacle detection, and object recognition, as it helps to differentiate between the ground and non-ground points.

3.5.1 Proposed Approach

- Random Sample Consensus (RANSAC): RANSAC is designed to handle data sets with a significant number of outliers or noise. Its key idea is to iteratively select random subsets of data points (samples) to fit a model and then evaluate the quality of this model based on a predefined criterion (typically a distance threshold).

3.5.2 Alternative Approach

Region Growing Segmentation starts with seed points and grows regions by adding neighboring points that satisfy certain criteria (e.g., smoothness, color similarity, etc.). However, RANSAC is highly robust to noise and outliers. RANSAC explicitly searches for inliers and ignores outliers, making it suitable for datasets with a high level of noise.

4 Installation and Setup

In this section, we will be discussing how to install the necessary software from the following list.

4.1 Operating System and Libraries

- Ubuntu 22.04 LTS: version required by the ROS 2 distro
- ROS 2 Humble: ROS 2 current LTS distribution with EoL set for the May 2027
- Point Cloud Library (PCL): a set of tools for working with Point Cloud data

Assuming that you have installed Ubuntu 22.04 we will not be covering how to install it on your machine. But if you're having some trouble, please visit this website

```
https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview.
```

4.2 ROS 2 Humble

4.2.1 System setup

Set locale Make sure you have a locale that supports UTF-8. If you are in a minimal environment (such as a docker container), the locale may be something minimal like POSIX. We test with the following settings. However, it should be fine if you're using a different UTF-8 supported locale.

```
locale  # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale  # verify settings
```

4.2.2 Add the ROS 2 apt repository

You will need to add the ROS 2 apt repository to your system. First, ensure that the Ubuntu Universe repository is enabled.

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Now add the ROS 2 GPG key with apt.

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key\
    -o /usr/share/keyrings/ros-archive-keyring.gpg
```

Then add the repository to your sources list.

```
echo "deb [arch=$(dpkg --print-architecture)\
    signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]\
    http://packages.ros.org/ros2/ubuntu\
    $(. /etc/os-release && echo $UBUNTU_CODENAME)\
    main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

Install development tools and ROS tools, install common packages, and get ROS 2 Code.

```
sudo apt update && sudo apt install -y \
  python3-flake8-docstrings \
  python3-pip \
  python3-pytest-cov \
  ros-dev-tools
sudo apt install -y \
   python3-flake8-blind-except \
   python3-flake8-builtins \
   python3-flake8-class-newline \
   python3-flake8-comprehensions \
   python3-flake8-deprecated \
   python3-flake8-import-order \
   python3-flake8-quotes \
   python3-pytest-repeat \
   python3-pytest-rerunfailures
mkdir -p ~/ros2_humble/src
cd ~/ros2_humble
vcs import --input\
    https://raw.githubusercontent.com/ros2/ros2/humble/ros2.repos src
sudo apt update && sudo apt upgrade
sudo rosdep init
rosdep update
rosdep install --from-paths src --ignore-src -y\
    --skip-keys "fastcdr rti-connext-dds-6.0.1 urdfdom_headers"
```

Build the code in the workspace If you have already installed ROS 2 another way (either via Debians or the binary distribution), make sure that you run the below commands in a fresh environment that does not have those other installations sourced. Also ensure that you do not have source

```
/opt/ros/${ROS_DISTRO}/setup.bash
```

in your .bashrc. You can make sure that ROS 2 is not sourced with the command

```
printenv | grep -i ROS
```

The output should be empty. More info on working with a ROS workspace can be found in this project.

```
cd ~/ros2_humble/
colcon build --symlink-install
```

Try some examples In one terminal, source the setup file and then run a C++ talker:

```
. ~/ros2_humble/install/local_setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```
. ~/ros2_humble/install/local_setup.bash
ros2 run demo_nodes_py listener
```

4.3 Point Cloud Library (PCL)

Next, PCL. Again, this step is very simple, please enter this into your terminal.

```
sudo apt install libpcl-dev
sudo apt-get install ros-humble-pcl-ros
sudo apt-get install ros-humble-pcl-conversion
sudo apt-get install pcl-tools
```

Now that we have installed and set up everything, I believe it is time for us to move to the next step.

5 Robot Operating System 2 (ROS 2)

We will discuss how to implement the Pre-processing stage practically through a means of software. Firstly, we have to understand that the motivation of this project is to develop our LiDAR SLAM and implement it into an autonomous vehicle.

The commonly used software for handling communication between sensors and the controllers, i.e. motors and switches, is typically ROS or Robot Operating System. Numerous companies, including Toyota Research Institute, Clearpath Robotics, and Boston Dynamics, utilize ROS (Robot Operating System) for developing autonomous vehicles, industrial robots, and advanced robotics technologies.

ROS or Robot Operating System, is a flexible and powerful open-source framework designed to facilitate the development of robotic software. It provides a comprehensive suite of libraries, tools, and capabilities that enable developers to create complex and robust robot applications. Thus, it makes implementing any robotics application simple and does not require the user to **reinvent the wheel**, although, in this project, we will discuss how to reconstruct the wheel.

In this project, we will be using a newer version of ROS, ROS2, which adopts DDS for decentralized and more flexible communication.

To understand how to work with ROS 2, we will divide it into multiple sections and cover only the necessary area.

1. ROS 2: Workspace
2. ROS 2: Package
3. ROS 2: Node
4. ROS 2: Publisher
5. ROS 2: Subscription
6. ROS 2: CMakeList
7. ROS 2: Launch file
8. ROS 2: Rviz

5.1 ROS 2: Workspace

The workspaces are the basic unit of development and deployment. In other words, a ROS2 workspace is a directory where you organize, build, and install ROS 2 packages. It serves as the main working environment for developing ROS 2 applications.
The workspace might look like this.

```
desktop/
    workspace_directory_1/
        src/
    workspace_directory_2/
        src/
    ...
    workspace_directory_n/
        src/
```

To create a workspace, you must also include a directory **src** in it, since ROS 2 uses the Colcon (CMake-based) build system by default. The src directory is a standard location recognized by Colcon where it expects to find the source code of packages. When you invoke Colcon to build packages in your workspace, it scans the src directory to discover packages, read their build instructions (CMakeLists.txt), and compile them accordingly.

5.2 ROS 2: Package

A ROS 2 package is a directory containing software libraries within a package, executables, scripts, configuration files, and other resources that together provide specific functionality or capability within a ROS 2 system.
Your packages in a workspace might look like this.

```
workspace_directory/
    src/
      package_1/
          CMakeLists.txt
          package.xml

      package_2/
          setup.py
          package.xml
          resource/package_2
      ...
      package_n/
          CMakeLists.txt
          package.xml
```

To create a ROS2 package, a ROS2 CLI tool provides a set of commands for creating and managing the packages.
For example, in the **workspace_directory/src**, we will use this command

```
ros2 pkg create --build-type ament_cmake <package_name>
```

When the package is created and the workspace is ready for building, go back to the main workspace directory, **cd ..**, and the only required commands are

- Build the workspace: colcon build
- Source the workspace: source install/setup.bash

After that, the package is ready to be used as an overlay for the currently used ROS 2 environment.

5.3 ROS 2: Node

A node is a fundamental software entity that performs specific tasks within a robotic system. Nodes are executable programs that communicate with each other by publishing and subscribing to messages on ROS 2 topics, and they may also offer or consume services, although, we will not discuss servers and clients in this project. Your nodes in a package might look like this.

```
package_1/
    src/
        node_1.cpp
        node/
            node_2.cpp
        node_3.cpp
    CMakeLists.txt
    package.xml
```

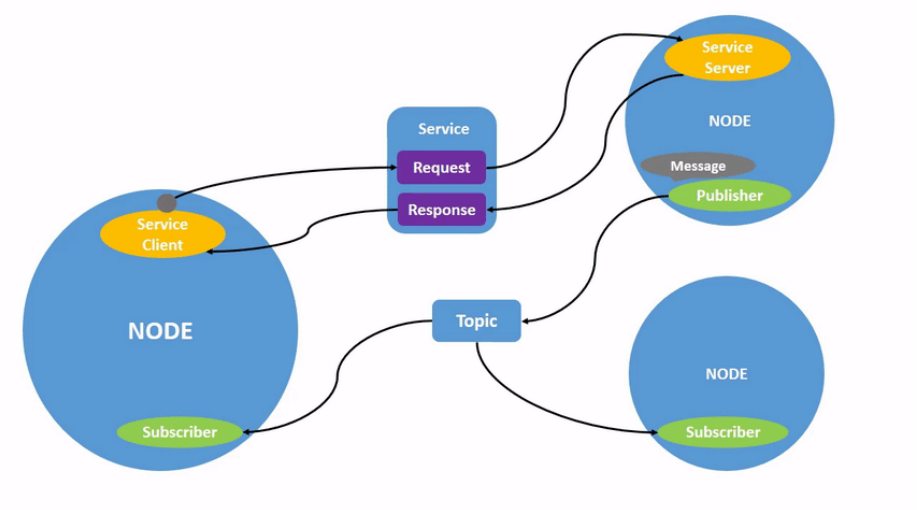

Figure 3: ROS2 nodes communication

To run a ROS2 node, a CLI tool provides some commands that are useful for executing them.
For example, we will run an existing **turtlesim_node** node from **turtlesim** package:

```
ros2 run turtlesim turtlesim_node
```

Here are some other useful commands for ROS2 Node.

Functionality	Command
Launch executable from a package:	ros2 run <pkg_name> <exec_name>
List all currently running nodes' names	ros2 node list
List of all subscribers, publishers, services, and actions	ros2 node info <node_name>

Table 1:

5.4 ROS 2: Topic

ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages. A feature that is currently only topic-specific is the ability to record and playback the data published on a topic. This is a useful feature for debugging and testing, as well as for replaying data from a previous run with ros2bag tool.

Functionality	Command
Nodes and topics graph	rqt_graph
List of topics [with types]	ros2 topic list [-t]
Echo topic data flow	ros2 topic echo <topic_name>
Get message type of the topic	ros2 topic info <topic_name>
Detailed structure of data	ros2 interface show <msg_type>
Publish to topic from CLI	ros2 topic pub [--1 | --rate <x Hz>] <topic_name> <msg_type> '<YAML_args>'
Get topic rate	ros2 topic hz <topic_name>
Record topic	ros2 bag record [-o <file_name>]<topic_name[s]>
Recording info	ros2 bag info <bag_file_name>
Replay recording	ros2 bag play <bag_file_name>

Table 2:

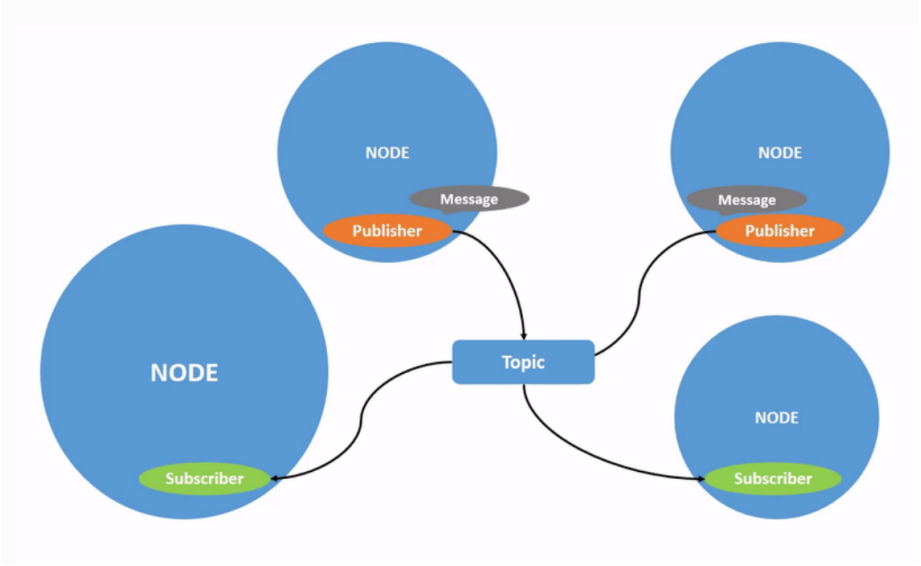

Figure 4: ROS 2 topic

5.5 ROS 2: Publisher

Nodes are executable processes that communicate over the ROS graph. In this section, the nodes will pass information in the form of string messages to each other over a topic. The example used here is a simple "talker" and "listener" system; one node publishes data and the other subscribes to the topic so it can receive that data.
In the workspace directory:

```
cd src
ros2 pkg create --build-type ament_cmake cpp_pubsub
cd cpp_pubsub/src
wget -O publisher_member_function.cpp\
    https://raw.githubusercontent.com/ros2/examples/\
    humble/rclcpp/topics/minimal_publisher/member_function.cpp
```

Now there will be a new file named **publisher_member_function.cpp**.
Open the file using your preferred text editor.

```
#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to register a
* member function as a callback from the timer. */

class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
      timer_ = this->create_wall_timer(
      500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }

  private:
    void timer_callback()
    {
      auto message = std_msgs::msg::String();
```

```
29        message.data = "Hello, world! " + std::to_string(count_++);
30        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
31        publisher_->publish(message);
32      }
33      rclcpp::TimerBase::SharedPtr timer_;
34      rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
35      size_t count_;
36  };
37
38  int main(int argc, char * argv[])
39  {
40    rclcpp::init(argc, argv);
41    rclcpp::spin(std::make_shared<MinimalPublisher>());
42    rclcpp::shutdown();
43    return 0;
44  }
```

Examine the code The top of the code includes the standard C++ headers you will be using. After the standard C++ headers is the **rclcpp/rclcpp.hpp** include which allows you to use the most common pieces of the ROS 2 system. Last is **std_msgs/msg/string.hpp**, which includes the built-in message type you will use to publish data.

```
1  #include <chrono>
2  #include <functional>
3  #include <memory>
4  #include <string>
5
6  #include "rclcpp/rclcpp.hpp"
7  #include "std_msgs/msg/string.hpp"
8
9  using namespace std::chrono_literals;
```

These lines represent the node's dependencies. Recall that dependencies have to be added to **package.xml** and **CMakeLists.txt**, which you'll do in the next section.

The next line creates the node class **MinimalPublisher** by inheriting from **rclcpp::Node**. Every **this** in the code is referring to the node.

```
1  class MinimalPublisher : public rclcpp::Node
```

The public constructor names the node **minimal_publisher** and initializes **count_** to 0. Inside the constructor, the publisher is initialized with the **String** message type, the topic name **topic**, and the required queue size to limit messages in the event of a backup. Next, **timer_** is initialized, which causes the **timer_callback** function to be executed twice a second.

```
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
    timer_ = this->create_wall_timer(
    500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }
```

The **timer_callback** function is where the message data is set and the messages are actually published. The **RCLCPP_INFO** macro ensures every published message is printed to the console.

```
private:
  void timer_callback()
  {
    auto message = std_msgs::msg::String();
    message.data = "Hello, world! " + std::to_string(count_++);
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
    publisher_->publish(message);
  }
```

Last is the declaration of the timer, publisher, and counter fields.

```
rclcpp::TimerBase::SharedPtr timer_;
rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
size_t count_;
```

Following the **MinimalPublisher** class is the **main**, where the node actually executes. **rclcpp::init** initializes ROS 2, and **rclcpp::spin** starts processing data from the node, including callbacks from the timer.

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

5.6 ROS 2: Subscription

Now we will begin discussing about subscription, the counterpart of the publisher. In the workspace directory:

```
cd src/cpp_pubsub/src
wget -O subscriber_member_function.cpp\
    https://raw.githubusercontent.com/ros2/examples/\
    humble/rclcpp/topics/minimal_subscriber/member_function.cpp
```

Check to ensure that these files exist:
publisher_member_function.cpp subscriber_member_function.cpp
Open the **subscriber_member_function.cpp** with your text editor.

```
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
  public:
    MinimalSubscriber()
    : Node("minimal_subscriber")
    {
      subscription_ = this->create_subscription<std_msgs::msg::String>(
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
    }

  private:
    void topic_callback(const std_msgs::msg::String & msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg.data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

Examine the code The subscriber node's code is nearly identical to the publisher's. Now the node is named **minimal_subscriber**, and the constructor

uses the node's **create_subscription** class to execute the callback.

There is no timer because the subscriber simply responds whenever data is published on the **topic** topic.

```
public:
  MinimalSubscriber()
  : Node("minimal_subscriber")
  {
    subscription_ = this->create_subscription<std_msgs::msg::String>(
    "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
  }
```

Recall from the topic tutorial that the topic name and message type used by the publisher and subscriber must match to allow them to communicate.

The **topic_callback** function receives the string message data published over the topic and simply writes it to the console using the **RCLCPP_INFO** macro.

The only field declaration in this class is the subscription.

```
private:
  void topic_callback(const std_msgs::msg::String & msg) const
  {
    RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg.data.c_str());
  }
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription;
```

The **main** function is exactly the same, except now it spins the **MinimalSubscriber** node. For the publisher node, spinning meant starting the timer, but for the subscriber, it simply meant preparing to receive messages whenever they came.

5.7 ROS 2: CMakeList

Now open the CMakeLists.txt file. Below the existing dependency find_package(ament_cmake REQUIRED), add the lines:

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

After that, add the executable and name it talker and listener so you can run your node using ros2 run:

```
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)
```

Finally, add the install(TARGETS...) section so ros2 run can find your executable:

```
install(TARGETS
  talker
  listener
  DESTINATION lib/${PROJECT_NAME})
```

You can clean up your CMakeLists.txt by removing some unnecessary sections and comments, so it looks like this:

```
cmake_minimum_required(VERSION 3.5)
project(cpp_pubsub)

Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)
```

```
22
23 install(TARGETS
24   talker
25   listener
26   DESTINATION lib/${PROJECT_NAME})
27
28 ament_package()
```

5.8 ROS 2: Launch file

We will briefly discuss about Launch file[4] and how to use them.
Open a new terminal and run:

```
ros2 launch turtlesim multisim.launch.py
```

This command will run the following launch file:

```
# turtlesim/launch/multisim.launch.py

from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            namespace= "turtlesim1", package='turtlesim',\
            executable='turtlesim_node', output='screen'),
        launch_ros.actions.Node(
            namespace= "turtlesim2", package='turtlesim',\
            executable='turtlesim_node', output='screen'),
    ])
```

For example, we will try to launch an Rviz package from ROS 2.
In the workspace directory:

```
mkdir src/cpp_pubsub/rviz src/cpp_pubsub/launch
cd src/cpp_pubsub/rviz
curl -OL https://raw.githubusercontent.com/Metastasiz/lidar_slam_preprocessing\
    /main/src/slam_0_bringup/rviz/rviz_kitti.rviz
touch ../launch/my_rviz.launch.py
cd ../launch
```

Now, we will work on the my_rviz.launch.py file.
From the example of the turtlesim launch file above, we will make Rviz launch file.

```
from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='', executable='', output='screen'),
    ])
```

Although, we will need to import some packages for Rviz

```
from ament_index_python.packages import get_package_share_directory
import os
```

Since Rviz is a package of ROS 2, we can run it in the terminal by this

```
ros2 run rviz2 rviz2
```

We will make a launch file of it

```
from launch import LaunchDescription
import launch_ros.actions
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='rviz2', executable='rviz2', output='screen'),
    ])
```

But we want to use our downloaded Rviz file, we can add some parameters for it to launch with flags.
Using this:

```
path_config = os.path.join(get_package_share_directory('slam_0_bringup'),\
'rviz','rviz_kitti.rviz')
arguments=['-d', path_config]
```

Making the launch file looks like this:

```
from launch import LaunchDescription
import launch_ros.actions
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description():
    path_config = os.path.join(get_package_share_directory('cpp_pubsub'),\
    'rviz','rviz_kitti.rviz')
    return LaunchDescription([
        launch_ros.actions.Node(
            package='rviz2', executable='rviz2', output='screen',
            arguments=['-d', path_config]),
    ])
```

Now, we before can launch the Rviz using the launch file, we must edit the CMakeList.txt.
We will be needing this line:

```
install(DIRECTORY
  launch
  rviz
  DESTINATION share/${PROJECT_NAME}/
)
```

Finally, we will be able to launch it but we need to do it in our workspace directory.

```
cd
cd ros2_ws
ros2 launch cpp_pubsub my_rviz.launch.py
```

5.9 ROS 2: Rviz

In this part, we will briefly discuss the purpose of Rviz since we have already talked about how to launch and use it from the discussion above.

RViz (short for "ROS Visualization") is a powerful visualization tool used within the Robot Operating System (ROS) framework, including ROS 2. It is essential for developing, debugging, and visualizing the state and behavior of robots in a simulated or real-world environment.

Here are the primary purposes of RViz in ROS 2:

- Visualization of Sensor Data: RViz allows developers to visualize various types of sensor data, such as LIDAR, camera images, point clouds, and more. This helps in understanding how sensors perceive the environment.
- Robot Model Visualization: It can render the robot's model in a 3D environment using URDF (Unified Robot Description Format) or SDF (Simulation Description Format) files. This helps in visualizing the robot's structure, joint movements, and current configuration.
- Real-Time Monitoring: RViz provides real-time monitoring of the robot's state, including its position, orientation, and any sensor data being published. This is critical for debugging and ensuring the robot is operating as expected.
- Navigation and Path Planning: RViz integrates with navigation stacks to display the robot's planned path, goal positions, and current navigation status. This is useful for developing and testing navigation algorithms.

5.10 ROS 2: rqt_graph

The rqt_graph is a graphical tool within the ROS framework that provides a visual representation of the computational graph in a ROS system. This graph consists of nodes (processes that perform computation) and topics (communication channels for exchanging messages). rqt_graph is part of the rqt suite, which includes various GUI tools for visualizing and interacting with ROS components.

Here are some examples of the main purposes and functionalities of rqt_graph:

- Visualizing Node and Topic Connections: rqt_graph displays nodes and the topics they are publishing to or subscribing from. This helps developers understand how different parts of the system are interconnected and how data flows between them.
- Debugging and Troubleshooting: By visualizing the ROS graph, developers can easily spot misconfigurations, such as nodes that are not correctly connected or topics that have no subscribers. This makes debugging more efficient by providing a clear overview of the system's communication structure.
- Monitoring System State: rqt_graph can show which nodes are currently active and which topics are being published or subscribed to. This real-time monitoring is useful for ensuring that the system is running as expected and for detecting any unexpected changes in the system's behavior.

You can start the rqt_graph by using the rqt_graph command in the terminal if you have a node running that is publishing or subscribed to some other nodes. The rqt_graph will show a visualization representation of it.

Example:

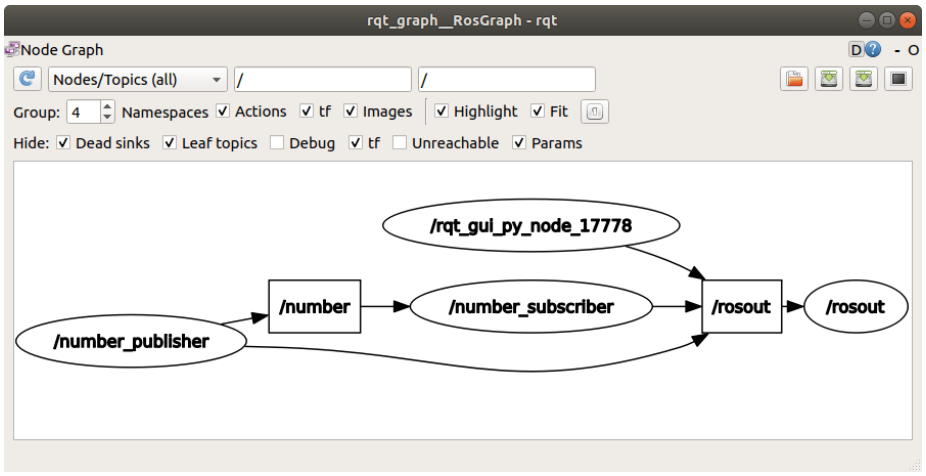

Figure 5: rqt_graph showing communication between nodes

6 Point Cloud Library (PCL)

In the last section, we were downloading and installing PCL. Now, we will discuss about what it is and why do we need it.
The Point Cloud Library (PCL) is a standalone, large-scale, open project for 2D/3D image and point cloud processing. PCL is released under the terms of the BSD license and is thus free for commercial and research use.

```
1 https://pointclouds.org/
```

6.1 What is PCL?

The Point Cloud Library (or PCL) is a large scale, open project [1] for 2D/3D image and point cloud processing. The PCL framework contains numerous state-of-the art algorithms including filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation. These algorithms can be used, for example, to filter outliers from noisy data, stitch 3D point clouds together, segment relevant parts of a scene, extract keypoints and compute descriptors to recognize objects in the world based on their geometric appearance, and create surfaces from point clouds and visualize them – to name a few.

PCL is released under the terms of the 3-clause BSD license and is open source software. It is free for commercial and research use. PCL is cross-platform, and has been successfully compiled and deployed on Linux, MacOS, Windows and Android. To simplify development, PCL is split into a series of smaller code libraries, that can be compiled separately. This modularity is important for distributing PCL on platforms with reduced computational or size constraints (for more information about each module see the documentation page). Another way to think about PCL is as a graph of code libraries, similar to the Boost set of C++ libraries. Here's an example:

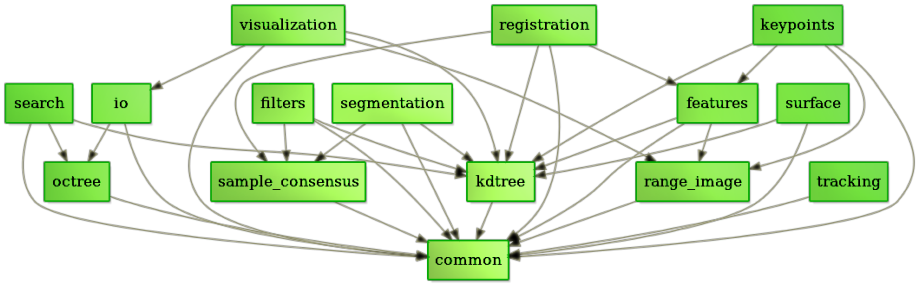

Figure 6: PCL functionality diagram

6.2 What is a Point Cloud?

A point cloud is a data structure used to represent a collection of multi-dimensional points and is commonly used to represent three-dimensional data. In a 3D point

cloud, the points usually represent the X, Y, and Z geometric coordinates of an underlying sampled surface. When color information is present (see the figures below), the point cloud becomes 4D.

Figure 7: Various 3D objects

Point clouds can be acquired from hardware sensors such as stereo cameras, 3D scanners, or time-of-flight cameras, or generated from a computer program synthetically. PCL supports natively the OpenNI 3D interfaces, and can thus acquire and process data from devices such as the PrimeSensor 3D cameras, the Microsoft Kinect or the Asus XTionPro.

6.3 Functionality

The Point Cloud Library (PCL) is a versatile open-source library designed for processing and analyzing 3D point cloud data. It provides a wide range of functionalities and algorithms that enable tasks such as filtering, segmentation, registration, feature extraction, visualization, and more.
Here are some key functionalities offered by the Point Cloud Library (PCL):
Point Cloud I/O:

- PCL provides modules and APIs to read and write point cloud data from various file formats (e.g., PCD, PLY, XYZ) and sensors (e.g., LiDAR, depth cameras).
- Example usage: Loading point cloud data captured from a LiDAR sensor or stored in a PCD file format for processing.

Point Cloud Visualization:

- PCL includes visualization tools for displaying point cloud data in 3D.
- Example usage: Rendering and interactively exploring point cloud scenes to visually inspect data quality or analyze geometric features.

Point Cloud Filtering:

- PCL offers a variety of filtering algorithms to preprocess point clouds, such as downsampling (reducing point density), outlier removal, and noise reduction.
- Example usage: Removing noise from LiDAR scans or downsampling a dense point cloud for faster processing.

Point Cloud Segmentation:

- PCL provides algorithms for segmenting point clouds into distinct regions or clusters based on geometric properties (e.g., plane fitting, region growing).
- Example usage: Segmenting objects from their background in a point cloud to facilitate further analysis or recognition tasks.

Point Cloud Registration:

- PCL supports point cloud registration algorithms to align multiple point clouds into a common coordinate system, such as Iterative Closest Point (ICP) or feature-based methods.
- Example usage: Combining overlapping point clouds from different viewpoints to create a unified 3D model of an environment.

Feature Extraction:

- PCL includes algorithms for extracting keypoints and computing descriptors from point clouds to represent local geometric features (e.g., keypoints, normals, shapes).
- Example usage: Extracting distinctive features from point clouds for object recognition, localization, or matching tasks.

Overall, the Point Cloud Library (PCL) offers a comprehensive set of functionalities that are essential for researchers, developers, and engineers working with 3D point cloud data across various domains, from robotics and autonomous systems to computer vision, augmented reality, and beyond.

7 Methodology

In this section, we will discuss how will we implement our discussed Pre-processing stage and our proposed approach algorithms. We will discuss them in this order.

1. **Initial Project Setup:** ROS 2 setup and Project Structure
2. **Sample Data & Packages:** Sample point cloud data from turtlebot3 and necessary packages
3. **Viewing the point cloud data:** Using PCL to view the sample point cloud data
4. **Adding Executable to the package:** Adding node executable to CMake-List
5. **Downsampling:** Voxel Grid Filter
6. **Downsampling:** Passthrough Filter
7. **Ground Plane Segmentation:** RANSAC for Normal Plane Segmentation
8. **Noise Removal and Clustering:** Euclidean Cluster Indices Extraction

7.1 Initial Project Setup: ROS 2 setup and Project Structure

Assuming that you have followed the installation and setup section. We will create another ROS 2 workspace and packages to use for this project.

In the home directory:

```
1 mkdir -p ros_slam_ws/src
2 echo "source ~/ros_slam_ws/install/setup.bash" >> ~/.bashrc
3 cd ros_slam_ws/src
4 ros2 pkg create slam_1_pc-process --built-type ament_cmake
5 cd ..
6 colcon build && source ~/.bashrc
```

Right now we see that Colcon has completed the build.

7.2 Sample Data & Packages

Let's talk about the sample point cloud data we will use to test our algorithms since we will not be discussing how to obtain this data. We will use my personal data generated from the turtlebot3 simulation. Luckily this data is publically available and able to be downloaded using these simple commands.

In the workspace directory (Please note that line 3 and 4 are should be the same line):

```
mkdir src/slam_1_pc-process/data
cd src/slam_1_pc-process/data
curl -OL https://raw.githubusercontent.com/Metastasiz/lidar_slam_preprocessing\
    /main/src/slam_1_pc-process/data/raw_world.pcd
```

Now you should have **raw_world.pcd** inside your data directory, this is our raw point cloud data.
Next, we will discuss our packages and how to add them to the **CMakeLists.txt**, please proceed to **src** directory

In the home directory:

```
cd ros_slam_ws/src/slam_1_pc-process
```

You will see that a file name **CMakeLists.txt** is there, please open it and we will add the necessary packages.
On line 9, you will see **find_package(ament_cmake REQUIRED)**. We will add the necessary packages under the line 9.

This includes:

- rclcpp: A C++ ROS 2 package
- PCL and pcl_conversions: From PCL package
- sensor_msgs, and visualization_msgs: From ROS 2 package that we will be using

```
find_package(PCL REQUIRED)
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(pcl_conversions REQUIRED)
find_package(visualization_msgs REQUIRED)
```

And we also need to add this line for the package to be able to find necessary executables and directories.

```
install(TARGETS
  DESTINATION lib/${PROJECT_NAME})

install(DIRECTORY
  DESTINATION share/${PROJECT_NAME}/
)
```

Now the file should look something similar to this

```
...
find_package(ament_cmake REQUIRED)
```

```
find_package(PCL REQUIRED)
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(pcl_conversions REQUIRED)
find_package(visualization_msgs REQUIRED)
...
install(TARGETS
  DESTINATION lib/${PROJECT_NAME})

install(DIRECTORY
  DESTINATION share/${PROJECT_NAME}/
)
ament_package()
```

7.3 Viewing the point cloud data

Right now, we have the sample data downloaded, but we need to be able to visualize it. Luckily we are using PCL package and it is very simple to do.
In the workspace directory:

```
pcl_viewer src/slam_1_pc-process/data/raw_world.pcd
```

Right now a window should pop up and we should be able to see something similar to this.

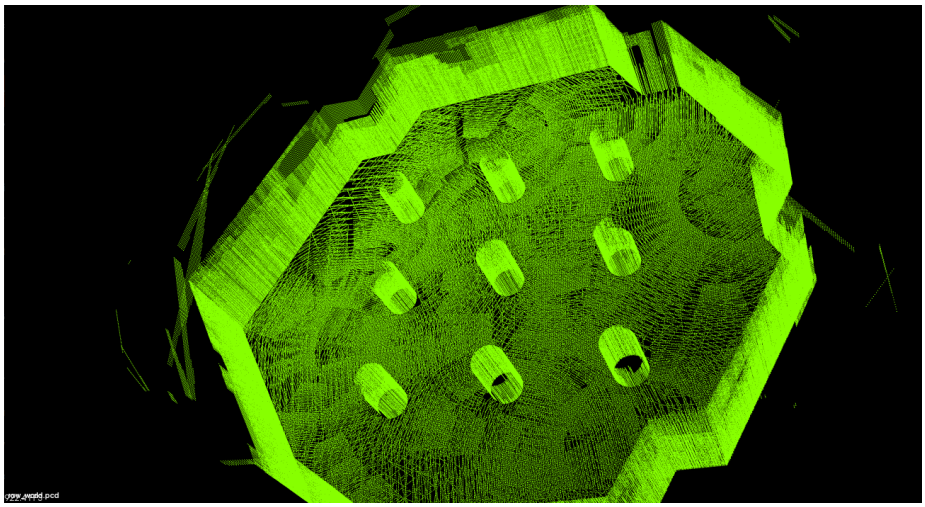

Figure 8: Comparison between before and after

7.4 Adding Executable to the package

As discussed previously in the ROS 2: CMakeList section, we will mention it again very briefly.
If we want to be able to run and ROS 2 node, we have to add an executable to the **CMakeList.txt**. However, adding them is very simple and we will be discussing it a little bit.
Please open CMakeList.txt of the package and we will be adding an example above this line.

```
install(TARGETS
  DESTINATION lib/${PROJECT_NAME})
```

Firstly, we need to add an executable

```
add_executable(3_voxel_grid src/3_voxel_grid.cpp)
target_link_libraries(3_voxel_grid ${PCL_LIBRARIES})
```

This means that we are adding an executable name **3_voxel_grid** from this path **src/3_voxel_grid.cpp**.
Next, we will need to add it to the target also

```
install(TARGETS
    3_voxel_grid
  DESTINATION lib/${PROJECT_NAME})
```

Now the file should look something similar to this

```
...
add_executable(3_voxel_grid src/3_voxel_grid.cpp)
target_link_libraries(3_voxel_grid ${PCL_LIBRARIES})
...
install(TARGETS
    3_voxel_grid
  DESTINATION lib/${PROJECT_NAME})
...
```

We will not be discussing this any further, so please remember to add this when you make a new node.

7.5 Downsampling: Voxel Grid Filter

A Voxel Grid Filter is a common downsampling technique used in point cloud processing. It reduces the number of points in a point cloud, which helps in improving computational efficiency while retaining the overall structure of the data.

7.5.1 Purpose

- Define Voxel Size: Determine the size of the voxels. This size controls the resolution of the downsampled point cloud. Smaller voxel sizes result in a higher resolution point cloud but less downsampling, while larger voxel sizes lead to more downsampling but lower resolution.
- Create a 3D Grid: Overlay a 3D grid of voxels over the entire point cloud. Each voxel is a cube with sides of length equal to the specified voxel size.
- Assign Points to Voxels: Assign each point in the point cloud to a voxel based on its coordinates.
- Compute Centroids: For each voxel that contains points, compute a representative point. This is typically the centroid (average position) of all the points within that voxel.
- Generate Downsampled Point Cloud: Replace all the points within each voxel with the corresponding centroid point. The resulting point cloud is the downsampled version of the original point cloud.

7.5.2 Explanation

A Voxel Grid Filter works by creating a 3D grid of voxels (volumetric pixels) over the input point cloud data and then replacing the points inside each voxel with a representative point, typically the centroid of all the points within the voxel.
The process of applying a Voxel Grid Filter involves the following steps:

1. Define Voxel Size: Determine the size of the voxels. This size controls the resolution of the downsampled point cloud. Smaller voxel sizes result in a higher resolution point cloud but less downsampling, while larger voxel sizes lead to more downsampling but lower resolution.
2. Create a 3D Grid: Overlay a 3D grid of voxels over the entire point cloud. Each voxel is a cube with sides of length equal to the specified voxel size.
3. Assign Points to Voxels: Assign each point in the point cloud to a voxel based on its coordinates.
4. Compute Centroids: For each voxel that contains points, compute a representative point. This is typically the centroid (average position) of all the points within that voxel.

5. Generate Downsampled Point Cloud: Replace all the points within each voxel with the corresponding centroid point. The resulting point cloud is the downsampled version of the original point cloud.

Here is a pseudocode example:

```
def voxel_grid_filter(point_cloud, voxel_size):
    voxel_grid = {}

    for point in point_cloud:
        voxel_index = (
            int(point.x / voxel_size),
            int(point.y / voxel_size),
            int(point.z / voxel_size)
        )

        if voxel_index not in voxel_grid:
            voxel_grid[voxel_index] = []

        voxel_grid[voxel_index].append(point)

    downsampled_points = []

    for voxel in voxel_grid.values():
        centroid = compute_centroid(voxel)
        downsampled_points.append(centroid)

    return downsampled_points

def compute_centroid(points):
    x = sum(point.x for point in points) / len(points)
    y = sum(point.y for point in points) / len(points)
    z = sum(point.z for point in points) / len(points)
    return Point(x, y, z)
```

7.5.3 Implementation

Let's begin by creating the cpp file name **3_voxel_filter.cpp** inside **src** of the **slam_1_pc-process** package

```
touch src/slam_1_pc-process/src/3_voxel_filter.cpp
```

Since we are using the PCL library, implementing this process is very simple and relieves us from having to code the Voxel Grid Filter from scratch.
From the code below, the main idea here is, that in line 35, we set the X, Y, and Z **voxel_size** similar to the pseudocode.
Here is our practical implementation using PCL in **3_voxel_filter.cpp**:

```cpp
#include <iostream>
#include <filesystem>

#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>

#include <pcl/filters/voxel_grid.h>

typedef pcl::PointXYZ PointT;

int main()
{
    // ********************************         Import PCD
    pcl::PointCloud<PointT>::Ptr input_cloud (new pcl::PointCloud<PointT>);

    pcl::PCDReader cloud_reader;
    pcl::PCDWriter cloud_writer;

    std::string import_file_pcd         = "raw_world.pcd";
    std::string export_file_pcd_before  = "3_world_before.pcd";
    std::string export_file_pcd_after   = "3_world_after.pcd";
    std::string dir_pcd                 = "slam_1_pc-process/data";
    std::filesystem::path path_ros_ws   = std::filesystem::current_path();
    std::filesystem::path path_dir_pcd  = path_ros_ws/"src"/dir_pcd;
    std::string path_import             = path_dir_pcd/import_file_pcd;
    std::string path_export_before      = path_dir_pcd/export_file_pcd_before;
    std::string path_export_after       = path_dir_pcd/export_file_pcd_after;

    cloud_reader.read(path_import,*input_cloud);

    // ********************************   Voxel Filter
    pcl::PointCloud<PointT>::Ptr export_cloud (new pcl::PointCloud<PointT>);

    pcl::VoxelGrid<PointT> voxel_filter;
    voxel_filter.setLeafSize(0.05 , 0.05, 0.05);
    //
    voxel_filter.setInputCloud(input_cloud);
    voxel_filter.filter(*export_cloud);

    // ********************************         Export PCD

    cloud_writer.write<PointT>(path_export_before,*input_cloud);
    cloud_writer.write<PointT>(path_export_after,*export_cloud);
    std::cout << "Saved PCD to: " << path_export_before << std::endl;
    std::cout << "Saved PCD to: " << path_export_after << std::endl;
```

```
46
47    return 0;
48 }
```

7.5.4 Result

Let's check the result with this command

```
1 colcon build && source ~/.bashrc
2 ros2 run slam_1_pc-process 3_voxel_filter.cpp
3 pcl_viewer -multiview 2 src/slam_1_pc-process/data/3*
```

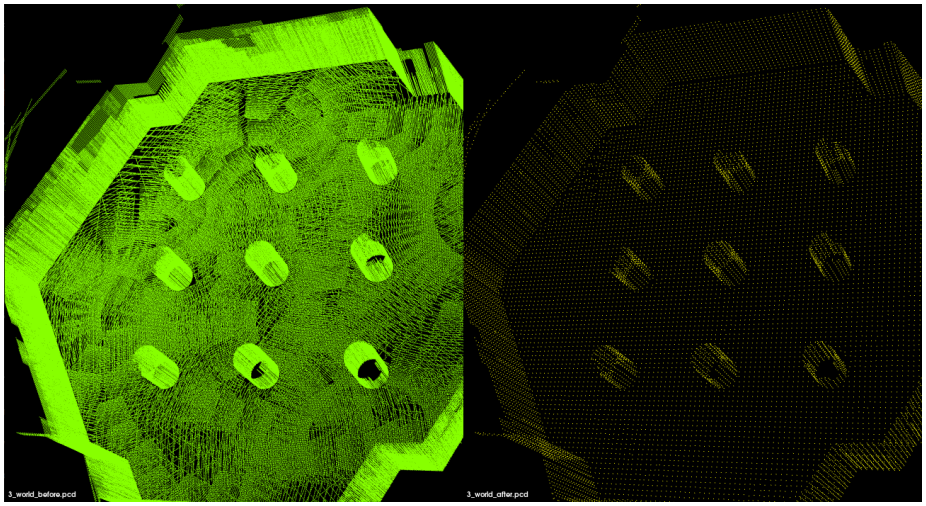

Figure 9: Comparison between before and after

We can see that the point cloud data size has been decreased and is now in a grid compared to the figure on the right.

7.6 Downsampling: Passthrough Filter

A Passthrough Filter is a simple and efficient technique used in point cloud processing to filter out points based on their spatial location along specified dimensions (usually x, y, and z axes). Unlike downsampling methods that reduce the number of points by merging or averaging them, a Passthrough Filter reduces the dataset by discarding points outside a defined range along a particular axis. This is particularly useful for focusing on regions of interest within a point cloud.

7.6.1 Purpose

- Region of Interest Extraction: To focus on a specific region of the point cloud by filtering out points that are outside the area of interest.
- Noise Reduction: To remove outliers and irrelevant points that lie outside the expected range, thereby cleaning up the point cloud.

7.6.2 Explanation

The Passthrough Filter works by specifying a range of acceptable values along one or more axes. Points that fall outside these ranges are removed from the point cloud, effectively "passing through" only the points within the desired bounds.
The process of applying a Passthrough Filter involves the following steps:

1. Specify Filter Bounds: Define the minimum and maximum values for the axis (or axes) of interest. For example, to filter points along the z-axis, you might specify a range of [z_min, z_max].
2. Apply Filter: Iterate through the point cloud and keep only the points that fall within the specified bounds for the selected axis (or axes).

Here is a pseudocode example:

```
def passthrough_filter(point_cloud, axis, axis_min, axis_max):
    filtered_points = []

    for point in point_cloud:
        if axis == 'x' and axis_min <= point.x <= axis_max:
            filtered_points.append(point)
        elif axis == 'y' and axis_min <= point.y <= axis_max:
            filtered_points.append(point)
        elif axis == 'z' and axis_min <= point.z <= axis_max:
            filtered_points.append(point)

    return filtered_points
```

7.6.3 Implementation

Let's begin by creating the cpp file name **4_passthrough.cpp** inside **src** of the **slam_1_pc-process** package

```
touch src/slam_1_pc-process/src/4_passthrough.cpp
```

Again, we will be using the PCL library, implementing this process is very simple and relieves us from having to code from scratch.
From the code below, the main idea here is, that in lines 37 and 45, we set the X and Y **axis_min** and **axis_max** similar to the pseudocode.
Here is our practical implementation using PCL in **4_passthrough.cpp**:

```
#include <iostream>
#include <filesystem>

#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>

#include <pcl/filters/passthrough.h>

typedef pcl::PointXYZ PointT;

int main()
{
    // ******************************         Import PCD
    pcl::PointCloud<PointT>::Ptr input_cloud (new pcl::PointCloud<PointT>);

    pcl::PCDReader cloud_reader;
    pcl::PCDWriter cloud_writer;

    std::string import_file_pcd         = "3_world_after.pcd";
    std::string export_file_pcd_before  = "4_world_before.pcd";
    std::string export_file_pcd_after   = "4_world_after.pcd";
    std::string dir_pcd                 = "slam_1_pc-process/data";
    std::filesystem::path path_ros_ws   = std::filesystem::current_path();
    std::filesystem::path path_dir_pcd  = path_ros_ws/"src"/dir_pcd;
    std::string path_import             = path_dir_pcd/import_file_pcd;
    std::string path_export_before      = path_dir_pcd/export_file_pcd_before;
    std::string path_export_after       = path_dir_pcd/export_file_pcd_after;

    cloud_reader.read(path_import,*input_cloud);

    // ******************************   Voxel Filter
    pcl::PointCloud<PointT>::Ptr export_cloud (new pcl::PointCloud<PointT>) ;

    // Along X Axis
```

```
35      pcl::PassThrough<PointT> passing_x;
36      passing_x.setFilterFieldName("x");
37      passing_x.setFilterLimits(-1.7,1.7);
38      //
39      passing_x.setInputCloud(input_cloud);
40      passing_x.filter(*export_cloud);
41
42      // Along Y Axis
43      pcl::PassThrough<PointT> passing_y;
44      passing_y.setFilterFieldName("y");
45      passing_y.setFilterLimits(-1.7,1.7);
46      //
47      passing_y.setInputCloud(export_cloud);
48      passing_y.filter(*export_cloud);
49
50      // ******************************         Export PCD
51
52      cloud_writer.write<PointT>(path_export_before,*input_cloud);
53      cloud_writer.write<PointT>(path_export_after,*export_cloud);
54      std::cout << "Saved PCD to: " << path_export_before << std::endl;
55      std::cout << "Saved PCD to: " << path_export_after << std::endl;
56
57    return 0;
58  }
```

7.6.4 Result

```
colcon build && source ~/.bashrc
ros2 run slam_1_pc-process 4_passthrough.cpp
pcl_viewer -multiview 2 src/slam_1_pc-process/data/4*
```

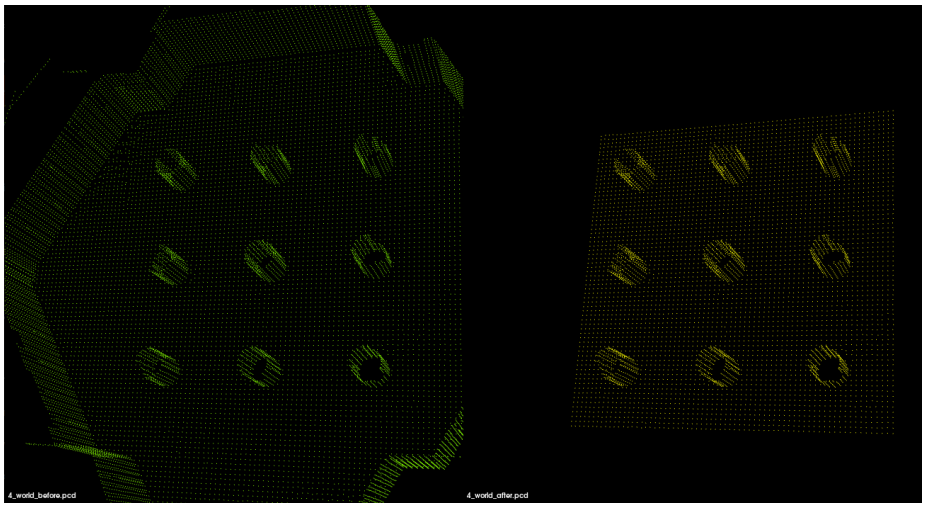

Figure 10: Comparison between before and after

We can see that the point cloud data size has been decreased and is now in a rectangular shape due to any point cloud outside of the axis min and max being completely removed.

7.7 Ground Plane Segmentation: RANSAC for Normal Plane Segmentation

RANSAC (Random Sample Consensus) is a robust algorithm for fitting models to data that contains a significant proportion of outliers. In the context of normal plane segmentation in point clouds, RANSAC is used to identify and extract planar regions from a 3D point cloud. By incorporating Normal Distance Weight, the algorithm improves the robustness and accuracy of the plane fitting process.

7.7.1 Purpose

- Robust Plane Detection: Identify and segment planar surfaces in a point cloud, even with significant noise and outliers.
- Improved Accuracy: By weighting points based on their distance to the plane, the resulting plane model is more accurately fitted to the data.
- Data Simplification: Reduce the complexity of the point cloud by extracting meaningful geometric structures.
- Plane Segmentation: Segmenting Plane regions from object or vertical regions from point cloud

7.7.2 Explanation

RANSAC for normal plane segmentation involves iteratively selecting random subsets of points, fitting a plane to these points, and then determining the number of inliers (points that lie close to the plane). The Kd-Tree could be used for Normals estimation.
The process of applying a Plane Segmentation involves the following steps:

1. Initialization: Define the plane model to fit and set parameters such as the maximum number of iterations, distance threshold, and minimum number of points required to define the model.
2. Random Sampling: Randomly select a minimal subset of points required to fit a plane. For a plane in 3D, this requires three points.
3. Model Fitting:
    - Fit a plane to the selected points by computing the plane equation $\mathbf{aX + bY + cZ + d = 0}$
    - Inlier Counting with Weighting: For each point in the point cloud, calculate its perpendicular distance to the plane.
    - Determine if the point is an inlier based on a distance threshold.
    - Compute a weight for each inlier based on its distance: weight = max(0, 1 - distance/distance_threshold). Accumulate a weighted error, where closer points have higher weights.

4. Iteration: Repeat the random sampling, model fitting, and inlier counting steps for a predefined number of iterations or until a model with a sufficiently high number of weighted inliers is found.

5. Selection: Select the model with the highest number of inliers or the lowest weighted error as the best-fit plane.

Here is a pseudocode example:

```
import numpy as np
import random

def ransac_plane_segmentation(points, max_iterations, distance_threshold):
    best_plane = None
    best_inliers = []
    best_weighted_error = float('inf')

    for _ in range(max_iterations):
        # Randomly sample three points
        sample = random.sample(list(points), 3)
        p1, p2, p3 = sample[0], sample[1], sample[2]

        # Compute the plane coefficients (a, b, c, d) from the sampled points
        normal = np.cross(p2 - p1, p3 - p1)
        a, b, c = normal
        d = -np.dot(normal, p1)
        normal_length = np.linalg.norm(normal)

        inliers = []
        weighted_error = 0

        # Determine the inliers for the current plane model
        for point in points:
            distance = abs(a * point[0] + b * point[1] + c * point[2] + d)\
                / normal_length
            if distance < distance_threshold:
                inliers.append(point)
                # Calculate the weight based on the distance
                weight = max(0, 1 - (distance / distance_threshold))
                weighted_error += distance * weight

        # Update the best model if the current one has more inliers
        # or a lower weighted error
        if  len(inliers) > len(best_inliers)\
            or (len(inliers) == len(best_inliers)\
            and weighted_error < best_weighted_error):
            best_inliers = inliers
```

```
            best_plane = (a, b, c, d)
            best_weighted_error = weighted_error

    return best_plane, best_inliers
```

7.7.3 Implementation

Let's begin by creating the cpp file name **5_segmentation.cpp** inside **src** of the **slam_1_pc-process** package

```
touch src/slam_1_pc-process/src/5_segmentation.cpp
```

Again, we will be using the PCL library, implementing this process is very simple and relieves us from having to code from scratch.
From the code below, the main idea here is, that on line 47, we use KD-tree for the method to estimate our normals which later on being used for segmentation on line 63. We also set the weight, max iteration, and distance threshold as described in the pseudocode. Lastly, we will need to set the negative to true on line 69 since we want the opposite, objects and walls, of the plane of the point cloud.
Here is our practical implementation using PCL in **5_segmentation.cpp**:

```
#include <iostream>
#include <filesystem>

#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>

#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/segmentation/extract_clusters.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/features/normal_3d.h>

typedef pcl::PointXYZ PointT;

int main()
{
    // ******************************          Import PCD
    pcl::PointCloud<PointT>::Ptr input_cloud (new pcl::PointCloud<PointT>);

    pcl::PCDReader cloud_reader;
    pcl::PCDWriter cloud_writer;

    std::string import_file_pcd         = "4_world_after.pcd";
```

```
25      std::string export_file_pcd_before  = "5_world_before.pcd";
26      std::string export_file_pcd_after   = "5_world_after.pcd";
27      std::string dir_pcd                 = "slam_1_pc-process/data";
28      std::filesystem::path path_ros_ws   = std::filesystem::current_path();
29      std::filesystem::path path_dir_pcd  = path_ros_ws/"src"/dir_pcd;
30      std::string path_import             = path_dir_pcd/import_file_pcd;
31      std::string path_export_before      = path_dir_pcd/export_file_pcd_before;
32      std::string path_export_after       = path_dir_pcd/export_file_pcd_after;
33
34      cloud_reader.read(path_import,*input_cloud);
35
36      //  ******************************    Plane Segmentation
37      pcl::PointCloud<PointT>::Ptr export_cloud (new pcl::PointCloud<PointT>) ;
38
39      pcl::PointCloud<pcl::Normal>::Ptr plane_normals\
40          (new pcl::PointCloud<pcl::Normal>);
41      pcl::PointIndices::Ptr plane_inliers(new pcl::PointIndices);
42      pcl::ModelCoefficients::Ptr plane_coefficients(new pcl::ModelCoefficients);
43
44      //  Extracting Normals
45      pcl::NormalEstimation<PointT, pcl::Normal> normal_extractor;
46      pcl::search::KdTree<PointT>::Ptr tree(new pcl::search::KdTree<PointT>());
47      normal_extractor.setSearchMethod(tree);
48      normal_extractor.setKSearch(30);
49      //
50      normal_extractor.setInputCloud(input_cloud);
51      normal_extractor.compute(*plane_normals);
52
53      //  Parameters for Plane Segmentation
54      pcl::SACSegmentationFromNormals<PointT, pcl::Normal> plane_segmentator_norms;
55      plane_segmentator_norms.setOptimizeCoefficients(true);
56      plane_segmentator_norms.setModelType(pcl::SACMODEL_NORMAL_PLANE);
57      plane_segmentator_norms.setMethodType(pcl::SAC_RANSAC);
58      plane_segmentator_norms.setNormalDistanceWeight(0.5);
59      plane_segmentator_norms.setMaxIterations(100);
60      plane_segmentator_norms.setDistanceThreshold(0.4);
61      //
62      plane_segmentator_norms.setInputCloud(input_cloud);
63      plane_segmentator_norms.setInputNormals(plane_normals);
64      plane_segmentator_norms.segment(*plane_inliers,*plane_coefficients);
65
66      //  Extracting from indices
67      pcl::ExtractIndices<PointT> plane_extract_indices;
68      plane_extract_indices.setIndices(plane_inliers);
69      plane_extract_indices.setNegative(true);
70      //
```

```
    plane_extract_indices.setInputCloud(input_cloud);
    plane_extract_indices.filter(*export_cloud);

    // ******************************         Export PCD

    cloud_writer.write<PointT>(path_export_before,*input_cloud);
    cloud_writer.write<PointT>(path_export_after,*export_cloud);
    std::cout << "Saved PCD to: " << path_export_before << std::endl;
    std::cout << "Saved PCD to: " << path_export_after << std::endl;

  return 0;
}
```

7.7.4 Result

```
colcon build && source ~/.bashrc
ros2 run slam_1_pc-process 5_segmentation.cpp
pcl_viewer -multiview 2 src/slam_1_pc-process/data/5*
```

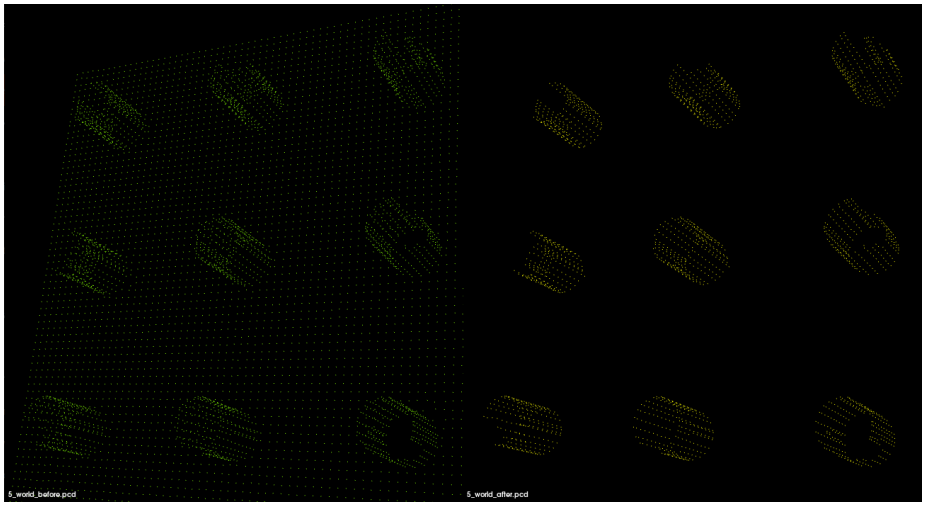

Figure 11: Comparison between before and after

We can see that in the point cloud has no plane in the after figure, this is a good thing because now we do not have to worry about large data size from the plane region and only prioritize on the objects and walls.

7.8 Noise Removal and Clustering: Euclidean Cluster Indices Extraction

Euclidean Cluster Indices Extraction is a method used in point cloud processing to group points into clusters based on their spatial proximity. This technique is particularly useful for noise cancellation and clustering tasks. By identifying and isolating clusters, it becomes easier to filter out noise and analyze distinct objects or regions within the point cloud.

7.8.1 Purpose

- Noise Cancellation: By identifying clusters, it is possible to filter out isolated points that are likely to be noise. Small clusters or single points that do not meet a minimum cluster size threshold can be considered noise and removed from the dataset.
- Clustering: The technique is used to group points into meaningful clusters, which can represent different objects or features in the point cloud. This is useful for tasks such as object detection, scene understanding, and segmentation.

7.8.2 Explanation

Euclidean Cluster Indices Extraction involves segmenting a point cloud into clusters where each cluster consists of points that are close to each other in Euclidean space. The method relies on a distance threshold to determine whether points belong to the same cluster.

1. Kd-Tree Construction: Build a Kd-Tree (k-dimensional tree) from the point cloud data to facilitate efficient nearest neighbor searches.
2. Initialization: Start with an empty list of clusters and mark all points as unvisited.
3. Seed Point Selection: Select an unvisited point as a seed and mark it as visited.
4. Neighbor Search: Use the Kd-Tree to find all points within a predefined distance (tolerance) from the seed point.
5. Cluster Growth: Add the seed point and its neighbors to a new cluster. Recursively expand the cluster by searching for neighbors of each point in the cluster.
6. Cluster Completion: Once no more neighbors are found within the distance threshold, finalize the cluster.
7. Repeat: Repeat the process with new seed points until all points are visited.

8. Noise Removal: Discard clusters that are smaller than a specified minimum size, as they are likely to be noise.

Here is a pseudocode example:

```
import numpy as np
from sklearn.neighbors import KDTree

def euclidean_cluster_extraction(points, distance_threshold, min_cluster_size):
    kdtree = KDTree(points)
    clusters = []
    visited = np.zeros(len(points), dtype=bool)

    for i in range(len(points)):
        if not visited[i]:
            cluster = []
            queue = [i]
            visited[i] = True

            while queue:
                point_idx = queue.pop(0)
                neighbors = kdtree.query_radius([points[point_idx]],\
                                                r=distance_threshold)[0]

                for neighbor_idx in neighbors:
                    if not visited[neighbor_idx]:
                        queue.append(neighbor_idx)
                        visited[neighbor_idx] = True
                        cluster.append(neighbor_idx)

            if len(cluster) >= min_cluster_size:
                clusters.append(cluster)

    return clusters
```

7.8.3 Implementation

Let's begin by creating the cpp file name **6_clustering.cpp** inside **src** of the **slam_1_pc-process** package

```
touch src/slam_1_pc-process/src/6_clustering.cpp
```

Again, we will be using the PCL library, implementing this process is very simple and relieves us from having to code from scratch.
From the code below, the main idea here is, that again, on line 60 we are utilizing KD-tree and we set some parameters on line 57-59. But since we also want to

understand and visualize each cluster, we will iterate each of them on line 65.
Here is our practical implementation using PCL in **6_clustering.cpp**:

```
1  #include <iostream>
2  #include <filesystem>
3
4  #include <pcl/io/pcd_io.h>
5  #include <pcl/point_types.h>
6
7  #include <pcl/sample_consensus/method_types.h>
8  #include <pcl/sample_consensus/model_types.h>
9  #include <pcl/segmentation/sac_segmentation.h>
10 #include <pcl/segmentation/extract_clusters.h>
11 #include <pcl/filters/extract_indices.h>
12
13 #include <pcl/search/search.h>
14 #include <pcl/search/kdtree.h>
15 #include <pcl/features/normal_3d.h>
16 #include <pcl/point_cloud.h>
17
18 typedef pcl::PointXYZ PointT;
19
20 void export_pcd(const std::string& file_name,std::string& path,\
21                 pcl::PointCloud<PointT>::Ptr cloud_arg){
22     pcl::PCDWriter cloud_writer;
23     cloud_writer.write<PointT>(path+std::string(file_name),*cloud_arg);
24     std::cout << "Saved PCD to: " << path+std::string(file_name) << std::endl;
25 }
26
27 int main()
28 {
29     // ******************************         Import PCD
30     pcl::PointCloud<PointT>::Ptr input_cloud (new pcl::PointCloud<PointT>);
31
32     pcl::PCDReader cloud_reader;
33     pcl::PCDWriter cloud_writer;
34
35     std::string import_file_pcd         = "5_world_after.pcd";
36     std::string export_file_pcd_before  = "6_world_before.pcd";
37     std::string export_file_pcd_after   = "6_world_after.pcd";
38     std::string dir_pcd                 = "slam_1_pc-process/data";
39     std::filesystem::path path_ros_ws   = std::filesystem::current_path();
40     std::filesystem::path path_dir_pcd  = path_ros_ws/"src"/dir_pcd;
41     std::string path_import             = path_dir_pcd/import_file_pcd;
42     std::string path_export_before      = path_dir_pcd/export_file_pcd_before;
43     std::string path_export_after       = path_dir_pcd/export_file_pcd_after;
```

```
44
45      cloud_reader.read(path_import,*input_cloud);
46      //
47      std::string path_dir_pcd_string     = path_dir_pcd/"";
48
49      //  ******************************    Clustering Indices Extraction
50      pcl::PointCloud<PointT>::Ptr export_cloud (new pcl::PointCloud<PointT>);
51
52      std::vector<pcl::PointIndices> cluster_indices;
53      pcl::search::KdTree<PointT>::Ptr tree (new pcl::search::KdTree<PointT> ());
54
55      //  Extract indices using EC with K-D tree method
56      pcl::EuclideanClusterExtraction<PointT> euclid_extract_cluster;
57      euclid_extract_cluster.setClusterTolerance(0.25); // 2cm
58      euclid_extract_cluster.setMinClusterSize(150);
59      euclid_extract_cluster.setMaxClusterSize(2000);
60      euclid_extract_cluster.setSearchMethod(tree);
61      //
62      euclid_extract_cluster.setInputCloud(input_cloud);
63      euclid_extract_cluster.extract(cluster_indices);
64
65      //  ******************************    Save each Cluster
66      int loop_counter = 1;
67      for (size_t i = 0; i < cluster_indices.size(); i++)
68      {
69          pcl::PointCloud<PointT>::Ptr each_cluster (new pcl::PointCloud<PointT>);
70          pcl::IndicesPtr indices (new std::vector<int>(
71            cluster_indices[i].indices.begin(), cluster_indices[i].indices.end()));
72
73          //  Extracting from indices
74          pcl::ExtractIndices<PointT> extract;
75          extract.setIndices(indices);
76          extract.setNegative(false);
77          //
78          extract.setInputCloud(input_cloud);
79          extract.filter(*each_cluster);
80
81          // ******************************   Export PCD (each cluster)
82          std::stringstream cloud_name;
83          cloud_name << "6_cluster_" << loop_counter << ".pcd";
84          export_pcd(cloud_name.str(),path_dir_pcd_string,each_cluster);
85          loop_counter++;
86
87          // Add filtered cluster to filtered cluster group (export_cloud)
88          export_cloud->operator+=(*each_cluster);
89      }
```

```
90
91      // ******************************   Export PCD
92
93      cloud_writer.write<PointT>(path_export_before,*input_cloud);
94      cloud_writer.write<PointT>(path_export_after,*export_cloud);
95      std::cout << "Saved PCD to: " << path_export_before << std::endl;
96      std::cout << "Saved PCD to: " << path_export_after << std::endl;
97
98    return 0;
99  }
```

7.8.4 Result

```
1  colcon build && source ~/.bashrc
2  ros2 run slam_1_pc-process 6_clustering.cpp
3  pcl_viewer src/slam_1_pc-process/data/6_cluster*
```

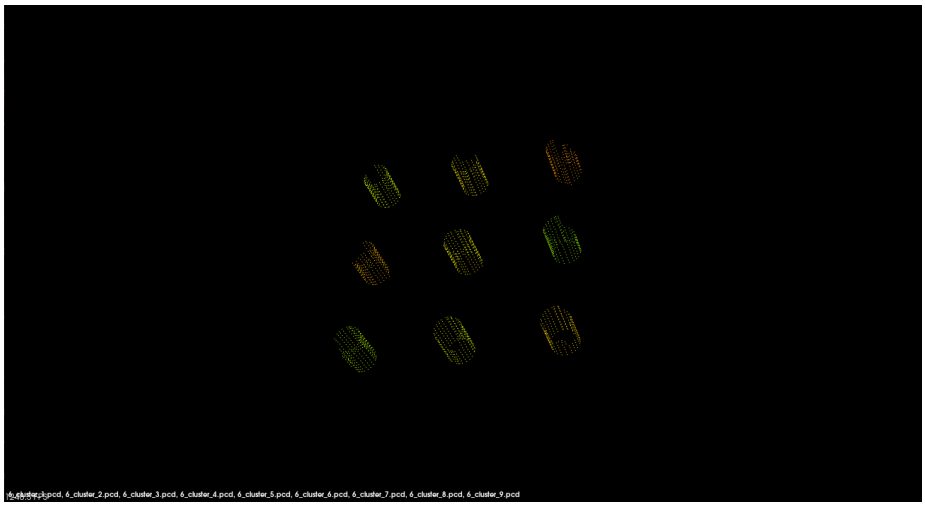

Figure 12: Point cloud of each cluster

We can see that each cluster is given a specific color so we can see a little bit more clearly. As we now can distinguish between each cluster or object, we are ready to use these algorithms on a real LiDAR dataset.

8 Kitti Dataset

The KITTI[2] dataset is a widely used benchmark dataset in the field of computer vision and robotics, specifically for tasks related to autonomous driving and scene understanding.

```
1 https://www.cvlibs.net/datasets/kitti/
```

Kitti takes advantage of Kitti's autonomous driving platform Annieway to develop novel challenging real-world computer vision benchmarks. Kitti's tasks of interest are stereo, optical flow, visual odometry, 3D object detection, and 3D tracking. For this purpose, Kitti equipped a standard station wagon with two high-resolution color and grayscale video cameras. Accurate ground truth is provided by a Velodyne laser scanner and a GPS localization system. Kitti's datasets are captured by driving around the mid-size city of Karlsruhe, in rural areas, and on highways. Up to 15 cars and 30 pedestrians are visible per image. Besides providing all data in raw format, Kitti extracts benchmarks for each task. For each of Kitti's benchmarks, Kitti also provides an evaluation metric, and this evaluation is Kittibsite. Preliminary experiments show that methods ranking high on established benchmarks such as Middlebury perform below average when being moved outside the laboratory to the real world. Kitti's goal is to reduce this bias and complement existing benchmarks by providing real-world benchmarks with novel difficulties to the community.

8.1 Purpose and Contents

The KITTI dataset is a collection of high-resolution images, LiDAR point clouds, and sensor data collected from a moving vehicle in urban and rural environments.
It includes different types of data useful for autonomous driving research, such as:

- Camera Images: RGB images captured by a front-facing camera.
- LiDAR Point Clouds: 3D point cloud data generated by LiDAR sensors, representing the environment in 3D.
- GPS and IMU Data: Ground truth poses (position and orientation) of the vehicle obtained from GPS and Inertial Measurement Units (IMUs).
- Object Annotations: Labels and bounding boxes for vehicles, pedestrians, cyclists, and other relevant objects in the scene.

8.2 Sensor Setup

This page provides additional information about the recording platform and sensor setup we have used to record this dataset. Our recording platform is a

Volkswagen Passat B6, which has been modified with actuators for the pedals (acceleration and brake) and the steering wheel. The data is recorded using an eight-core i7 computer equipped with a RAID system, running Ubuntu Linux and a real-time database.

- We use the following sensors:
- 1 Inertial Navigation System (GPS/IMU): OXTS RT 3003
- 1 Laserscanner: Velodyne HDL-64E
- 2 Grayscale cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3M-C)
- 2 Color cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3C-C)
- 4 Varifocal lenses, 4-8 mm: Edmund Optics NT59-917

The laser scanner spins at 10 frames per second, capturing approximately 100k points per cycle. The vertical resolution of the laser scanner is 64. The cameras are mounted approximately level with the ground plane. The camera images are cropped to a size of 1382 x 512 pixels using libdc's format 7 mode. After rectification, the images get slightly smaller. The cameras are triggered at 10 frames per second by the laser scanner (when facing forward) with shutter time adjusted dynamically (maximum shutter time: 2 ms). Our sensor setup with respect to the vehicle is illustrated in the following figure. Note that more information on calibration parameters is given in the calibration files and the development kit (see raw data section).

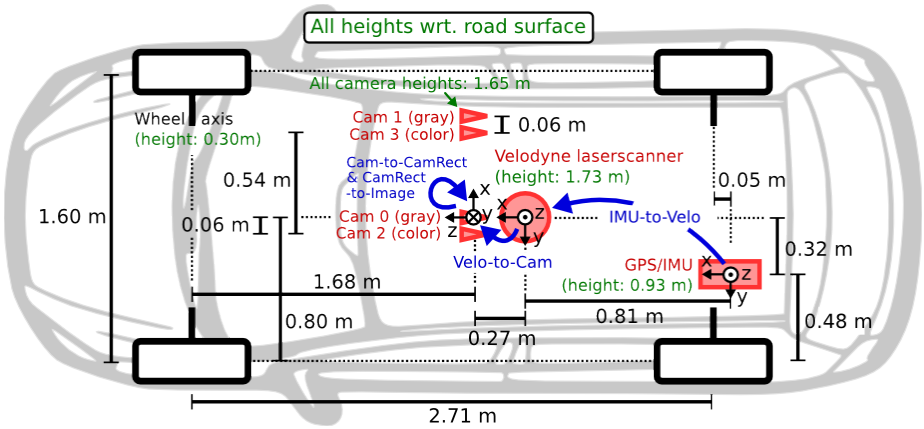

Figure 13: This figure shows Kitti's fully equipped vehicle

Figure 14: This figure shows Kitti's fully equipped vehicle

8.3 Implementing the Dataset to our Project

Before we can use Kitti's dataset we will need to download the dataset. Also, we will not be trying to implement a way to publish the data to our node so we will use an already-made package from umtclskn[6]

```
1 github.com/umtclskn/ros2_kitti_publishers
```

So, let's begin with downloading the Kitti's dataset. In the home directory:

```
1 mkdir ros_slam_ws/data
2 cd ros_slam_ws/data
3 sudo apt-get install unzip
4 curl -OL s3.eu-central-1.amazonaws.com/avg-kitti/raw_data_downloader.zip
5 chmod +x raw_data_downloader.zip
6 unzip raw_data_downloader.zip
```

If this process takes a long time, we can skip it when we have downloaded

```
1 2011_09_26/2011_09_26_drive_0015_sync
```

in our data directory, we have just created.
Next, we will be adding a Kitti dataset publisher to our workspace using git.
In our workspace directory:

```
cd src
sudo apt-get install git && git init
git submodule add https://www.github.com/umtclskn/ros2_kitti_publishers
```

Now, you should be able to see

```
ros2_kitti_publishers
```

in your **src** directory.
Now let's check if everything works correctly.
In our workspace directory:

```
colcon build && source ~/.bashrc
ros2 run ros2_kitti_publishers kitti_publishers
```

you should not see any error and should be seeing this log in the terminal.

```
...
[INFO] [...] [publisher_node]: OxTs size: '30'
[INFO] [...] [publisher_node]: get_path: '0'
...
```

9 Final Implementation

Finally, we will be doing our final implementation.
We will also discuss these topics in order:

- Topic and Message Type
- Implementation from our Methodology
- Visualization
- Publisher and Subscription
- Combining Pre-processing, Visualization, and Publisher & Subscription
- ROS 2 Launch File

9.1 Topic and Message Type

To be able to use Kitti's dataset in our project, we must receive a ROS 2 message from the publisher mentioned in the Kitti Dataset section. We must know which topic and message it produces. To do this is very simple since we know the commands for it from the ROS 2: Topic section. In our workspace directory:

```
colcon build && source ~/.bashrc
ros2 run ros2_kitti_publishers kitti_publishers
```

you should not see any error and should be seeing this log in the terminal.

```
...
[INFO] [...] [publisher_node]: OxTs size: '30'
[INFO] [...] [publisher_node]: get_path: '0'
...
```

Let's also see which topic it is publishing by doing

```
ros2 topic list
```

There are several topics but we will need a point cloud data or **/kitti/point_cloud** topic.
We should also check the message type of the topic by doing

```
ros2 topic info /kitti/point_cloud
```

Now, we know that the message type is: **sensor_msgs/msg/PointCloud2**, so our final implementation will need a subscription for that.

9.2 Implementation from our Methodology

We will, essentially, copy and paste our code from our previously discussed methodology, assuming we will have a subscription to the raw data name **input_cloud**. The code should look similar to this since we're combining all the steps mentioned.
For this dataset, after several tests, we have found different optimal parameters for each of these stages:

1. Voxel Grid: Left Size of 0.1 for X, Y, and Z. Since the distance between each point is a bit bigger than our previous test.

2. Euclidean Cluster Extraction: min cluster size of 600. Since we will need to filter out smaller relevant details from the point cloud.

```
1  pcl::PointCloud<PointT>::Ptr pcl_cloud (new pcl::PointCloud<PointT>);
2  pcl::fromROSMsg(*input_cloud, *pcl_cloud);
3
4  //  ******************************    Passthrough
5  pcl::PointCloud<PointT>::Ptr passthru_cloud (new pcl::PointCloud<PointT>) ;
6  //
7  int radius = 15;
8  pcl::PassThrough<PointT> passing_x;
9  //  Along X Axis
10 passing_x.setInputCloud(pcl_cloud);
11 passing_x.setFilterFieldName("x");
12 passing_x.setFilterLimits(-radius,radius);
13 passing_x.filter(*passthru_cloud);
14
15 //  Along Y Axis
16 pcl::PassThrough<PointT> passing_y;
17 passing_y.setInputCloud(passthru_cloud);
18 passing_y.setFilterFieldName("y");
19 passing_y.setFilterLimits(-radius,radius);
20 passing_y.filter(*passthru_cloud);
21
22 //  ******************************    Voxel Filter
23 pcl::PointCloud<PointT>::Ptr voxel_cloud (new pcl::PointCloud<PointT>) ;
24 //
25 pcl::VoxelGrid<PointT> voxel_filter;
26 voxel_filter.setInputCloud(passthru_cloud);
27 voxel_filter.setLeafSize(0.1 , 0.1, 0.1);
28 voxel_filter.filter(*voxel_cloud);
29
30 //  ******************************    Plane Segmentation
31 pcl::PointCloud<PointT>::Ptr plane_cloud(new pcl::PointCloud<PointT>);
32 //
```

```
33 pcl::PointCloud<pcl::Normal>::Ptr\
34     plane_normals(new pcl::PointCloud<pcl::Normal>);
35 pcl::PointIndices::Ptr plane_inliers(new pcl::PointIndices);
36 pcl::ModelCoefficients::Ptr plane_coefficients(new pcl::ModelCoefficients);
37
38 //  Extracting Normals
39 pcl::NormalEstimation<PointT, pcl::Normal> normal_extractor;
40 pcl::search::KdTree<PointT>::Ptr tree(new pcl::search::KdTree<PointT>());
41 normal_extractor.setSearchMethod(tree);
42 normal_extractor.setInputCloud(voxel_cloud);
43 normal_extractor.setKSearch(30);
44 normal_extractor.compute(*plane_normals);
45
46 //  Parameters for Plane Segmentation
47 pcl::SACSegmentationFromNormals<PointT, pcl::Normal> plane_segmentator_norms;
48 plane_segmentator_norms.setOptimizeCoefficients(true);
49 plane_segmentator_norms.setModelType(pcl::SACMODEL_NORMAL_PLANE);
50 plane_segmentator_norms.setMethodType(pcl::SAC_RANSAC);
51 plane_segmentator_norms.setNormalDistanceWeight(0.5);
52 plane_segmentator_norms.setMaxIterations(100);
53 plane_segmentator_norms.setDistanceThreshold(0.4);
54 plane_segmentator_norms.setInputCloud(voxel_cloud);
55 plane_segmentator_norms.setInputNormals(plane_normals);
56 plane_segmentator_norms.segment(*plane_inliers,*plane_coefficients);
57
58 //  Extracting from indices
59 pcl::ExtractIndices<PointT> plane_extract_indices;
60 plane_extract_indices.setInputCloud(voxel_cloud);
61 plane_extract_indices.setIndices(plane_inliers);
62 plane_extract_indices.setNegative(true);
63 plane_extract_indices.filter(*plane_cloud);
64
65 //  ******************************    Object Clustering
66 std::vector<pcl::PointIndices> cluster_indices;
67 tree->setInputCloud(plane_cloud);
68
69 pcl::EuclideanClusterExtraction<PointT> euclid_extract_cluster;
70 euclid_extract_cluster.setClusterTolerance(0.25); // 2cm
71 euclid_extract_cluster.setMinClusterSize(600);
72 euclid_extract_cluster.setMaxClusterSize(5000);
73 euclid_extract_cluster.setSearchMethod(tree);
74 euclid_extract_cluster.setInputCloud(plane_cloud);
75 euclid_extract_cluster.extract(cluster_indices);
76
77 pcl::PointCloud<PointT>::Ptr all_clusters (new pcl::PointCloud<PointT>);
78
```

```
79  struct BBox
80  {
81    float x_min;
82    float x_max;
83    float y_min;
84    float y_max;
85    float z_min;
86    float z_max;
87    double r = 1.0;
88    double g = 0.0;
89    double b = 0.0;
90  };
91  std::vector<BBox> bbox_list;
92
93  for (size_t i = 0; i < cluster_indices.size(); i++)
94  {
95    pcl::PointCloud<PointT>::Ptr each_cluster (new pcl::PointCloud<PointT>);
96    pcl::IndicesPtr indices(new std::vector<int>(
97      cluster_indices[i].indices.begin(), cluster_indices[i].indices.end()));
98
99    //  Extracting from indices
100   pcl::ExtractIndices<PointT> extract;
101   extract.setInputCloud (plane_cloud);
102   extract.setIndices(indices);
103   extract.setNegative (false);
104   //
105   extract.filter(*each_cluster);
106
107   all_clusters->operator+=(*each_cluster);
108
109   // adding bbox
110   Eigen::Vector4f min_pt, max_pt;
111   pcl::getMinMax3D<PointT>(*each_cluster, min_pt, max_pt);
112   BBox bbox;
113   bbox.x_min = min_pt[0];
114   bbox.y_min = min_pt[1];
115   bbox.z_min = min_pt[2];
116   bbox.x_max = max_pt[0];
117   bbox.y_max = max_pt[1];
118   bbox.z_max = max_pt[2];
119   bbox_list.push_back(bbox);
120 }
```

Most of our code is similar to the one we discussed before until line 75 and 105.
The purpose of this is to publish an **MarkerArray** message to our visualization.
Which are bounding-boxes for containing cuboids of our clusters.

9.3 Virtualization

Next, we will discuss how to turn it into a **MarkerArray** message. We will provide some brief comments for each block of code for their purposes since they're simple in its nature.
Now we will assume that there are 2 publishers for publishing each Bounding-box and the cluster, name **publisher_marker** and **publisher_** respectively.

```
//  ******************************    Visualization
visualization_msgs::msg::MarkerArray marker_array;

int id = 0;
const std_msgs::msg::Header& inp_header = input_cloud->header;
// Create a marker for each bounding box
for (const auto& bbox : bbox_list)
{
  // Create the marker for the top square
  visualization_msgs::msg::Marker top_square_marker;
  top_square_marker.header = inp_header;
  top_square_marker.ns = "bounding_boxes";
  top_square_marker.id = id++;
  top_square_marker.type = visualization_msgs::msg::Marker::LINE_STRIP;
  top_square_marker.action = visualization_msgs::msg::Marker::ADD;
  top_square_marker.pose.orientation.w = 1.0;
  top_square_marker.scale.x = 0.06;
  top_square_marker.color.r = bbox.r;
  top_square_marker.color.g = bbox.g;
  top_square_marker.color.b = bbox.b;
  top_square_marker.color.a = 1.0;

  // Add the points to the top square marker
  geometry_msgs::msg::Point p1, p2, p3, p4;
  p1.x = bbox.x_max; p1.y = bbox.y_max; p1.z = bbox.z_max;
  p2.x = bbox.x_min; p2.y = bbox.y_max; p2.z = bbox.z_max;
  p3.x = bbox.x_min; p3.y = bbox.y_min; p3.z = bbox.z_max;
  p4.x = bbox.x_max; p4.y = bbox.y_min; p4.z = bbox.z_max;
  top_square_marker.points.push_back(p1);
  top_square_marker.points.push_back(p2);
  top_square_marker.points.push_back(p3);
  top_square_marker.points.push_back(p4);
  top_square_marker.points.push_back(p1);

  // Add the top square marker to the array
  marker_array.markers.push_back(top_square_marker);

  // Create the marker for the bottom square
  visualization_msgs::msg::Marker bottom_square_marker;
```

```
40  bottom_square_marker.header = inp_header;
41  bottom_square_marker.ns = "bounding_boxes";
42  bottom_square_marker.id = id++;
43  bottom_square_marker.type = visualization_msgs::msg::Marker::LINE_STRIP;
44  bottom_square_marker.action = visualization_msgs::msg::Marker::ADD;
45  bottom_square_marker.pose.orientation.w = 1.0;
46  bottom_square_marker.scale.x = 0.04;
47  bottom_square_marker.color.r = bbox.r;
48  bottom_square_marker.color.g = bbox.g;
49  bottom_square_marker.color.b = bbox.b;
50  bottom_square_marker.color.a = 1.0;
51
52  // Add the points to the bottom square marker
53  geometry_msgs::msg::Point p5, p6, p7, p8;
54  p5.x = bbox.x_max; p5.y = bbox.y_max; p5.z = bbox.z_min;
55  p6.x = bbox.x_min; p6.y = bbox.y_max; p6.z = bbox.z_min;
56  p7.x = bbox.x_min; p7.y = bbox.y_min; p7.z = bbox.z_min;
57  p8.x = bbox.x_max; p8.y = bbox.y_min; p8.z = bbox.z_min;
58
59  bottom_square_marker.points.push_back(p5);
60  bottom_square_marker.points.push_back(p6);
61  bottom_square_marker.points.push_back(p7);
62  bottom_square_marker.points.push_back(p8);
63  bottom_square_marker.points.push_back(p5);
64  // connect the last point to the first point to close the square
65
66  // Add the bottom square marker to the marker array
67  marker_array.markers.push_back(bottom_square_marker);
68
69  // Create the marker for the lines connecting the top and bottom squares
70  visualization_msgs::msg::Marker connecting_lines_marker;
71  connecting_lines_marker.header = inp_header;
72  connecting_lines_marker.ns = "bounding_boxes";
73  connecting_lines_marker.id = id++;
74  connecting_lines_marker.type = visualization_msgs::msg::Marker::LINE_LIST;
75  connecting_lines_marker.action = visualization_msgs::msg::Marker::ADD;
76  connecting_lines_marker.pose.orientation.w = 1.0;
77  connecting_lines_marker.scale.x = 0.04;
78  connecting_lines_marker.color.r = 0.0;
79  connecting_lines_marker.color.g = 1.0;
80  connecting_lines_marker.color.b = 0.0;
81  connecting_lines_marker.color.a = 1.0;
82
83  // Add the points to the connecting lines marker
84  connecting_lines_marker.points.push_back(p1);
85  connecting_lines_marker.points.push_back(p5);
```

```
86    connecting_lines_marker.points.push_back(p2);
87    connecting_lines_marker.points.push_back(p6);
88    connecting_lines_marker.points.push_back(p3);
89    connecting_lines_marker.points.push_back(p7);
90    connecting_lines_marker.points.push_back(p4);
91    connecting_lines_marker.points.push_back(p8);
92
93    // Add the connecting lines marker to the marker array
94    marker_array.markers.push_back(connecting_lines_marker);
95
96
97    // Create a marker for the corners
98    visualization_msgs::msg::Marker corner_marker;
99    corner_marker.header = inp_header;
100   corner_marker.ns = "bounding_boxes";
101   corner_marker.id = id++;
102   corner_marker.type = visualization_msgs::msg::Marker::SPHERE;
103   corner_marker.action = visualization_msgs::msg::Marker::ADD;
104   corner_marker.pose.orientation.w = 1.0;
105   corner_marker.scale.x = 0.4;
106   corner_marker.scale.y = 0.4;
107   corner_marker.scale.z = 0.4;
108   corner_marker.color.r = bbox.r;
109   corner_marker.color.g = 0.2;
110   corner_marker.color.b = 0.5;
111   corner_marker.color.a = 0.64;
112
113   // Create a sphere for each corner and add it to the marker array
114   corner_marker.pose.position = p1;
115   corner_marker.id = id++;
116   marker_array.markers.push_back(corner_marker);
117   corner_marker.pose.position = p2;
118   corner_marker.id = id++;
119   marker_array.markers.push_back(corner_marker);
120   corner_marker.pose.position = p3;
121   corner_marker.id = id++;
122   marker_array.markers.push_back(corner_marker);
123   corner_marker.pose.position = p4;
124   corner_marker.id = id++;
125   marker_array.markers.push_back(corner_marker);
126   corner_marker.pose.position = p5;
127   corner_marker.id = id++;
128   marker_array.markers.push_back(corner_marker);
129   corner_marker.pose.position = p6;
130   corner_marker.id = id++;
131   marker_array.markers.push_back(corner_marker);
```

```
132     corner_marker.pose.position = p7;
133     corner_marker.id = id++;
134     marker_array.markers.push_back(corner_marker);
135     corner_marker.pose.position = p8;
136     corner_marker.id = id++;
137     marker_array.markers.push_back(corner_marker);
138 
139     // Publish each bbox
140     publisher_marker->publish(marker_array);
141 }
142 // Convert to ros2 message
143 sensor_msgs::msg::PointCloud2 ros2_cluster_cloud;
144 pcl::toROSMsg(*all_clusters, ros2_cluster_cloud);
145 ros2_cluster_cloud.header = input_cloud->header;
146 
147 publisher_->publish(ros2_cluster_cloud);
```

9.4 Publisher and Subscription

Remember that we need to publish 2 messages, **MarkerArray** and **PointCloud2**, and a subscription for Kitti **PointCloud2** data.
We know that the Kitti dataset publisher topic name is **/kitti/point_cloud** from our Initial installation section.
We will name our **MarkerArray** and **PointCloud2** topic **visualization_marker_array** and **cluster_cloud** respectively.

```
public:
  Pre_process()
  : Node("pre_processing")
  {
    subscription_ =
    this->create_subscription<sensor_msgs::msg::PointCloud2>(
      "/kitti/point_cloud", 10, std::bind(&VoxelGrid_filter::timer_callback,\
        this, std::placeholders::_1));

    publisher_ =
    this->create_publisher<sensor_msgs::msg::PointCloud2>(
      "cluster_cloud", 10);

    publisher_marker =
    this->create_publisher<visualization_msgs::msg::MarkerArray>(
      "visualization_marker_array", 10);
  }
```

9.5 Combining Pre-processing, Visualization, and Publisher & Subscription

Now, we will not go into detail on how to implement each section of our code together since the concept should be simple if you follow the Methodology section but generally, the final code should look like this.

```
#include <chrono>
#include <memory>
#include <string>

#include "std_msgs/msg/string.hpp"
#include "visualization_msgs/msg/marker_array.hpp"
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/point_cloud2.hpp"
#include "pcl_conversions/pcl_conversions.h"
#include "pcl/filters/voxel_grid.h"
#include <pcl/filters/extract_indices.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/segmentation/extract_clusters.h>
#include <pcl/features/normal_3d.h>
#include <pcl/segmentation/sac_segmentation.h>
#include "visualization_msgs/msg/marker_array.hpp"
#include <pcl/filters/passthrough.h>
#include <pcl/octree/octree_pointcloud.h>
#include <pcl/octree/octree.h>

#include <pcl/point_cloud.h>
using namespace std::chrono_literals;
typedef pcl::PointXYZ PointT;

class Pre_processing : public rclcpp::Node
{
  public:

    ...
    ...
    ...

  private:
    void timer_callback(const sensor_msgs::msg::PointCloud2::SharedPtr\
                        input_cloud)
    {
      pcl::PointCloud<PointT>::Ptr pcl_cloud (new pcl::PointCloud<PointT>);
      pcl::fromROSMsg(*input_cloud, *pcl_cloud);

```

```
40        ...
41        ...
42        ...
43
44
45        for (size_t i = 0; i < cluster_indices.size(); i++)
46        {
47
48          ...
49          ...
50          ...
51
52          // adding bbox
53          Eigen::Vector4f min_pt, max_pt;
54          pcl::getMinMax3D<PointT>(*each_cluster, min_pt, max_pt);
55          BBox bbox;
56          bbox.x_min = min_pt[0];
57          bbox.y_min = min_pt[1];
58          bbox.z_min = min_pt[2];
59          bbox.x_max = max_pt[0];
60          bbox.y_max = max_pt[1];
61          bbox.z_max = max_pt[2];
62          bbox_list.push_back(bbox);
63        }
64
65        //  ******************************    Visualization
66        visualization_msgs::msg::MarkerArray marker_array;
67
68        ...
69        ...
70        ...
71
72        publisher_->publish(ros2_cluster_cloud);
73      }
74    rclcpp::Publisher<visualization_msgs::msg::MarkerArray>::SharedPtr publisher_marker;
75    rclcpp::Publisher<sensor_msgs::msg::PointCloud2>::SharedPtr publisher_;
76    rclcpp::Subscription<sensor_msgs::msg::PointCloud2>::SharedPtr subscription_;
77  };
78
79  int main(int argc, char * argv[])
80  {
81    rclcpp::init(argc, argv);
82    rclcpp::spin(std::make_shared<Pre_processing>());
83    rclcpp::shutdown();
84    return 0;
85  }
```

My implementation on Github is located at:

```
curl -OL https://raw.githubusercontent.com/Metastasiz/lidar_slam_preprocessing\
    /main/src/slam_1_pc-process/src/main.cpp
```

And please do not forget about the CMakeList.txt and if you have not downloaded my Rviz config file, you can follow an instruction in the ROS 2: launch file section.
Please be sure to replace the package name with this package name.

9.6 Launch the final implementation

Finally, we will be able to use the Kitti dataset, our Pre-processing algorithms, and using Rviz to visualize it.
If you have not made your own launch file yet, we will need 3 terminals for this process.

1. Kitti dataset publisher node
2. Pre-processing node
3. Rviz Visualization node

Assuming you name the final implementation as main.cpp.

```
ros2 launch slam_1_pc-process my_rviz.launch.py
ros2 run slam_1_pc-process main
ros2 run ros2_kitti_publishers kitti_publishers
```

Alternatively, if you're launching it from our source code or

```
https://github.com/Metastasiz/lidar_slam_preprocessing
```

Please use this command instead

```
ros2 launch slam_0_bringup my_rviz.launch.py
ros2 launch slam_0_bringup publish_kitti.launch.py
```

Now let's use ROS 2 built-in command **rqt_graph** to see the communication between those nodes.

```
rqt_graph
```

We should be able to see this

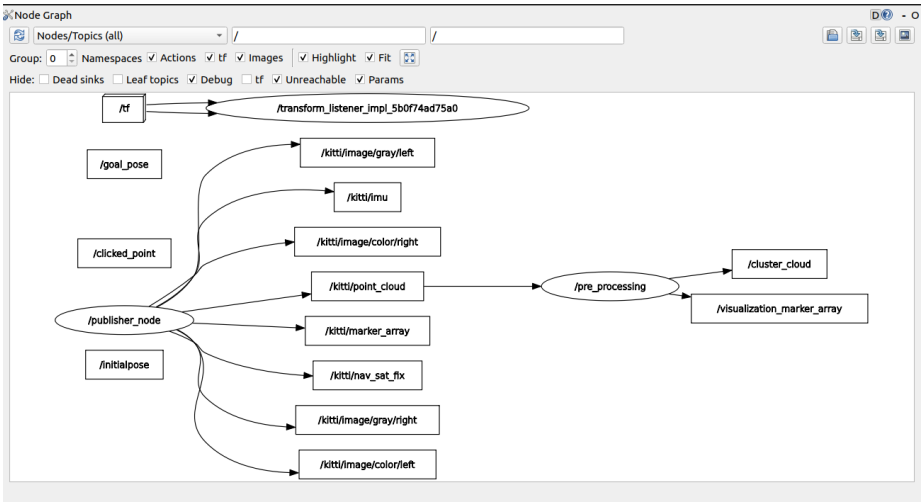

Figure 15: rqt_graph of the final implementation

Now if we only look at the node **publisher_node** we can see that it publishes **kitti/point_cloud** message to the **pre_processing** node which publishes **cluster_cloud** and **visualization_marker_array**

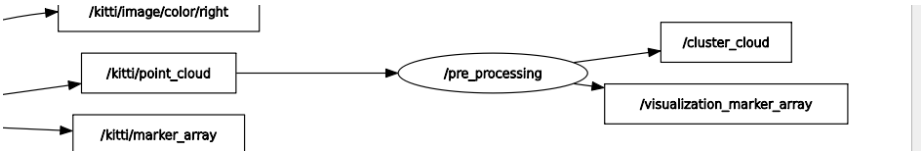

Figure 16: A focused rqt_graph on our Pre_processing node

This is as expected which means our implementation works correctly.

10 Result and Conclusion

In this section, we will be discussing the result of the final implementation and the conclusion of this project.

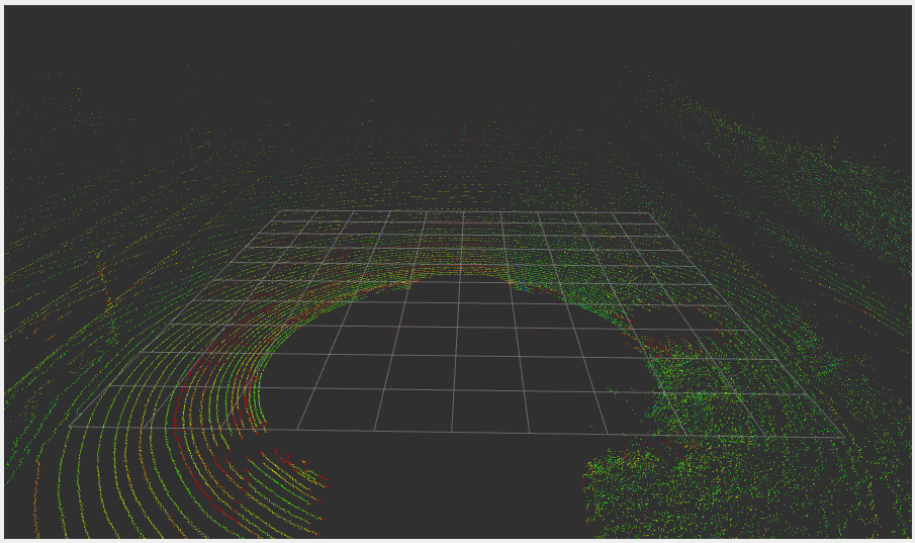

Figure 17: Visualization of raw LiDAR data

This is the visualization of the raw LiDAR data, as you can see, there is a lot of data in the point cloud. Our goal is, hopefully, to reduce it.

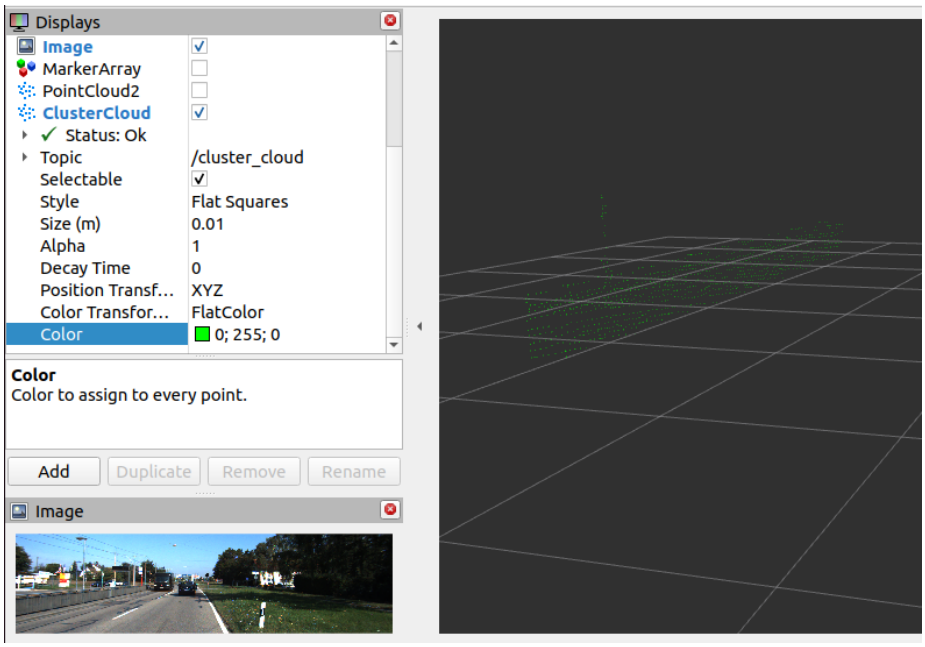

Figure 18: Visualization of only cluster

As we use our discussed methodology to cluster the point cloud, this can be seen when the black car appears from the left side lane (on the bottom left of Figure 16) we can see that our visualization has detected and received information about a cluster of point clouds.

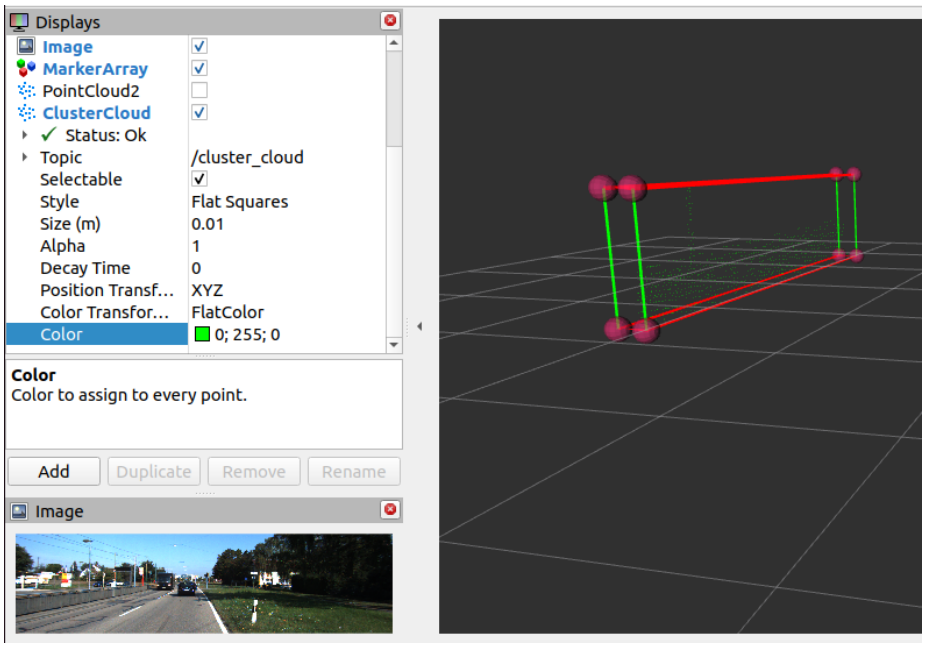

Figure 19: Visualization of the LiDAR SLAM pre-processing with bounding box

Now, we can see that our LiDAR SLAM pre-processing step has prepared the raw dataset for the next stage of SLAM, the frontend stage, which most likely needs to have the feature matching step.

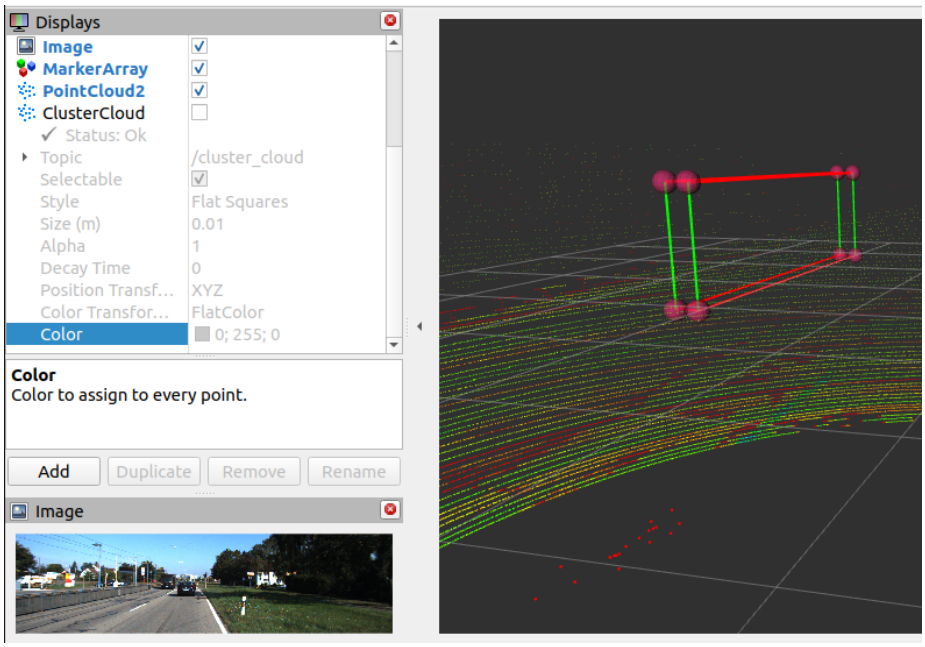

Figure 20: Visualization of raw LiDAR data with bounding box

Removing unnecessary point clouds and leaving it with only distinguishable features, e.g. large objects and large walls, has made the feature matching step both faster and possible to render in real-time. It also reduces the data size which, you can possibly guess, is significantly smaller.

Conclusion

In this work, it has been shown that the application of ROS 2 with PCL makes the implementation of such tedious SLAM technology seem like an easy task.

Without ROS 2, we would have to reinvent the wheel on how to communicate with the LiDAR dataset and the visualization. And without PCL, we would have to write the whole algorithm from scratch. This would not be viable in a state where researchers have to deploy multiple types of applications, some are relevant to the research project.

However, as much as I believe that a quick implementation is necessary, I also believe that it is vital for researchers to understand the mechanism of the algorithms to have a full grasp of what breakthrough could be hiding.

My motivation for this project has always been trying to implement SLAM from scratch, to understand how and why each step and algorithm is used, and I believe what I have accomplished today will be a building block for what I will research and refine in the future.

References

[1] Xiang Gao, Tao Zhang, Yi Liu, and Qinrui Yan. *14 Lectures on Visual SLAM: From Theory to Practice*. Publishing House of Electronics Industry, 2017.

[2] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[3] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[4] roboticsbackend.com. *roboticsbackend*. Edouard Renard, 2016.

[5] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[6] Umut Çalışkan. Ros2 kitti dataset publishers. github.com/umtclskn/ros2_kitti_publishers.