MASTER

Performance evaluation and improvement of LiDAR-based obstacle detection algorithm

Sun, Ran

Award date:
2021

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Performance evaluation and improvement of LiDAR-based obstacle detection algorithm

Master Thesis

Ran Sun (1355937)

Supervisors:
Assistant Professor Dr. Dip Goswami
Ph.D. Candidate Saeid Dehnavi

1.0 version

Eindhoven, July 2021

Abstract

Autonomous mobile robots have relied on the real-time operation to achieve safe, deterministic behavior and smooth trajectory control. The environment in which mobile robots operate may be irregular and changing with time. One of the key features required to provide autonomy to mobile robots is the perception of geometry and the detection of obstacles. Thus, it is crucial to provide fast, accurate, and efficient obstacle detection for autonomous mobile robots.

In this work, we first discuss the obstacle detection algorithm based on 2D LiDAR in a ROS-based system. The study of obstacle detection algorithms using 2D LiDAR and the implementation of the selected algorithm are introduced. Obstacles existing in the robot workspace are represented by point sets against their outlines, and their information is initialized and updated via the raw laser measurement points. The algorithm follows three main steps: pre-processing, splitting and clustering, and classification of consequent measurements. Then, we propose a pipeline implementation for the obstacle detection algorithm, which allows splitting complex processing tasks into separate processes and aims to reduce run time for 2D laser point cloud input within the ROS framework. Experiments performed in the simulation indicate that the data overhead for message transmission is a bottleneck of the performance of the proposed method with the node configuration and separate processes and influences timing negatively. The total transmission overhead takes up to around 50% of the end-to-end response time in the pipelined implementation. The end-to-end latency increases 84% comparing to the execution time of the sequential implementation.

Preface

This thesis marks the end of my graduate study. I am deeply grateful to everyone who has helped and supported me during my master education.

I would first thank my supervisor, Dr. Goswami, for providing guidance and feedback through each stage of the project. Without his patient instructions and constructive suggestions, this project would not have been completed.

I would also like to thank my tutor Saeid Dehnavi, who helped me solve the problems and gave me encouragement when facing difficulties in this project.

Finally, I would express my gratitude to my family and friends. Their support and company helped me go through all the toughness during my studies, especially during the pandemic.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Autonomous mobile robots have become more significant recently due to their relevance and applications in the world. They are used in different scenarios such as hospitals, institutions, agriculture, companies, and homes to improve services and perform daily tasks [4]. The general indoor and outdoor applications of autonomous mobile robots are shown in Figure 1.1.

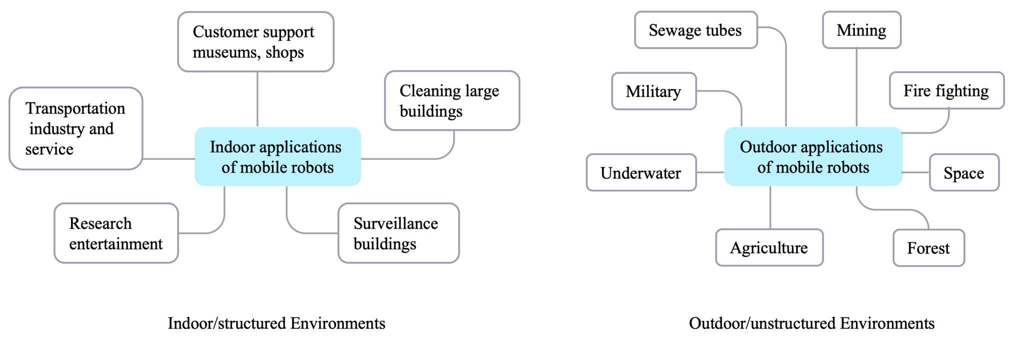
Customer support museums, shops
Transportation industry and service
Cleaning large buildings
Indoor applications of mobile robots
Research entertainment
Surveillance buildings
Indoor/structured Environments
Sewage tubes
Mining
Military
Fire fighting
Underwater
Outdoor applications of mobile robots
Space
Agriculture
Forest
Outdoor/unstructured Environments

Figure 1.1: The general indoor and outdoor applications of autonomous mobile robots. [28]

As the new concept of Industry 4.0 has emerged and is representing the current trend of automation and data exchange in manufacturing technology by creating a ”smart factory” [51], it has brought new challenges to autonomous mobile robots, that is, they have real-time and safety requirements. Industry 4.0 is characterized by increasing reliance on automation and system interconnection, thus mobile robot navigation turns to be an important tool. Autonomous mobile robots must have the ability to react correctly and safely to eventualities when moving around, which means that they must avoid obstacles placed within the confinement of movement without human intervention. Because mobile robots undertake the tasks of transportation and coordination in a dynamic environment, one of the important issues that need to be solved is the real-time identification of obstacles. Furthermore, these robots are consequently equipped with sensors that can respond quickly and processing units with the ability to interpret all the data in real-time. In this case, they often have constraints in memory, processing speed, and other aspects.

The sensors should be effective and real-time, some of the commonly used sensors include vision sensors, ultrasonic sensors, and LiDAR sensors. The vision sensors excel at semantic understanding, but require high processing and material costs and are sensitive to illumination. Ultrasonic sensors are less expensive and require less power, but their accuracy and resolution are inferior to those of the others. Compared to other types of sensors, LiDAR sensors have the advantages in distance measurement, such as high scanning precision, large detection range, large detection range, and insensitivity to lighting conditions [8]. Thus, they are widely used in obstacle detection systems. In this thesis, we consider the 2D point-cloud measurement from a 2D LiDAR, which makes sensing the presence of surrounding material obstacles reliable and simple.

The objective of the thesis is to implement an obstacle detection algorithm and parallel the algorithm to achieve time efficiency.

1.2 Motivation

Robot mapping and navigation are the basis of many robotic applications, such as service robots, warehouse management, patrolling and autonomous driving [42]. The recent focus is on developing applications in a variety of cyber-physical systems (CPS). CPSs are developed to integrate physical processes and virtual computing processes [12]. The robot's ability to intelligently interact with the world supporting embedded computing and communication, real-time control, and perception of the surrounding environment.

With the development of robot navigation and sensor compactness, robots can map the environment accurately and complete complex autonomous navigation tasks [42]. The task of the navigation system is to guide the robot to the target point without a collision with obstacles. In the presence of obstacles, one of the significant issues in robotics is to design a fast and efficient procedure for obstacle detection. The obstacle detection algorithm should have the ability to determine the underlying model of the obstacles.

Obstacle detection plays a crucial role in perceiving the environment. When mobile robots run in an unstructured environment, their execution is often uncertain, because the prior knowledge of the environment may not be known and the environment may not be static [43]. Thus, autonomous mobile robots should perceive the environment and detect obstacles accurately in real-time when they are moving. It is also considered a bottleneck due to complex logic control architecture and dynamic environments that must be processed. Thus, fast, accurate, and efficient obstacle detection is a must for autonomous mobile robots.

1.3 Thesis structure

This thesis is divided into eight chapters. The rest of the chapters are organized as follows. Chapter 2 describes the related works on LiDAR-based obstacle detection approaches and pipelined execution in ROS. Chapter 3 discusses the problems to be solved in this thesis and lists the contribution of the project. Chapter 4 presents the simulation setup and case studies. Chapter 5 talks about the selection of obstacle detection algorithm. Chapter 6 explains the implementation of the pipeline version of the selected algorithm. Chapter 7 evaluates the results and gives an analysis. Finally, a brief conclusion and future work suggestions are given in Chapter 8.

Chapter 2

Related Work

In this chapter, we introduce some related work about the thesis. Chapter 2.1 discusses 2D LiDAR-based obstacle detection methods, which include two main categories to describe the environment: grid-based and vector-based methods. Next, we introduce the pipeline in ROS and how ROS can be a great help for the pipeline.

2.1 LiDAR-based obstacle detection

One of the most important features of autonomous mobile robots is their perception of environmental geometry. The perception system of robots often interprets the surroundings using visual cameras and/or laser scanners. The visual sensors have advantages, particularly in scene understanding, but one of their major drawbacks is their sensitivity to light changes, and the detection accuracy is greatly decreased in complex shadows or bad weather conditions. We consider the 2D point cloud measurement from a 2D LiDAR, which makes sensing the presence of surrounding material obstacles reliable and simple due to 2D LiDAR's advantages of high precision, large detection range with a wide FOV, fast scanning frequency, and ability in inclement weather conditions. However, in many cases, utilizing raw data from LiDAR devices is not effective for obstacle detection. Extracting more concise information from obtained data is critical between sensing and actuation [39].

Detection methods based on 2D LiDAR often incorporate region segmentation or clustering algorithms to divide distance measurements into correlated blocks from which different practical properties can be derived [49]. Then, the shapes of obstacles can be estimated and the stored data amount can be decreased. The segmentation process often compares two subsequent laser scan points from the LiDAR measurements, where the distance threshold is determined as constants or adapted to the scenarios. In [35], the authors divided the measurements into segments and constructed polygonal obstacles using line fitting and corner extraction. The extracted lines and corners were matched with the global lines and corners to obtain the pose of the robot. A detailed comparison and explanation of some segment extraction methods from 2D Lidar laser points are shown in [32]. Premebida [38] discussed some algorithms for segmenting 2D laser scans, as well as different approaches for feature recognition and extraction of three geometric primitives, including lines, circles, and ellipses. Clustering laser point clouds is to extract ordered obstacle information and from cluttered point clouds and then intuitively obtain obstacle information [54]. In [17], Fernández et al. used a convolution operation to slit the laser data into clusters, then detected lines in each previously detected cluster. The clustering process helps the robot fully recognize obstacles and perceive the environment.

In terms of representing the surrounding spatial environment, the obstacle detection process using LiDAR sensors can be mainly classified into two representation approaches: grid-based and vector-based methods.

2.1.1 Grid-based obstacle detection method

To depict the environment, many techniques use a 2D occupancy grid (grid-based). Occupancy grid maps are divided into evenly spaced grid cells and represent complex geometries for detection and tracking obstacles if the grid resolution is sufficiently high. The laser points lying into a grid cell indicate that the cell is occupied by the obstacle. A sample of occupancy grid representing the environment cited from [30] is shown in Figure 2.1, where black cells are marked as obstacles, gray cells describe the unknown area in the environment, and white cells are empty space. Each cell models a square area of the environment and stores a value representing whether the area is occupied. The cells can be labeled with either three occupancy state values of "occupied", "unknown" and "free" or occupancy probability [5]. Grid-based methods have no a priori knowledge of objects that can be present in the scene, meaning that the types of elements it can detect has no restriction [49]. Vu et al. in [50] updated the surrounding occupancy grid maps incrementally and dynamic objects were detected with no need of a priori knowledge of the targets. In [31], Mori et al. used grid trajectories acquired by ordering laser points on the grid map to detect dynamic obstacles, which helped in segmentation and depicting the speed and size of dynamic objects. Chen et al. proposed a grid-based approach to identify real-time moving obstacles through comparing previous temporal sequential grid maps to achieve the state of the same grid cells [9]. The local map stored in the occupancy grid map was considered as a pixel map, which was merged into the CNN to determine the road direction in [22].

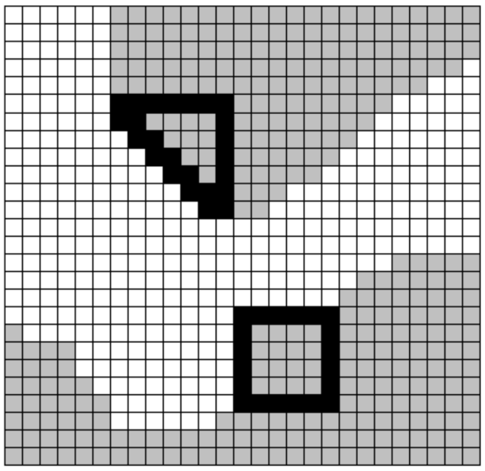

Figure 2.1: An example of occupancy grid representation of the environment [30]

Grid-based methods can describe environments where features are difficult to identify and detect, making them particularly suitable for noise sensors in outdoor environments. The methods also have a mechanism for integrating different types of sensors in the same framework while considering the uncertainty of sensors' data input [50]. Since the densely packed laser points can be mapped to the same grid cell in grid maps, the point clouds will be reduced. If an appropriate resolution is chosen, the data can be searched easily while also keeping adequate information of detected obstacles [52]. However, higher grid resolution demands more memory and extensive computational power for the divided grids. Additionally, the boundaries of the environment require to be established. For example, in a setting with many open spaces, such as a park, it is difficult to acquire enough free space. The grid-based methods compromise accuracy to simplify and reduce bounded data processing [30].

2.1.2 Vector-based obstacle detection method

Another widely adopted method to represent the environment is the vector-based method, which models obstacles by predefined geometric features such as lines, circles, boxes, ellipses, and rectangles. Therefore, obstacles described by geometric features can be expressed by the pose of the feature, including its position and orientation [15]. This representation enables the use of analytical geometry to calculate the distance and similarity between objects [14]. Mertz et al. proposed a system aiming to deal with non-rigid objects such as pedestrians or foliage. They figured out the stable features, reasoned about occlusion geometry, and then identified objects [27]. Petrovskaya and Thrun [36] modeled the geometric features of the tracked vehicles and predicted their forms, resulting in more stable vehicle reference points over time. MacLachlan and Mertz [26] trailed fitting a rectangular model to the laser points and used the rectangle's corners as features according to the characteristics of the laser scanner. In [53], the authors introduced a feature extraction approach based on prediction to detect lines and circles from the LiDAR data without prior information of the surrounding environment. In [6], they detected the legs of people in an indoor environment by extracting fourteen features from the laser scan and using AdaBoost to develop a classifier model of legs.

The obstacle detection methods based on geometric features require each obstacle to have at least one major feature or defined shape. These methods have prior knowledge about the elements to be detected and search for them at each frame. However, it brings the downside of the methods that the real shape of the obstacles is lost. As a result, knowing the similarity of two obstacles provides little information [14]. Compared with the grid-based methods, the vector-based approach with the compact representation of the surrounding environment is suitable for sparse scenarios because of the low memory usage [15].

2.2 Pipelined execution in ROS

Using a processing pipeline within Robot Operating System (ROS) framework[1] aims to generate a measurement, position, or messages[2] to the reuse of the components. Coşar and Bellotto in [11] focused on a software pipeline for human re-identification with a thermal camera. The complete pipeline was implemented as a ROS node[3] and the node subscribed to the raw and color thermal images and published a ROS topic[4] of the name of the recognized person. In [23], Josyula et al. proposed a point cloud segmentation pipeline, which first divided point clouds into slices based on the radical measurements from the LiDAR. Each algorithm step turned to be a parallel stage in the pipeline, passing slices from one stage to the next. The goal of the pipeline was to optimize the execution time of the conventional algorithm and maximize its throughput. In their method, the stages were implemented using the ROS nodelets[5]. It allows zero-copy transport between publisher and subscriber nodelets belonging to the same node. The run time of the pipeline was defined by the timing of each stage. For this reason, the pipeline was optimized by making all the stages execute as close as possible. In [25], the authors showed a real-time perception pipeline, which began with people detection modules based on ROS using RGB-D and 2D LiDAR data and progressed with nodes for aggregating detection from different sensors. The individual components in all stages of the pipeline can be reused for custom setups, where the communication between the stages of the pipeline used ROS messages. The node-based modular structure in ROS allows for the easy separation of complex processing tasks into independent processes that can run on multiple CPU cores or even other computers without considerable extensive reconfiguration. They also integrated a GPU-accelerated detector into the framework to speedup the processing of RGB images.

[1] `http://wiki.ros.org/ROS/Introduction`
[2] `http://wiki.ros.org/Messages`
[3] `http://wiki.ros.org/Nodes`
[4] `http://wiki.ros.org/Topics`
[5] `http://wiki.ros.org/nodelet`

Chapter 3

Problem Statement

In this chapter, we introduce the research questions in this thesis and summarize the contributions of the thesis.

3.1 Problem Statement

Autonomous mobile robots have relied on the real-time operation to achieve safe, deterministic behavior and smooth trajectory control. Since autonomous robots are designed to operate in dynamic and may face unstructured environments, they need to adjust their behavior to new operating conditions and consider multiple goals to settle effective action plans. Their capabilities of adaptability and autonomous operation are achieved through more intelligent perception, movement, and reasoning ability. It is crucial for autonomous mobile robots to perceive the environment and detect obstacles accurately in real-time when they are moving, which is also considered a bottleneck due to complex logic control architecture and dynamic environments that must be processed [7]. Additionally, a large number of correlated vectors put forward high requirements for the real-time operation of obstacle detection. Thus, fast, accurate, and efficient obstacle detection is a premise for autonomous mobile robots.

Therefore, the research questions for this thesis are listed below:

- What obstacle detection algorithm can be implemented on multi-core platforms?
- How to deploy a selected obstacle detection algorithm in a ROS-based system?
- How to deploy pipelined execution of the obstacle detection algorithm in a ROS-based system?

3.2 Contributions

The contribution of this thesis include the following:

1. Study obstacle detection algorithms and justify their feasibility.
2. Select the appropriate obstacle detection algorithm and create a simulation environment in Gazebo to evaluate the performance.
3. Propose a pipeline of the obstacle detection algorithm and implement it in a ROS-based system.
4. Analyze the performance of the pipeline version of the obstacle detection algorithm.

Chapter 4

Simulation setup and case studies

4.1 Environment configuration

The table below lists the system configuration used in the development. The environment is deployed on the Ubuntu 18.04 operation system. An Intel i7-7820HQ processor with x86 64 architecture is utilized. The default Gazebo version is used for the ROS Melodic distribution, it should be noted that using a gazebo version other than the official version delivered from the ROS repositories may result in conflicts or other issues with ROS packages.

Parameters	Type and version
Virtual Machine	Virtualbox 6.1
Host OS Platform	macOS Catalina 10.15.7 version
Guest OS Platform	Ubuntu 18.04.5 LTS
CPU	Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz
processor numbers	4
VM memory size	8429MB
ROS version	ROS Melodic Morenia

Table 4.1: System configuration

4.2 Simulation setup

In this section, we introduce some background information about ROS communication mechanisms, robot simulation in the Gazebo simulator, and the LiDAR configuration.

4.2.1 ROS

Robot Operating System (ROS) is an open-source middleware, which has been used for various robotics applications. The official ROS package provides sufficient functionalities for common robotic tasks, as well as APIs to build packages or interact with external systems [18]. Functionally, ROS presents standard operating system services and high-level functionalities such as asynchronous and synchronous calls, centralized database, and robot configuration systems [33].

The essential components of ROS implementation contain nodes, messages, topics, and services [16, 41]. One of the main goals of ROS is to promote communication between ROS nodes. Nodes are processes performing computations or tasks, which represent executable codes. Each node can accomplish specific tasks of a system. For example, one node can obtain the sensing data from a LiDAR and send it to another node after processing, the second node can use the received data to perform some detection or navigation algorithms and send some commands to a third node to give feedback to control the robot. A node communicates with other nodes through topics, which define the transmitted messages. Messages are data formats that support a structured combination of primitive types, constants, or arrays.

There are mainly two types of message communication between nodes, including the topic-based asynchronous publisher/subscriber mode and service-based synchronous server/client pattern. In the publisher/subscriber mode, the publisher nodes send messages into the topics directly to any subscriber nodes that want to receive the data. The topic is one-way and pertains to the sensor data that requires continuous message transmission. Figure 4.1 illustrates an example of the pub/sub communication in ROS. In the server/client pattern, a node advertises a service on request. Any service client can connect to it and wait for the response after sending the request call. Unlike topics in the publisher/subscriber mode, the service is a one-time message communication. Thus, the two connected nodes will disconnect once the request and response of the service are complete [13]. Figure 4.2 shows an example of service-based communication in ROS. A ROS master node manages the naming and registration for the nodes in the system, tracks the topics and services for both communication mechanisms, and notifies the nodes of the information.

Since the continuous sensor data is kept advertising at a certain rate, other nodes continue to perform subsequent operations after receiving the data, the asynchronous publisher/subscriber pattern is adopted in this project, while the service-based approach is more suitable for logic processing.

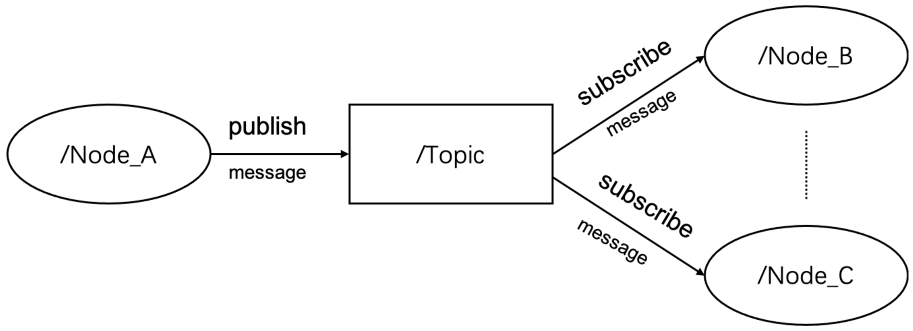

Figure 4.1: The publisher/subscriber ROS communication

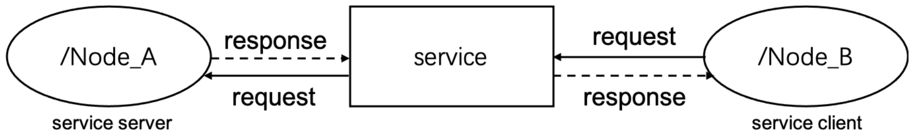

Figure 4.2: The server/client ROS communication

4.2.2 Gazebo

Gazebo is a 3D open-source dynamics simulator that can accurately simulate a population of robots, sensors, and objects in indoor and outdoor environments. Similar to game engines, Gazebo provides higher-fidelity physical simulation, a set of sensors, and interfaces for users and programs. It not only provides realistic sensor feedback but also generates physical interactions between objects, including accurate simulation of rigid body physics [40]. Even though Gazebo was designed to close the gap in realistic robot simulation in outside environments, it is largely utilized by academics and developers for indoor simulation. It attempts to create a real-world for robots relying heavily on physics-based features [44]. The Gazebo simulator has many crucial advantages, such as a robust physics engine, open-source code, and user-friendly graphical interfaces. Users can import existed simulated robots or construct new models using geometrical primitives. The actual robots can be substituted by the simulation model with the robot movement calculation through odometer and sensor data [47].

Figure 4.3 depicts a general architecture of Gazebo components. In Gazebo, a *world* contains a collection of robots and objects in the simulated environment, including the robot model, the created environment, and global parameters such as ambient, lighting, and physics properties. In the middle hierarchy, the model consists of a combination of joints, sensors, and bodies. Libraries interact with Gazebo at the bottom level, which keeps the model from being influenced by the changes of specific tools. Lastly, models use the shared memory interface to receive client commands and return data [24].

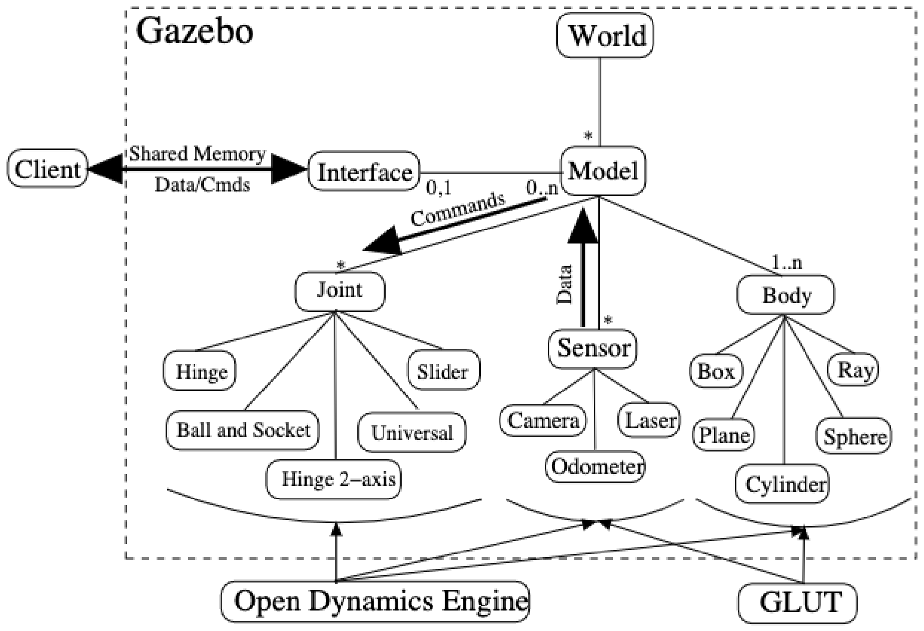

Figure 4.3: The structure of Gazebo components [24]

When designing a mobile robot, we aim to create a model that has the required capabilities (such as accuracy and speed), as well as the levels of reliability and maneuverability to stabilize the mechanical structures. The analysis is morphology-dependent and can be used to analyze the optimal arrangement of components such as actuators and sensors to meet the robot's goal [44].

The Universal Robot Description Format (URDF) is used [2] to introduce the robot model in ROS. URDF is an XML file format used in ROS to describe the dynamic and kinematic properties of a robot. The two main basic elements are ⟨joint⟩and ⟨link⟩. The ⟨link⟩forms the structure of the robot, such as its physical forms and mass, friction, bounce factor, and rendering characteristics like color, texture, and transparency. The ⟨joint⟩forms the connections between links of the robot body and sets motion constraints. Figure 4.4 shows the urdf model of the created robot. The created robot comprises multiple links and joints, a LiDAR sensor is linked to the main body of the robot. The base_link is the robot's rigid body. As the links are connected by joints, then the position of a link is determined by the joint that connects it. The xyz and rpy parameters specify the coordinate transformation relative to the parent link, where xyz is the relative position and rpy represents roll, pitch, and yaw angles in radians.

To integrate ROS with stand-alone Gazebo, gazebo_ros_pkgs, a collection of ROS packages, provide interfaces to simulate robots in Gazebo using ROS messages, services, and dynamic reconfiguration [1]. From the package, it provides Gazebo plugins [37] to give the models access to the functionalities through C++ classes and tie in ROS messages and service calls for motor inputs and sensor output. The skid steering drive plugin provided by $libgazebo\_ros\_diff\_drive.so$ is used to control the wheels of the robot.

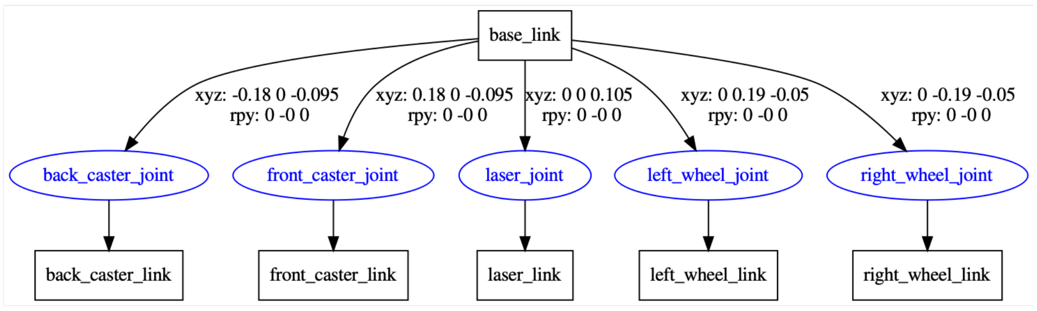

Figure 4.4: The urdf model of the robot

4.2.3 2D LiDAR sensor

LiDAR is a remote sensing technology, configured as a range measurement sensor that uses rotating beams to repeatedly transmit pulsed light into 360° direction, and calculate the distance between the obstacle and itself based on the reflected pulsed light. By repeating the operation in quick succession, the LiDAR uses the surface it measures to establish a complex distance map of the surrounding operating environment. LiDAR has 2D and 3D types. 2D LiDAR does not have depth information and is flattened by 2D space, while 3D LiDAR generates a large amount of point cloud data when obtaining height information, which leads to high computation cost and is more expensive than 2D LiDAR [19]. In this project, we focus on a 2D LiDAR sensor.

The RPLiDAR [3] is a 360-degree two-dimensional laser range finder. While measuring the range, it rotates 360 degrees and continuously obtains angle information. Figure 4.5 illustrates the LiDAR scanning in one axis with a single beam. The upper images show the laser path going through a basic scene and the bottom images depict the LiDAR sensor output in Cartesian coordinates. The blue square and the green circle represent the LiDAR and an obstacle respectively. The LiDAR has a maximum range of 6 meters. The motor speed frequency ranges from 5 Hz to 15 Hz; its ranging frequency can be customized, which is up to 8 kHz. The measurement performance of YDLiDAR X4 is shown in Table 4.2. In the Gazebo simulator, this sensor attached to the robot is exposed by a Gazebo laser scan plugin interface to acquire point cloud data, which is provided by *libgazebo_ros_ray_sensor.so*.

Parameters	Values
Sampling Frequency	8000 Hz
Scanning Frequency	5Hz-15Hz
Range	0.1m - 6m
Angular Range	360 deg
Distance Resolution	1% of the range
Angular Resolution	1 deg

Table 4.2: The measurement performance of RPLiDAR

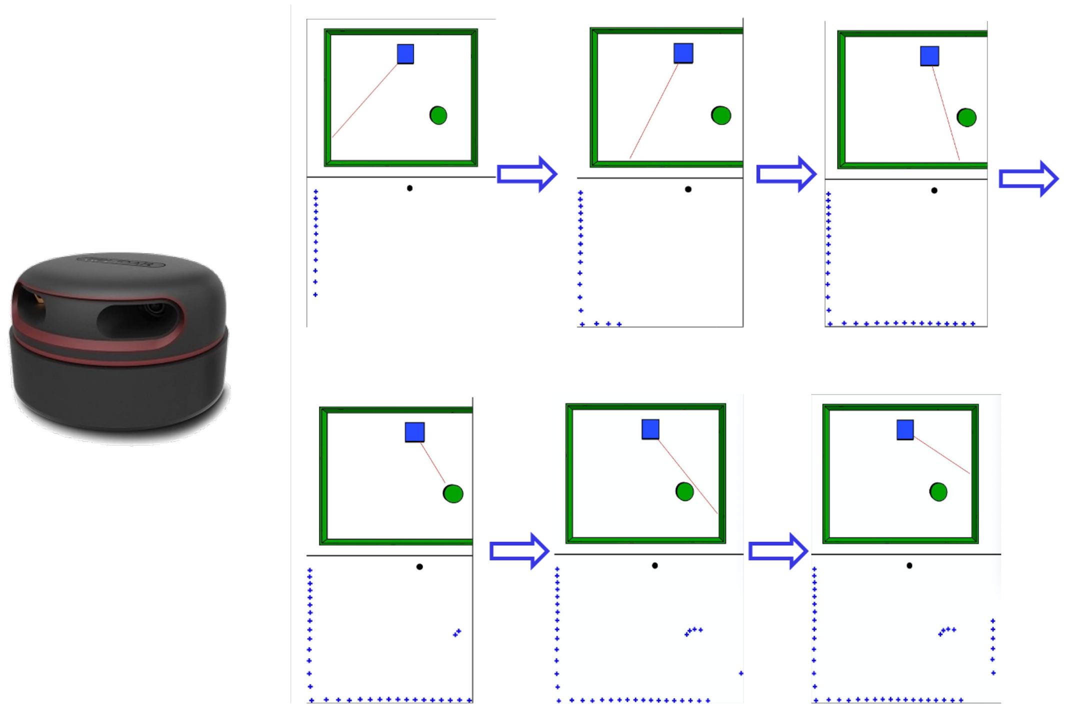
Figure 4.5: RPLiDAR and an overview of a LiDAR scanning in one axis with a single beam [29]. The blue square and the green circle represent the LiDAR and an obstacle respectively. The top images depict the laser's path through a basic scene with a cylinder obstacle; the bottom images show the sensor's output after it has been converted from polar to Cartesian coordinates.

4.3 Case studies

This section presents the robot and the simulation environment in Gazebo used to perform the obstacle detection task.

4.3.1 Environment simulation in Gazebo

The physics of the represented environment must be simulated visually. In Gazebo the graphical environment is called *world*, in which various static and/or dynamic objects are represented [44]. A world file contains a set of robots and objects in the simulated environment, including the robot model, the created environment, and global parameters such as ambient lighting and physics properties. Gazebo has a graphical *building editor* to assist in the creation of the environment. We may change the levels of the structures, edit the walls, floors, windows, doors, add colors and textures with this tool.

The world environment in Gazebo is shown in Figure 4.6. The spheres, cubes and cylinders, and walls denote obstacles in the simulation. The objects and models created are linked to the Gazebo world, their sizes and detailed information are listed in the URDF files. The spheres and cylinders have a radius of 0.5m. The edge of the cube is 1m. The wall covers the dimensions of 12x12 m^2. Table 4.3 lists the coordinates values of the center of the obstacles on the x-y plane with respect to the Gazebo world origin.

4.3.2 Interaction between robot and world simulation

The movement of the robot in the simulation can be conducted via the keyboard keys, which commands the robot to move in the forward, left, and right direction. The *teleop_node* converts the linear and angular input velocities and publishes the robot speed information to a topic named

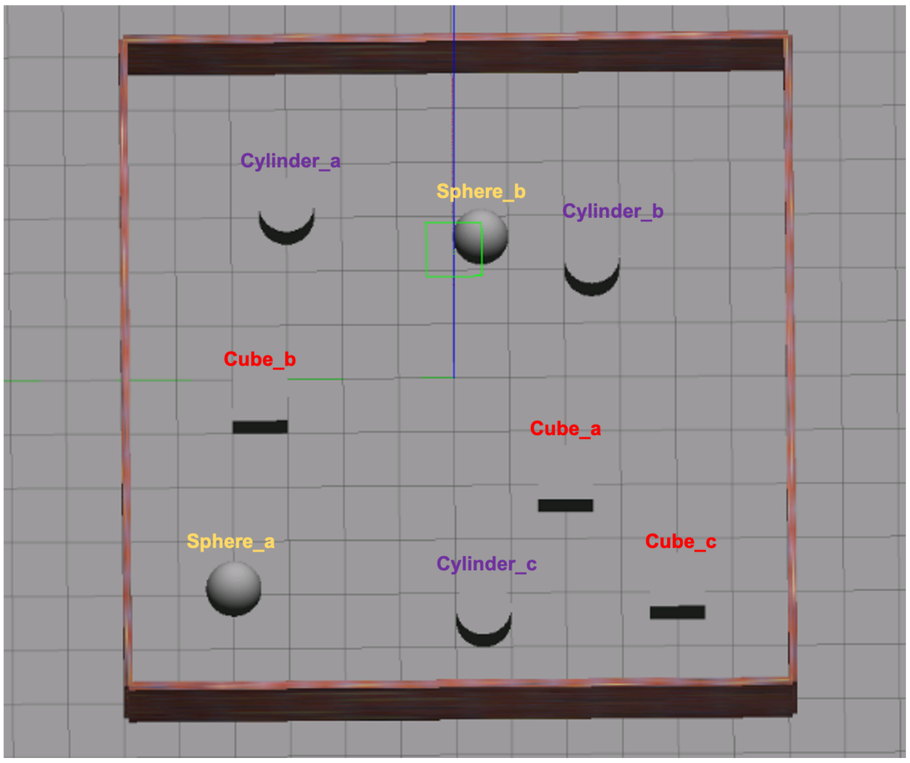

Figure 4.6: The created Gazebo world environment

Obstacle	X-coordinate (m)	Y-coordinate (m)
Sphere_a	-4	4
Sphere_b	2.5	-0.5
Cube_a	-2	-2
Cube_b	-0.5	3.5
Cube_c	-4	-4
Cylinder_a	3	3
Cylinder_b	2	-2.5
Cylinder_c	-4.5	-0.5

Table 4.3: The coordinate of the obstacles in the world

cml_vel. Following that, the controller steers the robot's wheels responding to the received velocity inputs.

With the LiDAR attached to the robot, ROS initiates the communication between them. When ROS sends a request message to the LiDAR, the LiDAR sends a return message back and starts to perform related operations after receiving the request message from ROS. The LiDAR attached publish sensor_msgs/LaserScan message to the *scan* topic by the Gazebo plugin.

The data generated from these topics can then be used for the obstacle detection algorithm.

Chapter 5

Selection of the algorithm

As discussed in Chapter 2, there are two mains approaches of spatial data representation for obstacle detection grid-based and vector-based forms. Compared to the first category, the vector-based approach with the compact representation of the surrounding environment is especially suited for describing a sparse scenario due to its low memory consumption. In this thesis, we concentrate on obstacle detection by 2D vector-based representatives.

The process of 2D vector-based obstacle detection algorithms available in the literature generally contains three parts: point cloud preprocessing, segmentation and clustering, and feature extraction. The obstacle information is obtained via LiDAR and stored in point clouds. Point cloud preprocessing consists primarily of correction and filtering, with correction restoring the distorted point cloud and filtering removing unnecessary points. Segmentation and clustering aim to group measured laser point clouds and extract ordered obstacle information from cluttered point clouds and construct intuitive obstacle information. The feature extraction considers certain shape properties of the objects expected to be identified.

The algorithms available in the literature have a similar process as discussed and are slightly different in each step. In this chapter, we present the selection of an obstacle detection algorithm. The study of the work in three papers from [15, 34, 39] are described and the selection is justified.

5.1 Algorithm framework

The framework of the algorithms is shown in Figure 5.1. Considering the properties of the laser point cloud data, the whole process is mainly divided into these steps: data preprocessing, segmentation and merging, feature extraction, and obstacle structure prediction.

5.2 Data pre-processing

The accuracy of measurement may be affected by environmental factors. The outliers can lead certain laser points to fail to represent an obstacle because they may be unstable or noise points in real scenarios. Thus, the laser data points need to be processed further to get closer to the real data after acquisition.

The point clouds obtained by LiDAR is in the polar coordinates, a set p with N measurement points is from each scan. We represent each raw point p_i in p in polar coordinate (R_i, θ_i) as follows:

$$p \triangleq \{p_i = (R_i, \theta_i)\}, i \in [1, N] \tag{5.1}$$

To facilitate the calculation for subsequent steps, the sequence is converted from the polar coordinate system to the Cartesian coordinate system. The coordinate conversion formula is as below.

$$\begin{cases} x_i = R_i cos\theta_i \\ y_i = R_i sin\theta_i \end{cases} \tag{5.2}$$

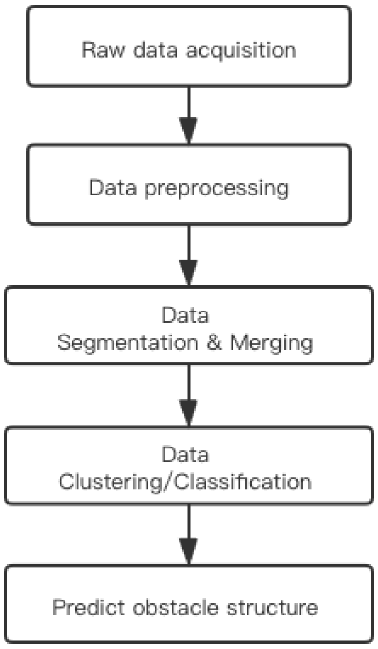
Raw data acquisition
Data preprocessing
Data Segmentation & Merging
Data Clustering/Classification
Predict obstacle structure

Figure 5.1: Obstacle detection algorithm process flow

In [15, 34], the authors use a 3x3 median filtering window at the laser-point cloud data. A range filter is also integrated to exclude useless points with invalid values or beyond the range. Median filtering can remove occasional noise peaks while maintaining the integrity of the original data. A 3-neighbor temporal median filter window shown below is applied to clean up noise in the data from the LiDAR sensor. At the current time t, the distance of a laser point i from a laser scan is $R(t, i)$. The median of the nine values in the median filter matrix is taken as the distance between the laser point i and the LiDAR attached to the robot at time t. In this way, it discards noise points taking space and time into account, decreases the complexity, and improves the accuracy of obstacle detection.

$$\begin{bmatrix} \text{R(t-1,i-1)} & \text{R(t-1,i)} & \text{R(t-1,i+1)} \\ \text{R(t,i-1)} & \text{R(t,i)} & \text{R(t,i+1)} \\ \text{R(t+1,i-1)} & \text{R(t+1,i)} & \text{R(t+1,i+1)} \end{bmatrix} \tag{5.3}$$

5.3 Splitting and clustering

The distance data collected by the LiDAR is preprocessed to generate the point cloud as the input of the obstacle detection algorithm. The algorithm extracts local geometrical obstacles from 2D laser point cloud data. The visible part of the laser point cloud sampled from surrounding objects is interpreted.

Due to the uneven distribution of laser-scan points, the measurement density in the LiDAR data diminishes and point distribution becomes sparse as the measured distance grows. After filtering out the noise points, the distance between consecutive laser points is first examined to distinguish laser-point cloud blocks for each frame. The distance difference of two sequential points, point $\vec{p}_i$ with distance R_i at angle θ_i and its consecutive point $\vec{p}_{i-1}$ with distance R_{i-1} at angle θ_{i+1} $(\theta_i + \Delta\theta)$, should approach 0 for all laser points bundled on the same obstacle. If the distance between two consecutive laser points is less than a certain threshold, the two points belong to the same laser-point cloud block. Otherwise, the point is assigned to represent a new obstacle [45].

In [39]: The thresholds for segmentation are selected for different laser scan points to be consistent with the real environment feature model. In this step, a collection of point subsets S_j ($j \in 1, ..., N_S$) is provided to represent possibly separate obstacles. The point $\vec{p}_i$ is assigned to the same laser-point cloud block of point $\vec{p}_{i-1}$ in the case of the distance criterion below, where $d(R_{i-1}, R_i)$ denotes the measurement distance between points $\vec{p}_i$ and $\vec{p}_{i-1}$, R_i is the range of laser point $\vec{p}_i$, d_p is defined as the proportion factor to enlarge the distance between consecutive points, and d_{group} defines the parameter for grouping the points. Finally, for each frame of distance data, the laser-point cloud is divided into several independent laser-point cloud groups. The process is shown in Figure 5.2.

$$d(R_{i-1}, R_i) < R_i d_p + d_{group} \tag{5.4}$$

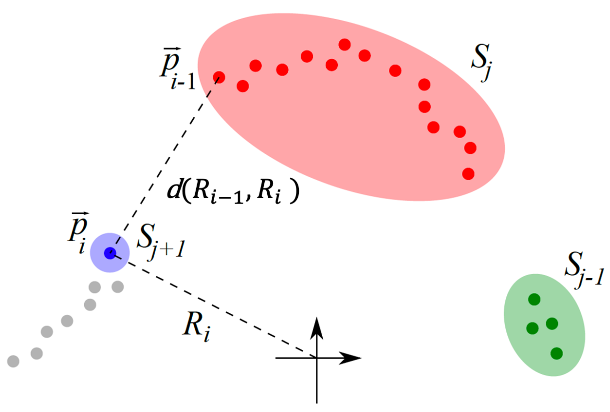

Figure 5.2: Grouping laser-point groups [39]

Each laser-point cloud group S_j is further subdivided into separate point subsets. If the number of points in a separate group exceeds the threshold N_p, the procedure of splitting relies on the Iterative End Point Fit algorithm, which builds a line to connect the start point $\vec{p}_s$ and termination point $\vec{p}_t$ of the group, then identify the point $\vec{p}_f$ within the group locating the farthest from the line and acquire the maximum distance D_{max} between this point and the line. The procedure is illustrated in Figure 5.3. The laser-point cloud group that containing fewer points than the threshold N_p is not processed for simplicity. The criterion that divides the groups into point subsets is defined as below:

$$D_{max} > R_f d_p + d_{split} \tag{5.5}$$

where R_f denotes the range of the farthest laser point $\vec{p}_f$ from the line in the group and d_{split} defines the parameter for subset splitting. For each obtained new subset, the above process is recursively repeated until all the subsets fail to match the split criterion.

In [15]: The threshold for segmentation is set in the case that if the distance $d(R_{i-1}, R_i)$ between points $\vec{p}_i$ and $\vec{p}_{i-1}$ is larger than kW, the laser-point cloud block is segmented into two small blocks, where W is the robot width and k is a dynamic amplification factor to adjust the distance threshold between the two consecutive points with the width of the robot and the measured distance changing, as shown below. T_d is a threshold of the measured distance.

$$\begin{cases} k = \frac{W \cdot R_i \cdot R_{i-1}}{100(R_i + R_{i-1})}, \frac{R_i + R_{i-1}}{2} > T_d \\ k = 0.5, 0 < \frac{R_i + R_{i-1}}{2} \leq T_d \end{cases} \tag{5.6}$$

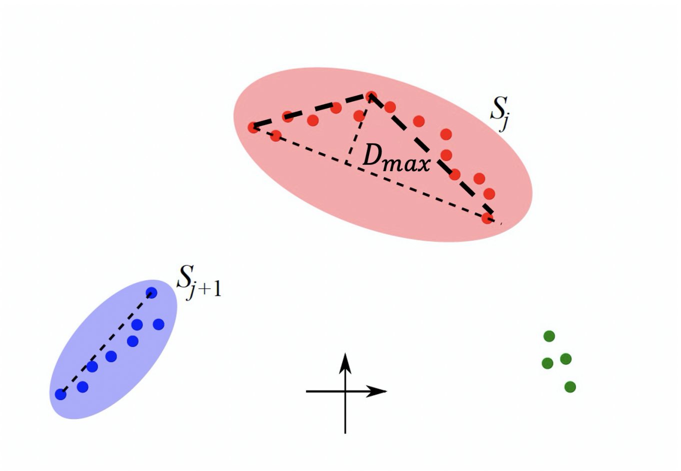

Figure 5.3: Splitting laser-point groups [39]

The two laser-point cloud blocks can be clustered into one when the gap between two blocks $L < W$. N compensated points are added between the two blocks to make them spatially continuous. The distance of the compensated point i is represented as follows, where d_{aver} is the average distance of the adjacent points of two point-cloud blocks.

$$R_i = R_p + i \cdot \frac{R_q - R_p}{N}, N = \frac{L}{d_{aver}} \tag{5.7}$$

In [34]: The proposed algorithm uses the same method as [15] in this step, except that the parameter k is not specifically defined.

5.4 Obstacle structure classification

After splitting and clustering, the raw LiDAR point cloud forms a set of independent point cloud blocks. The blocks can subsequently be expressed as fundamental outlines, such as line, rectangular, and circle shapes. The underlying model of the obstacles that can be approximated is considered acceptable since the shapes extracted mean to enclose the laser points while not covering much open space rather than find a best-fitting graph that represents the points. The geometric approximation of the features of the blocks brings the benefits of simplifying the computation. Instead of using a collection of discrete laser points, simple parameters like the radius and position of extreme points are used to represent the obstacles. Also, the averaging effect of model extraction can be robust to the measurement noise, which may be accompanied by sacrificing some unnecessary motion space loss of the robot.

To express the obstacles, a set of obstacles is constructed as $\mathbb{O}$, $\mathbb{O} \triangleq \{\mathbb{L}, \mathbb{C}, \mathbb{R}\}$, where $\mathbb{L}, \mathbb{C}, \mathbb{R}$ describes line, circle and rectangle shape obstacles.

5.4.1 Segmentation and merging

In [39]: After splitting, every point cloud subset S_m $(m \in 1, ..., N_S^+)$ is extracted using a fitting line l_m. The fitting line requires finding the optimum equation of lines that is close to the discrete N_m laser points in the subset [10]. The process is shown in Figure 5.4. There are many line fitting methods, including the standard least-squares method and the total least squares regression method. The standard least-squares method only minimizes the sum of the squared vertical distances between the laser points and the fitting line, which may result in inadequate parameter estimation, while the total least squares regression minimizes the sum of the squared perpendicular distance of each laser point to the fitting line [20]. For this reason, the total least squares regression is adopted to model the fitting line with the equation:

$$ax + by + c = 0 \tag{5.8}$$

where a, b, c are coefficients, and at least one of a and b is not 0. The distance d_l from a laser point (x, y) within the subset to its fitting line is calculated by the equation (5.9).

$$d_l = \frac{|ax + by + c|}{\sqrt{a^2 + b^2}} \tag{5.9}$$

To provide a solution to the line fitting model, set $c \neq 0$ and a, b can be solved by equation (5.10). The final segment is constructed by projecting the start and termination points onto the fitting line.

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix}^{\dagger} \begin{bmatrix} -c \\ \vdots \\ -c \end{bmatrix} \tag{5.10}$$

A merging process is considered to combine the extracted segments. From Figure 5.4, d_0 is the distance between the end point of the subset S_m and the start point of the subset S_n ($S_m, S_n \in \mathbb{S}$). $l_{threshold}$ is the threshold for connecting the segments. The closeness of the two segments should satisfy $d_0 < l_{threshold}$. Then the new fitting line is constructed based on the combined segment.

The segments collinear is also checked. In criterion (5.11), d_1, d_2, d_3, d_4 are distances between the starting and termination index points of segments and the new fitting line, $d_{threshold}$ is a parameter for combining the two segments into one segment. If both conditions are met, the extreme points of segments are projected onto the new fitting line and the two original segments are combined into a new segment.

$$max(d_1, d_2, d_3, d_4) < d_{threshold} \tag{5.11}$$

The extracted segment is compared with any other segments to see if they can be merged, as the merging process can happen between any two segments in the set. Set $\mathbb{L}$ is composed of the final extracted segments.

In [15], if the number of points in a separate group exceeds the threshold N_p, equation (5.8) is also used to connect the starting point and termination point of the group, then identify the point $\overrightarrow{p}_f$ within the group locating farthest from the line and acquire the maximum distance D_{max} between this point and the line. $|s|$ is the distance between the starting point $\overrightarrow{p}_s$ and the termination point $\overrightarrow{p}_t$. If $D_{max} < 0.2\,|s|$, this point-cloud block will be fit to a line segment with the two endpoints. Otherwise, if $D_{max} > 0.2\,|s|$, for each obtained new subset, the above process is recursively repeated until the number of points in a separate group is smaller than the threshold N_p.

While **in** [34], it is different from the method in [15] that if $D_{max} > 0.2\,|s|$, the points in the group are clustered into rectangle, which return the rectangle's four endpoints. This method can be treated as 4 lines.

5.4.2 Circle extraction and merging

In [39]: The circle shape is extracted based on each segment l_n in the set $\mathbb{L}$. Firstly, an equilateral triangle with a base of the segment and its central point that is not at the origin. Then, using the triangle as a foundation, we create an escribed circle with radius and center as in Figure 5.5, where r_n is the radius of the circle and $\bar{l}_n$ is segment length, $\vec{n}_n$ is the normal vector of the segment pointing towards the origin. The radius of the circle is also enlarged by a margin value defined as a safe distance r_d. The circle is added to the set $\mathbb{C}$ if the resulting radius is smaller than the threshold r_{max}, otherwise, it will be discarded.

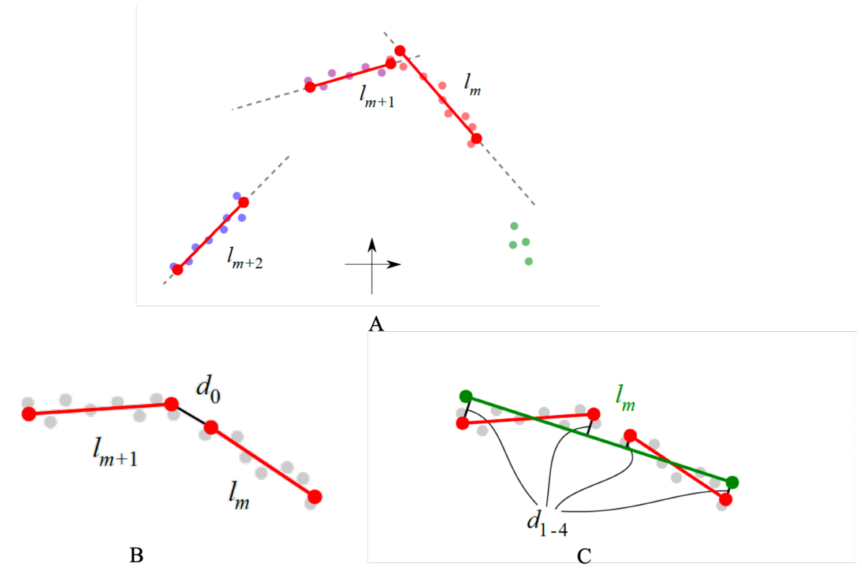

Figure 5.4: Segmentation from each subset (A), testing the connection between segments (B), testing segments collinear (C) [39]

$$r_n = \frac{\sqrt{3}}{3}\bar{l}_n, \vec{p}_{0n} = 0.5(\vec{p}_{1n} + \vec{p}_{2n} - r_n\vec{n}_n) \tag{5.12}$$

The circles in the set are further processed to a merging process. When two circles intersect, a new segment is formed between their centers, and a new circle is formed upon this segment. The new circle's radius is then enlarged by the larger obstacle's radius. If it is smaller than r_{max}, the new circle replaces the two previous ones to add in the set $\mathbb{C}$. Figure 5.5 shows the process of extracting a circle from a segment and circles merging.

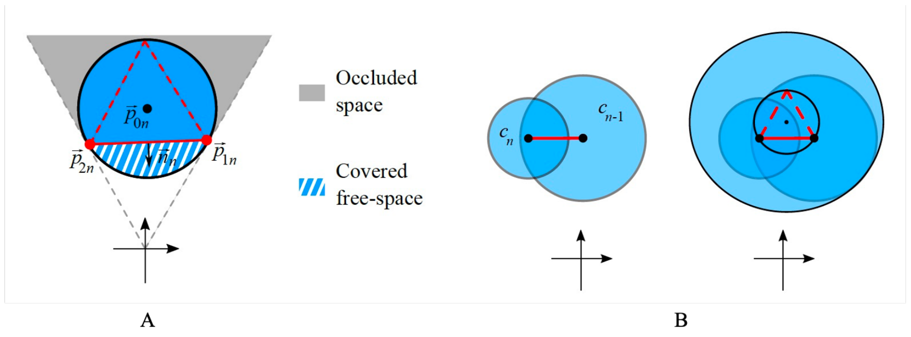

Figure 5.5: Extracting circle from a segment (A), circles merging (B) [39]

In [15, 34], if the number of points in the point-cloud block is less than N_p, the circle shape is extracted. The midpoint of the line segment between the starting point p_s and termination point p_t is set as the center of the circle.

Specifically, **In** [15], when $D_{max} < 0.2\,|s|$, the radius is the distance between the center and the maximum-distance point in the group. Otherwise, an algorithm based on vector cross product is used to determine the convexity or concavity of the point-cloud block. The radius of the circle is also enlarged by a margin value as in [39]. The process is shown in Figure 5.6. The right three graphs depict how to extract the circle shape based on the convexity or concavity of the point-cloud block.

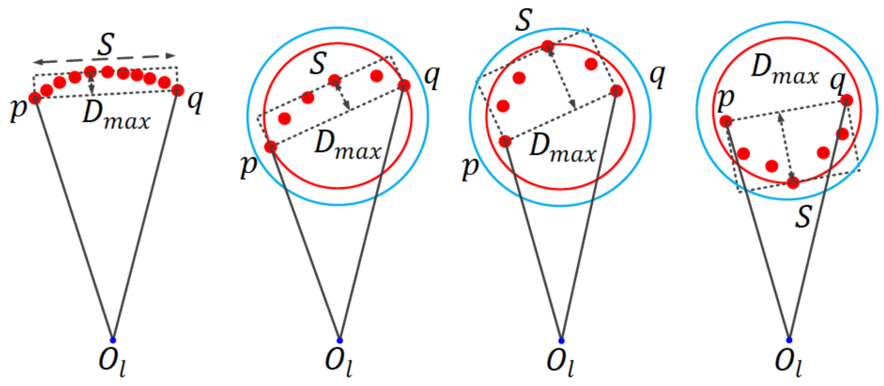

Figure 5.6: The circle shape extraction [15]

5.5 Algorithm selection

Based on the study of the papers, the obstacle detection algorithms discussed have a similar process flow. As the presented papers not only focus on obstacle detection but also involve obstacle tracking or avoidance algorithm, the direct experiments and performance of the obstacle detection algorithm are not given. Since our goal is not to propose an obstacle detection algorithm, we adopt the work in paper [39]. It provides a ROS *obstacle_detector* package, which provides utilities to detect and track obstacles from data provided by 2D LiDARs and is easier for us to work on pipelining or paralleling the existing algorithm.

The representation of the line and the circle obstacle models is threefold. First of all, large obstacles such as long walls are expressed by lines with two endpoints. Then, smaller obstacles such as pillars or desks in real scenarios are modeled as cylinder shapes. Last, the circular model in the 2D plane requires fewer computing resources compared to occupancy grids. By adding a margin to the dimension of the circles, it allows a safe distance between the robot and the obstacle. In terms of preprocessing, we first apply the median filter to remove noise from the 2D point cloud obtained from the 2D LiDAR before segmenting it into discrete laser-point cloud blocks. After that, the grouping of laser-point cloud blocks is accomplished based on the distance gap between consecutive laser points, followed by splitting laser-point cloud blocks using Iterative End Point Fit algorithm. Furthermore, all detected obstacles are modeled as lines and circles.

Chapter 6

Pipelined implementation in ROS

6.1 Implementation of the normal version in ROS

6.1.1 Overall structure

In general, the obstacle detection algorithm runs as an independent node and subscribes messages from the */scan* topic published by the LiDAR attached to the robot. If the algorithm works properly, the algorithm will publish the custom geometry information of line-shape and circle-shape obstacles and point clouds of the filtered LiDAR data and obstacles. The ROS system architecture for the obstacle detection node is illustrated in Figure 6.1.

The process flow of the detection can be divided into several components, including subscriber, median filtering, point processing, and publishers. Some Details of the components are explained as follows:

- Subscriber. The subscriber is responsible for receiving messages by subscribing to */scan* topic from the simulation. It makes a call to the ROS master node, which keeps a registry of the subscribing and publishing nodes. The messages in the topic are passed to a callback function and fed into the detection node with time stamps.
- Median filtering. When the callback function gets called, a 3x3 median filtering matrix is applied to new messages of the laser data.
- Points processing. This component processes the filtered LiDAR data and contains the main execution of the obstacle detection algorithm, including grouping points, merging segments, detecting circles, merging circles, and publishing obstacle information.
- Publishers. The publishers keep publishing geometry information of the obstacles, which are defined in ROS and custom message format. The publishers are registered to the ROS master node.

6.1.2 Obstacle detection in ROS

The implementation structure of the obstacle detection algorithm is shown in Figure 6.2.

Median filtering

The LiDAR data is passed in as a ROS laser data format $sensor\_msgs :: LaserScan$. Each raw measurement point is expressed in the form of polar coordinates (distance and angle value) and Cartesian coordinate relative to the LiDAR. The distance value is also restricted within the range limit which is defined in the $sensor\_msgs :: LaserScan$. After that, the 3x3 median filtering matrix mentioned in (5.3) is applied to smoothing the laser point data.

The implementation of median filtering is shown in the pseudocode in Algorithm 1.

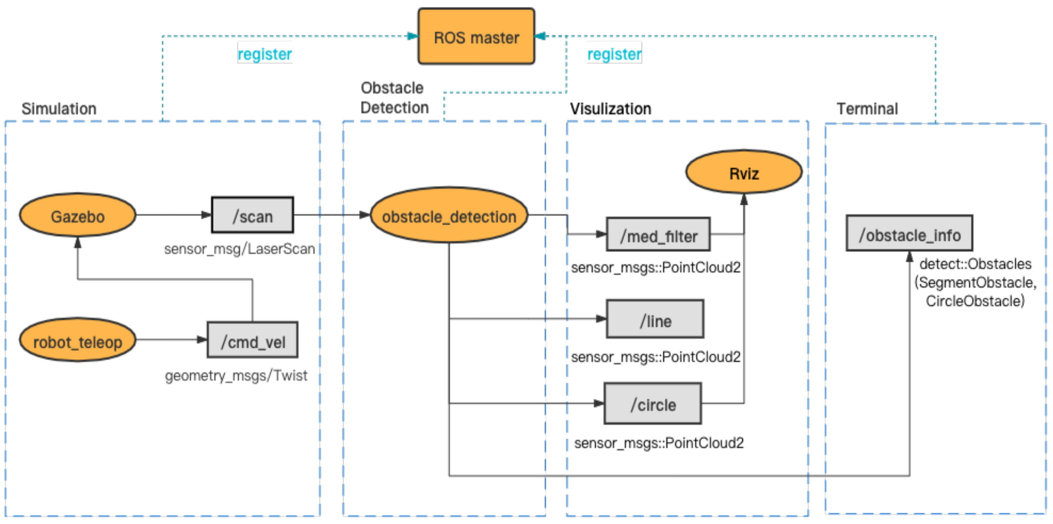

Figure 6.1: ROS Implementation architecture for obstacle detection

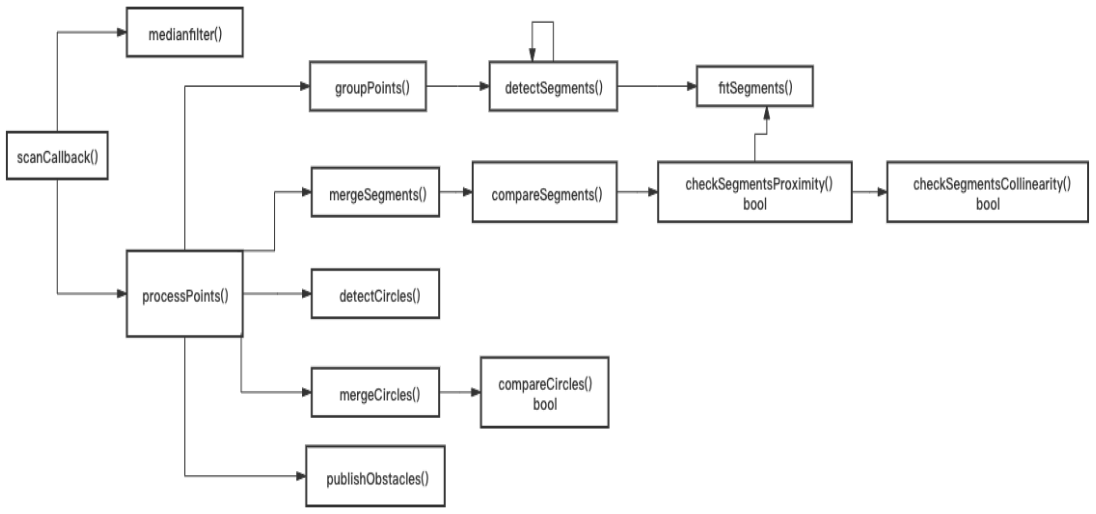

Figure 6.2: Structure of the obstacle detection algorithm

Point cloud processing

The point cloud is generated after preprocessing distance data which serves as input to obstacle detection algorithm. The algorithm consists of several functions: *groupPoints*(), *mergeSegments*(), *detectCircles*(), *mergeCircles*(), *publishObstacles*().

- *groupPoints*() : the laser points in the filtered laser point cloud are divided into several independent groups following the criterion 5.4, while also detecting segments for each point cloud block using a function *detectSegments*(). The *detectSegments*() function uses the Iterative End Point Fit algorithm to split the divided independent groups. Then if the criterion 5.5 is met, for each obtained new subset, the function is recursively repeated until all the subsets fail to match the split criterion. Also, to approximate the line fitting model for every laser point subsets, a *fitSegment*() function uses Armadillo C++ library [46] for

Algorithm 1 Implementation of the 3-neighbor temporal median filtering

```
Input: curPoints: vector storing the laser points at time $i$; prevPoints: vector storing the laser
    points at time $i-1$; prevPlusPoints: vector storing laser points at time $i-2$;
Output: inputPoints: vector storing laser points for further steps after median filtering
 1: init ← bool parameter checking whether to start median filtering, initindex ← index to count
    the frame of received laser data, tmpRadius ← vector storing the distance of points in the
    3x3 median filtering matrix.
 2: if $init \leftarrow false$ then
 3:     repeat
 4:         $prevPlusPoints.swap(prevPoints)$;
 5:         $prevPoints.swap(curPoints)$;
 6:     until $initindex > 3$;
 7:     $init \leftarrow true$ ;
 8: else
 9:     for $i \leftarrow$ 1to $curPoints.size - 1$ do
10:         if $i < prevPlusPoints.size - 1$ and $i < prevPoints.size().size - 1$ then
11:             push back ranges of $curPoints[i]$, $curPoints[i+1]$, $curPoints[i-1]$ into $tempRadius$;
12:             push back ranges of $prevPoints[i]$, $prevPoints[i+1]$, $prevPoints[i-1]$;
                    into $tempRadius$;
13:             push back ranges of $prevPlusPoints[i]$, $prevPlusPoints[i+1]$,
                    $prevPlusPoints[i-1]$ into $tempRadius$;
14:             $sort(tmpRadius.begin(), tmpRadius.end())$;
15:             push back range of $tempRadius[4]$ into $inputPoints$;
16:         end if
17:     end for
18:     $PrevPlusPoints.swap(PrevPoints)$;
19:     $PrevPoints.swap(curPoints)$;
20: end if
```

matrix algebra.

- *mergeSegments*() : as explained in Chapter 5, the segment merging process means to find pairs of segments that could comprise a single object, which is composed of two parts, the connectivity test and collinear test. These two tests are executed separately by bool function *checkSegmentsProximity*() and *checkSegmentsCollinearity*().

- *detectCircles*() : this function extracts circles based on the extracted segments and enlarges the radius by an additional margin.

- *mergeCircles*() : the circle merging process compares the two circles and forms the final circles as described in section 5.4.2.

- *publishObstacles*() : two types of messages are published, one is point clouds ($sensor\_msgs :: PointCloud2$ type) of line segments and circles, and the other is custom message format $detect :: Obstacles$, which consists of arrays of *SegmentObstacle* and *CircleObstacle*.

  A *SegmentObstacle* message has the following attributes:

  – first point [$geometry\_msgs/Point$]: the coordinate position of start point of the segment relative to the LiDAR.

  – last point [$geometry\_msgs/Point$]: the coordinate position of the termination point of the segment relative to the LiDAR.

  A *CircleObstacle* message contains the following attributes:

– center [$geometry\_msgs/Point$]: the coordinate position of the central point of the extracted circle.

– true_radius [$float64$]: the true measured radius of the circle.

– radius [$float64$]: the measured radius with additional margin.

6.2 Simulation results

6.2.1 Experiments

The algorithm's required parameter values are chosen empirically, based on intuition and trial. The list of parameters and their values used for the experiment is shown in Table 6.1.

Name	Parameter	Value
distance proportion	d_p	0.006 rad
min. group points num.	N_p	5
group threshold	d_{group}	0.055 m
splitting threshold	d_{split}	0.5 m
segment connectivity threshold	$l_{threshold}$	0.5 m
segment spread threshold	$d_{threshold}$	0.5 m
radius margin	r_d	0.3 m
max. circle radius	r_{max}	0.9 m

Table 6.1: List of parameters used for the experiment

Figure 6.3 shows the simulation result of the algorithm in ROS visualization tool Rviz[1]. The robot is located at the origin of the simulation. In the right image, the white points are raw LiDAR data, while the blue points indicate the filtered LiDAR data. The green points are laser point cloud that published in the $/circle$ topic, which means that the detected obstacles are modeled in the circle shape.

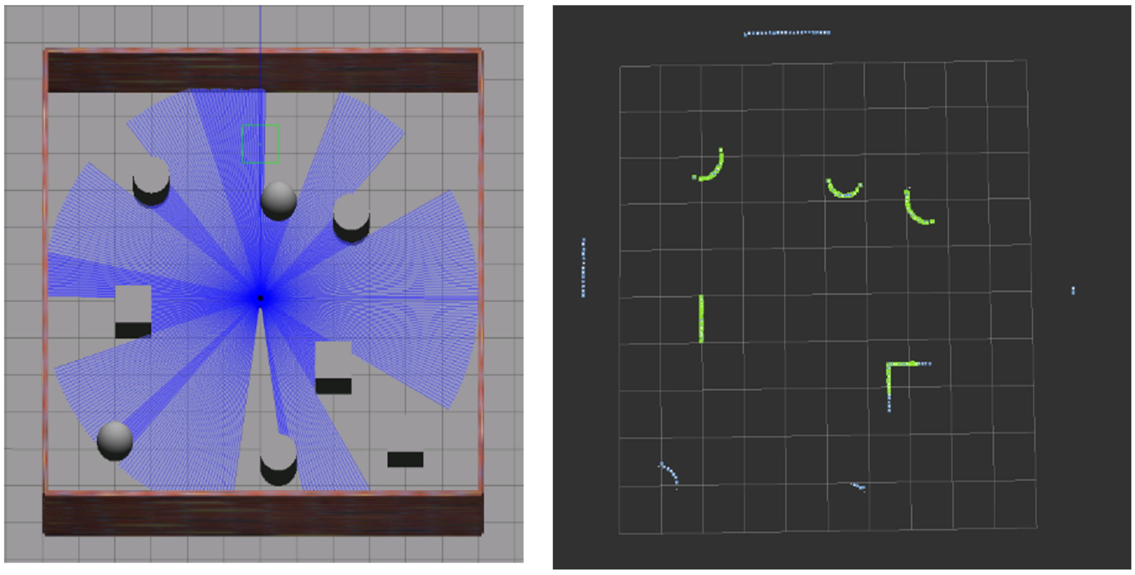

Figure 6.3: Simulation result of obstacle detection algorithm and visualize in Rvz

[1] http://wiki.ros.org/rviz

The obstacle information published in topic */obstacle_info* is shown below. From the snippet, the five objects in the simulation are all extracted as circles. The x-y coordinate of the central point of circles and their measured radius and radius with added margin are also given.

```
header:
  seq: 379
  stamp:
    secs: 1321
    nsecs: 857000000
  frame_id: "laser_link"
segments: []
circles:
  -
    center:
      x: -1.73287030017
      y: -1.70470579381
      z: 0.0
    true_radius: 0.325738516696
    radius: 0.625738516696
  -
    center:
      x: 1.810270813
      y: -2.28750880609
      z: 0.0
    true_radius: 0.353169548662
    radius: 0.653169548662
  -
    center:
      x: 2.2952118137
      y: -0.466178677633
      z: 0.0
    true_radius: 0.327052671133
    radius: 0.627052671133
  -
    center:
      x: 2.75255870275
      y: 2.78722595582
      z: 0.0
    true_radius: 0.295148330745
    radius: 0.595148330745
  -
    center:
      x: -0.491135327639
      y: 3.26856169893
      z: 0.0
    true_radius: 0.538672655174
    radius: 0.838672655174
---
```

6.2.2 Execution time analysis

Table 6.2 shows the average execution time of the main functions in the obstacle detection algorithm. The robot locates at the position shown in Figure 6.3 and we sample 2000 frames of LiDAR data input to calculate the average timings of the functions. The *std* :: *chrono* library in C++ is used to find out the time consumed by different parts of the algorithm. We use the *system_clock* provided by the library to achieve the current system time. We get the timepoints before and after each function is called, take the interval of the time passed in between, and cast the duration to the millisecond time unit. As mentioned in Chapter 4, one scan of the LiDAR can supply 360 measurements corresponding to an angular range from 0 to 360 degrees with an angular resolution of 1 degree. The average filtered laser point size is 191. The *queue_size* is set to 1.

According to the pie chart shown in Figure 6.4, the *groupPoints*() function accounts for slightly over 60% of the total execution time. The *medianfilter*() function is the next longest function that makes up nearly 20% proportion of whole execution. It is also noticeable that except for the execution time of the main functions in the algorithm, the remaining work costs 11.56% of the execution time, which contains the time to clear data in the vectors, load different data types to the buffer, and other related work. The other main functions altogether in the algorithm take less than 10% of the proportion.

Function	Time (ms)
medianfilter()	0.0804
groupPoints()	0.2571
mergeSegment()	0.0148
detectCircles()	0.0015
mergeCircles()	0.0004
publishObstacles()	0.0164
others	0.0484
scancallback()	0.4189

Table 6.2: Function execution time in the algorithm

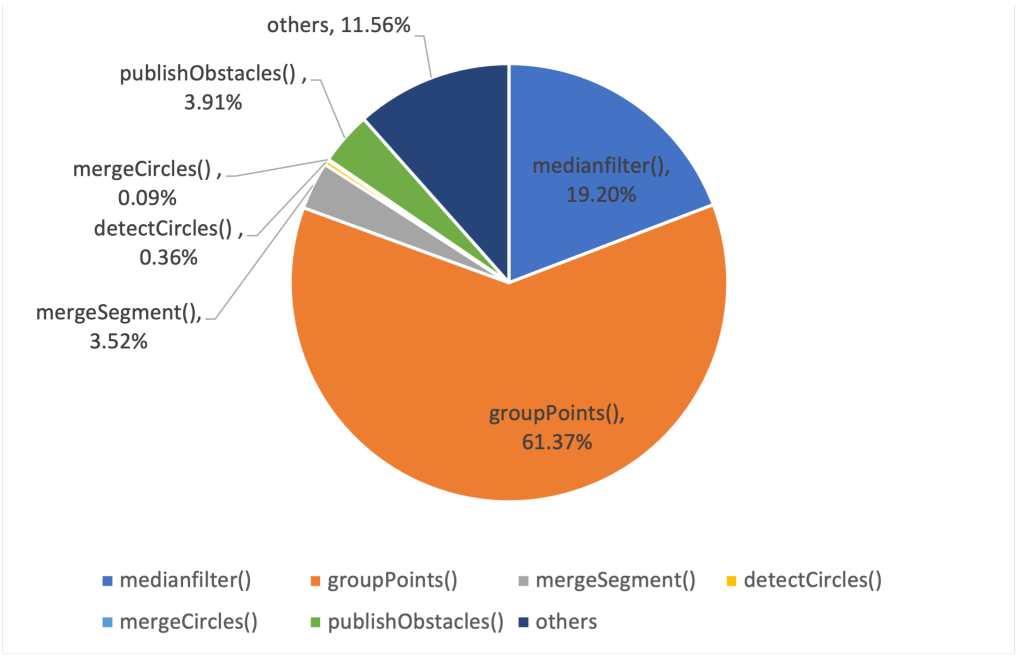

Figure 6.4: Pie chart of function execution time in the algorithm

6.3 Implementation of the pipelined version in ROS

The obstacle detection algorithm is sequentially executed and the implementation structure is shown in Figure 6.2, we can split the algorithm into a series of successive tasks which may contain a few of the functions so that the data flow in the direction specified by the interconnection structure. Thus, the algorithm can be formulated in a pipeline pattern. The principle of the pipeline is to divide the sequential steps of the algorithm into stages that run in parallel. Each pipeline stage passes necessary data to the next stage in the sequence.

From the execution time analysis in section 6.2.2, we can see that the $groupPoints()$ function and $medianfilter()$ function are the most time-consuming processing parts in the obstacle detection algorithm. Thus, we consider deploying these two functions as separate stages on different nodes and the other main functions in the algorithm are on one node. As a result, the stages are deployed on three ROS nodes. Each node subscribes to ROS topics containing messages that are the previous node's outputs. In this case, it is expected that the pipeline approach can yield an increase in execution processing speed.

While the ROS nodes are used to pipeline the algorithm, we need to consider the data overhead while streaming data between nodes as it may affect the performance of the pipeline. Data overhead refers to the time lost in transferring messages of inter-process communication between nodes. Data messages are serialized at the publisher's end, then transferred and deserialized at the subscriber's end. Therefore, it is possible that any processing time spent will be lost with regard to data overhead. The run time of the pipeline approach is determined by the run time of individual nodes.

Figure 6.5 shows the implementation architecture of the pipelined obstacle detection algorithm. Node 1 publishes and streams the filtered LiDAR data obtained from the $/scan$ topic to Node 2. Node 2 extracts raw segments and publishes a custom message format containing an array of point sets with attributes of a number of points $[int64]$, the coordinate position of extreme points of the segment $[float64]$, and a bool value indicating whether the point set is occluded by other point sets $[bool]$. The results are transferred to Node 3 and processed through the left main functions, the final obstacle information, and point clouds are published to the terminal and Rviz.

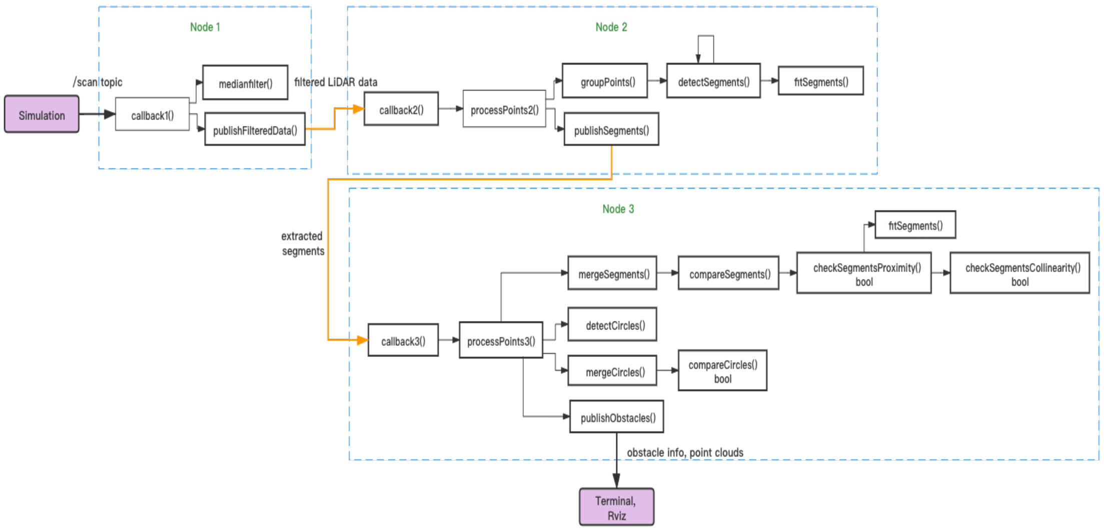

Figure 6.5: Structure of the pipelined obstacle detection algorithm

The pipelined algorithm is run in the same simulation as the normal version. Considering the performance of the implementation, the two main aspects are the algorithmic computation time and the communication overhead. Table 6.3 gives some time measurement of the three nodes in the pipeline, including the execution time of the main functions, the average communication overhead between nodes, the computation time of individual nodes, and the end-to-end latency. The transmitted data between nodes is different. Between node 1 and node 2, the filtered LiDAR data in the message format of $sensor\_msgs/LaseScan$ is transmitted, which contains 191 measurement points. An array of laser point subsets in the form of custom messages is sent from node 2 to node 3. The communication overhead is measured by recording the system-wide clock time from node i transmitting message to the message reception at node $i+1$ with an accuracy in the nanosecond range. The results are based on the sampled around 2000 frames of LiDAR data as input. The $queue\_size$ is set to 1.

Comparing to the execution time of the main functions in only one node shown in table

Node		Time (ms)
Node 1	medianfilter()	0.0810
	publish FilteredData()	0.0315
	Node 1 execution time	0.1050
Communication overhead between Node 1 and Node 2		0.2082
Node 2	receive data from Node 1	0.2545
	groupPoints()	0.1832
	publishSegments()	0.0366
	Node 2 execution time	0.2564
Communication overhead between Node 2 and Node 3		0.1776
Node 3	receive data from Node 2	0.0021
	mergeSegment()	0.0137
	detectCircles()	0.0011
	mergeCircles()	0.0006
	publishObstacles()	0.0026
	Node 3 execution time	0.0276
End-to-end response time		0.7708

Table 6.3: Execution time in the pipelined algorithm

6.2, we can see that for each separate function, the execution time in the pipelined version is similar with at most 0.001 ms time difference. The communication overhead is the time spent transmitting a message between nodes due to the ROS interaction and OS communication layer. The communication latency is measured as: $L_{i,i+1} = T_{i+1,start} - T_{i,end}$. Table 6.3 shows the average overhead $\overline{L}_{i,i+1}$ for all the two communication paths between three nodes. In ROS, the message communication among nodes is realized through network sockets using TCPROS and UDPROS, which are both connection-oriented and connection-less and defined on top of the standard TCP and UDP protocols. TCPROS transmits message data using standard TCP/IP sockets. Inbound connections are received via a TCP Server Socket, which includes a header with information of message data type and routing [48]. The time values for the latency in the table are in the sub-millisecond range when nodes are configured. However, the total transmission overhead takes up to around 50% of the end-to-end response time. Since the data packets transmitted between nodes in the implementation are not quite large, the latency stays rather steady, otherwise, the transmission latency would become rather unpredictable. The end-to-end response time of the pipelined algorithm is measured as a time duration from the start of node 1 to the end of node 3. The end-to-end latency increases 84% comparing to the execution time in running all in one node. Apart from data transmission between nodes, each node needs to publish or receive the data transferred, which also accounts for the time for data communication.

Chapter 7

Results and analysis

In Chapter 6, we analyze the performance of the normal implementation of the obstacle detection algorithm in one node and the pipelined version. The time measurement includes the execution time of the main functions in the obstacle detection algorithm, the average communication overhead between nodes, the computation time of individual nodes and the end-to-end latency. In addition, we evaluate two parameters that relates to the efficiency of the two implementation: 1) message loss: the messages unsuccessfully received and the total ROS messages sent over the network, 2) time measurement with different queue sizes, and 3) the throughput of the implementation.

7.1 Message loss

When creating publishers in ROS, the advertise() function is used to publish on a specific topic. The format is as follows:

```
ros::Publisher advertise(const std::string& topic, uint32_t queue_size, bool latch
    = false);
```

The second parameter to advertise() is the size of the outgoing message to be queued for publishing messages. If messages are published more quickly than we can send them, the number specifies how many messages to be buffered before dropping the old messages.

Also, when creating subscribers in ROS, the subscribe() call is used to show that we want to receive messages on a given topic. The format is shown below:

```
ros::Subscriber subscribe(const std::string& topic, uint32_t queue_size, <callback>
    , const ros::TransportHints& transport_hints = ros::TransportHints());
```

The *queue_size* indicates the maximum incoming message size that will be buffered before the the oldest ones are discarded if messages arrive faster than they can be processed. To subscribe to messages, the spin must be called to process events. $ROS :: spin()$ loops continuously until ROS invokes a shutdown and calls message callbacks as quick as possible when a new message arrives.

When a publisher is called, it adds the message into the queue and returns immediately. When the publish function returns, ROS decides what to do next through $ros :: spin()$, it may serialize and send incoming messages and publish. Hence, a queue is needed and it is useful for high-latency and low-bandwidth communication channels. The choice of the queue size depends on the system. Setting *queue_size* to none or zero is not recommended. The first one causes a blocking subscriber to block all publishing process and the latter one results in infinite memory usage. If the system is not overloaded then the dispatcher thread could pick up the queued messages within 1/10 second. Therefore, a queue size of 1, 2 or 3 would be sufficient for 10 Hz. Setting *queue_size* to one can make sure that new published data is processed. Bigger queue size with ten or more may prevent losing messages but uses more memory when processing falls behind.

In our experiments, we set a fixed scanning frequency of the LiDAR to 10 Hz, and increase the message queue sizes in the algorithm with the values 1, 10, 100 and 1000. All the queue sizes in the nodes are consistent and the experiment is carried out 2000 iterations.

- Normal version: as the measured in table 6.2, the execution time of the node is less than 0.5 ms, which is lower than the interval of the LiDAR sending data (0.1 s). Thus, the rate at which the LiDAR sends data is lower than the processing speed of the algorithm. The results of experiments show that the messages are all successfully received in the process.
- Pipelined version: the nodes are launched sequentially. Since the first node performs median filtering on the LiDAR data, the node publishes the filtered LiDAR data at the fourth frame based on the implementation.
  - *queue_size* =1: when the *queue_size* is set to one, the second node receives the 4th frame of the LiDAR data, and the third node executes 2 or three frames after the second node execution. After the third node receives the data, messages are successfully received.
  - *queue_size* =10: when the *queue_size* is set to 10, the second node receives the 7th frame of the LiDAR data, and the third node starts to execute 2 or 3 frames later than the execution of the second node. After the third node receives the data, messages are successfully received.
  - *queue_size* =100/1000: when the *queue_size* is set to 100 or greater, the second node receives around the 8th frame of the LiDAR data, and the third node starts to execute the same frame as the second node . After the third node receives the data, messages are successfully received.

The starting frames of each node in every experiment have slightly different results, as they can be affected by the initialization of launching the node and the execution of the operating system. Also, the low scanning frequency of the LiDAR used in the simulation indicates that the messages published in the */scan* topic would not arrive faster than they can be processed. According to the results, we can see that the few frames of messages are lost during initial transmission between nodes.

7.2 Latency

Figure 7.1 shows the sequential architecture of the obstacle detection algorithm running on a single node. In the figure, we have three LiDAR data sample frames from i to $i+2$. Each LiDAR sample data frame goes through the three stages, which correspond to the three components that are divided in the pipelined version. After completing all the stages, the next LiDAR sample data frame is obtained, and the three stages are completed consecutively.

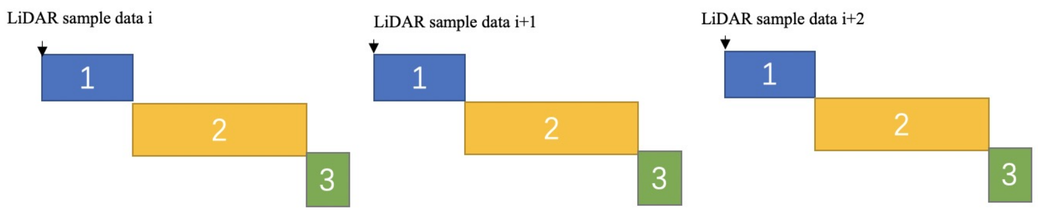

Figure 7.1: Sequential architecture of the obstacle detection algorithm running on a single node

In Chapter 6, the execution time of both implementations is measured when *queue_size* is set to 1. The scanning frequency of the LiDAR is set to 5.5 Hz. Under the same conditions, we increase the *queue_size* in the pipelined version of the algorithm to 10, 100. Table 7.1 and table 7.2 shows the time measurement when *queue_size* are 10 and 100. Comparing the results to the table 6.3, we can see that the execution time of each node, the communication overhead between nodes, and the end-to-end response time are not affected by the change of *queue_size*. Since the publishing rate of the LiDAR is set low, the nodes in the pipelined implementation process the messages immediately when they arrive. However, we can expect that if the messages arrive faster and are buffered in the queue, when the *queue_size* is also large, it might lead to the communication between nodes blocking for an indefinite amount of time.

Node		Time (ms)
Node 1	execution time	0.1364
Communication overhead between Node 1 and Node 2		0.1686
Node 2	execution time	0.2178
Communication overhead between Node 2 and Node 3		0.2183
Node 3	execution time	0.0212
End-to-end response time		0.7622

Table 7.1: Execution time in the pipelined algorithm when $queue\_size = 10$

Node		Time (ms)
Node 1	execution time	0.1102
Communication overhead between Node 1 and Node 2		0.2443
Node 2	execution time	0.2660
Communication overhead between Node 2 and Node 3		0.2079
Node 3	execution time	0.0242
End-to-end response time		0.8526

Table 7.2: Execution time in the pipelined algorithm when $queue\_size = 100$

Figure 7.2 shows the pipeline architecture of the obstacle detection algorithm running on three nodes. In the figure, we have three LiDAR data sample frames from i to $i + 2$. Each LiDAR sample data frame should experience the three stages to finish a obstacle detection job. The communication overhead between nodes is considered.

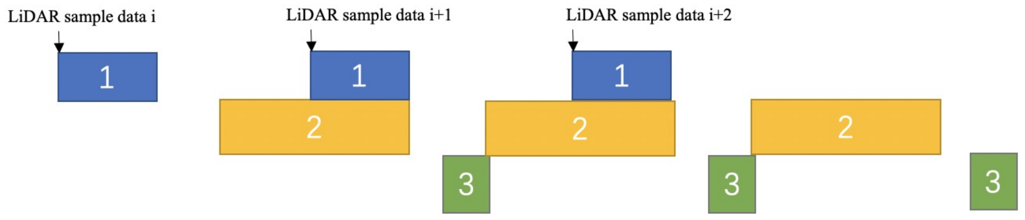

Figure 7.2: Pipelined architecture of the obstacle detection algorithm running on three nodes

7.3 Throughput

Here we compare the throughput of both implementations, we measure the throughput by the number of frames of LiDAR the implementations can process in two minutes. We set a fixed queue size in the implementation to 10, and vary the LiDAR scanning frequency within the range from 5 Hz to 15 Hz. Then, we calculate the number of frames that can be executed for both implementations in two minutes. As shown in Figure 7.3, we can see that as the scanning frequency of the LiDAR increases, the number of processed frames in both implementations increases. Because the pipelined version has a longer response time, the frames it can process are generally fewer than the normal implementation. When there are no data overhead and message serialization, the pipeline's run time is determined by the run time of individual nodes, which can result in a positive pipeline speedup. Thus, the data overhead when transferring data between nodes is a bottleneck. This disadvantage can be improved by deploying the algorithm on a multi-core system and intra-process communication in ROS.

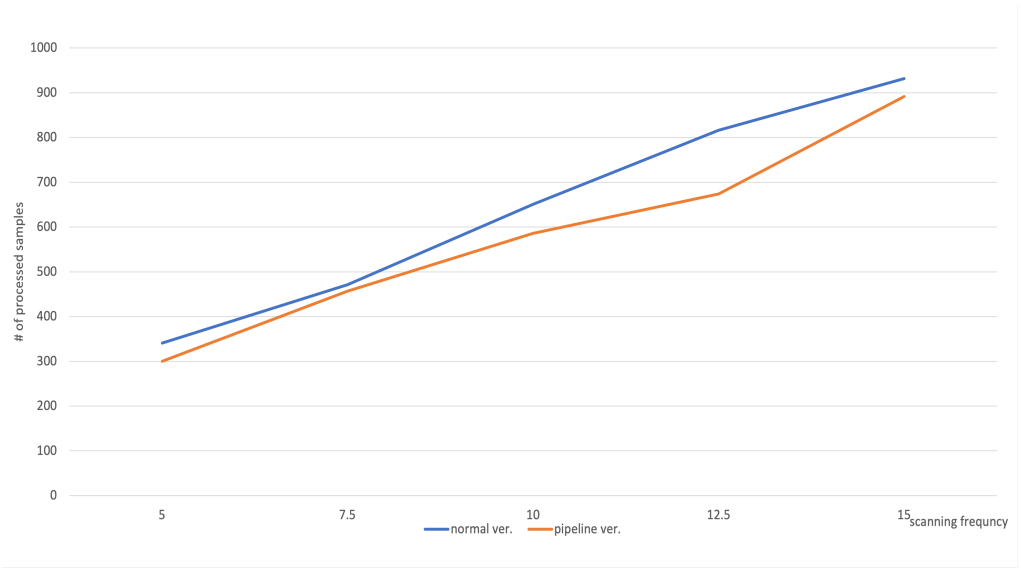

Figure 7.3: Throughput in the normal and pipelined version of implementation of the obstacle detection algorithm

Chapter 8

Conclusion and Future works

8.1 Conclusion

In this work, we first discuss the obstacle detection algorithm based on 2D LiDAR in a ROS-based system. The study of obstacle detection algorithms using 2D LidAR and the implementation of the selected algorithm are introduced. Obstacles existing in the robot workspace are represented by point sets against their outlines, and their information is initialized and updated via the raw laser measurement points. The algorithm follows three main steps: pre-processing, splitting and clustering, classification of consequent measurements.

Then, we propose a pipeline implementation for the obstacle detection algorithm, which allows splitting complex processing tasks into separate processes and aims to reduce run time for 2D laser point cloud input within the ROS framework.

Finally, we evaluate the ROS message communication overhead between nodes and analyze the latency and through the ROS system. Experiments performed in the simulation indicate that the proposed method with the node configuration and separate processes for each stage shows that the time guarantee for message transmission is a bottleneck of the performance and influences timing negatively. The total transmission overhead takes up to around 50% of the end-to-end response time in the pipelined implementation. The end-to-end latency increases 84% comparing to the execution time of the sequential implementation.

8.2 Future works

There are some future works for improvement and further research after the completion of the thesis.

In our work, we pipeline the obstacle detection algorithm on three nodes, which entails latency to the system. The experiments show that the main disadvantage is in the area of real-time performance and message transmission overhead. The node configuration and separate processes yield limited and somewhat unpredictable results. Thus, this work can be improved by using shared memory or intra-process communication to reduce latency significantly.

Message serialization and deserialization occur only during inter-process communication, which is the data exchange between nodes. Nodelets, i.e. intra-process communication, can be used for each stage of the obstacle detection algorithm, which has the benefit of omitting serialization overhead and highly reasonable timings. It is also suitable for large data packets transmission. Nodelets enable classes to be dynamically loaded to the same node but behave as independent processes. This realizes zero-copy transmission between publisher and subscriber nodelets, which means the pointers to data are passed between nodelets [23]. The difference between nodes and nodelets is that each node is assigned to a separate OS process, while nodelets can be grouped together within a single OS process with resource sharing.

Furthermore, ROS is not real-time capable due to its current system architecture, since real-time capabilities for inter-process and inter-machine communication is one of the main features of ROS 2, it can address the issues in our experiments and provide a system that is closer to the status quo of real-time processing [21].

Bibliography

[1] Gazebo ros pkgs. `https://wiki.ros.org/gazebo_ros_pkgs`. Accessed: 2021-07-17. 9

[2] Ros urdf. `https://wiki.ros.org/urdf`. Accessed: 2021-07-16. 9

[3] Rplidar a2. `https://www.slamtec.com/en/Lidar/A2Spec`. Accessed: 2021-07-17. 10

[4] M. B. Alatise and G. P. Hancke. A review on challenges of autonomous mobile robot and sensor fusion methods. *IEEE Access*, 8:39830–39846, 2020. 1

[5] A. J. Alvares, Gabriel F. Andriolli, Paulo R. C. Dutra, Márcio Sousa, and J. Ferreira. A navigation and path planning system for the nomad xr4000 mobile robot with remote web monitoring. 2003. 4

[6] Kai O. Arras, Oscar Martinez Mozos, and Wolfram Burgard. Using boosted features for the detection of people in 2d range data. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3402–3407, 2007. 5

[7] Giuseppe Beccari, Stefano Caselli, Monica Reggiani, and Francesco Zanichelli. Rate modulation of soft real-time tasks in autonomous robot control systems. In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*, pages 21–28. IEEE, 1999. 6

[8] Angelo Nikko Catapang and Manuel Ramos. Obstacle detection using a 2d lidar system for an autonomous vehicle. In *2016 6th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 441–445. IEEE, 2016. 2

[9] Baifan Chen, Zixing Cai, Zheng Xiao, Jinxia Yu, and Limei Liu. Real-time detection of dynamic obstacle using laser radar. In *2008 The 9th International Conference for Young Computer Scientists*, pages 1728–1732, 2008. 4

[10] Yun-Jian Cheng, Wenge Qiu, and Jin Lei. Automatic extraction of tunnel lining cross-sections from terrestrial laser scanning point clouds. *Sensors*, 16(10), 2016. 16

[11] Serhan Coşar and Nicola Bellotto. Human re-identification with a robot thermal camera using entropy-based sampling. *Journal of Intelligent Robotic Systems*, 98, 04 2020. 5

[12] D. D'Auria and F. Persia. A collaborative robotic cyber physical system for surgery applications. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 79–83, 2017. 2

[13] Mirco De Marchi, Francesco Lumpp, Enrico Martini, Michele Boldo, Stefano Aldegheri, and Nicola Bombieri. Efficient ros-compliant cpu-igpu communication on embedded platforms. *Journal of Low Power Electronics and Applications*, 11(2), 2021. 8

[14] Martin Dekan, Duchoň František, Babinec Andrej, Rodina Jozef, Rau Dávid, and Music Josip. Moving obstacles detection based on laser range finder measurements. *International Journal of Advanced Robotic Systems*, 15(1):1729881417748132, 2018. 5

[15] Huixu Dong, Ching-Yen Weng, Chuangqiang Guo, Haoyong Yu, and I-Ming Chen. Real-time avoidance strategy of dynamic obstacles via half model-free detection and tracking with 2d lidar for mobile robots. *IEEE/ASME Transactions on Mechatronics*, pages 1–1, 2020. vii, 5, 13, 14, 15, 16, 17, 18, 19

[16] Carol Fairchild and Thomas L. Harman. *ROS Robotics By Example*. Packt Publishing, 2016. 7

[17] Carlos Fernández, Vidal Moreno, Belen Curto, and J Andres Vicente. Clustering and line detection in laser range measurements. *Robotics and Autonomous Systems*, 58(5):720–726, 2010. 3

[18] S. Gatesichapakorn, J. Takamatsu, and M. Ruchanurucks. Ros based autonomous mobile robot navigation using 2d lidar and rgb-d camera. In *2019 First International Symposium on Instrumentation, Control, Artificial Intelligence, and Robotics (ICA-SYMP)*, pages 151–154, 2019. 7

[19] D. Ghorpade, A. D. Thakare, and S. Doiphode. Obstacle detection and avoidance algorithm for autonomous mobile robot using 2d lidar. In *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, pages 1–6, 2017. 10

[20] Gene H Golub and Charles F Van Loan. An analysis of the total least squares problem. *SIAM journal on numerical analysis*, 17(6):883–893, 1980. 17

[21] André-Marcel Hellmund, Sascha Wirges, Ömer Şahin Taş, Claudio Bandera, and Niels Ole Salscheider. Robot operating system: A modular software framework for automated driving. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1564–1570, 2016. 33

[22] Ryuki Higuchi and Yasutaka Fujimoto. Road and intersection detection using convolutional neural network. In *2020 IEEE 16th International Workshop on Advanced Motion Control (AMC)*, pages 363–366, 2020. 4

[23] Anjani Josyula, Bhaskar Anand, and P. Rajalakshmi. Fast object segmentation pipeline for point clouds using robot operating system. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 915–919, 2019. 5, 32

[24] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004. vii, 9

[25] Timm Linder and Kai O. Arras. *People Detection, Tracking and Visualization Using ROS on a Mobile Service Robot*, pages 187–213. Springer International Publishing, Cham, 2016. 5

[26] R. MacLachlan and C. Mertz. Tracking of moving objects from a moving vehicle using a scanning laser rangefinder. In *2006 IEEE Intelligent Transportation Systems Conference*, pages 301–306, 2006. 5

[27] Christoph Mertz, Luis Navarro-Serment, Robert Maclachlan, Paul Rybski, Aaron Steinfeld, Arne Suppé, Christopher Urmson, Nicolas Vandapel, Martial Hebert, Chuck Thorpe, and David Duggins. Moving object detection with laser scanners. *J. Field Robot.*, 30:17–43, 02 2013. 5

[28] Gyula Mester. „applications of mobile robots”. In *Proceedings of the 7th International Conference of Food Science, Szeged*, pages 1–5, 2006. vii, 1

[29] Mike1024. Lidar (appearance based on sick lms 219) with a single beam scanned in one axis. `https://upload.wikimedia.org/wikipedia/commons/c/c0/LIDAR-scanned-SICK-LMS-animation.gif`. vii, 11

[30] Rasoul Mojtahedzadeh. *Robot Obstacle Avoidance using the Kinect*. PhD thesis, 08 2011. vii, 4

[31] Taketoshi Mori, Takahiro Sato, Hiroshi Noguchi, Masamichi Shimosaka, Rui Fukui, and Tomomasa Sato. Moving objects detection and classification based on trajectories of lrf scan data on a grid map. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2606–2611, 2010. 4

[32] Viet Nguyen, Agostino Martinelli, Nicola Tomatis, and Roland Siegwart. A comparison of line extraction algorithms using 2d laser rangefinder for indoor mobile robotics. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1929–1934. IEEE, 2005. 3

[33] Arbnor Pajaziti and Petrit Avdullahu. Slam–map building and navigation via ros. *International Journal of Intelligent Systems and Applications in Engineering*, 2(4):71–75, 2014. 7

[34] Yan Peng, Dong Qu, Yuxuan Zhong, Shaorong Xie, Jun Luo, and Jason Gu. The obstacle detection and obstacle avoidance algorithm based on 2-d lidar. In *2015 IEEE International Conference on Information and Automation*, pages 1648–1653, 2015. 13, 14, 16, 17, 18

[35] Plamen Petrov, Veska Georgieva, Stiliyan Nikolov, and Antonia Mihaylova. Real-time laser obstacle detection system for autonomous mobile robot navigation. In *2019 X National Conference with International Participation (ELECTRONICA)*, pages 1–4, 2019. 3

[36] Anna Petrovskaya and Sebastian Thrun. Model based vehicle detection and tracking for autonomous urban driving. *Autonomous Robots*, 26(2):123–139, 2009. 5

[37] Gazebo plugins in ROS. Tutorial: Using gazebo plugins with ros. `https://gazebosim.org/tutorials?tut=ros_gzplugins`. Accessed: 2021-07-17. 9

[38] Cristiano Premebida and Urbano Nunes. Segmentation and geometric primitives extraction from 2d laser range data for mobile robot applications. *Robotica*, 2005:17–25, 2005. 3

[39] Mateusz Przybyła. Detection and tracking of 2d geometric obstacles from lrf data. In *2017 11th International Workshop on Robot Motion and Control (RoMoCo)*, pages 135–141. IEEE, 2017. vii, vii, vii, vii, 3, 13, 15, 16, 17, 18, 19

[40] W. Qian, Z. Xia, J. Xiong, Y. Gan, Y. Guo, S. Weng, H. Deng, Y. Hu, and J. Zhang. Manipulation task simulation using ros and gazebo. In *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*, pages 2594–2598, 2014. 8

[41] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009. 7

[42] Ankit A Ravankar, Abhijeet Ravankar, Takanori Emaru, and Yukinori Kobayashi. Multi-robot mapping and navigation using topological features. In *Multidisciplinary Digital Publishing Institute Proceedings*, volume 42, page 68, 2019. 2

[43] M. Ribeiro. Obstacle avoidance. 2005. 2

[44] Zandra B Rivera, Marco C De Simone, and Domenico Guida. Unmanned ground vehicle modelling in gazebo/ros-based environments. *Machines*, 7(2):42, 2019. 8, 9, 11

[45] Daniel Oñoro Rubio, Artem Lenskiy, and Jee-Hwan Ryu. Connected components for a fast and robust 2d lidar data segmentation. In *2013 7th Asia Modelling Symposium*, pages 160–165. IEEE, 2013. 15

[46] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 1(2):26, 2016. 21

[47] Maxim Sokolov, Roman Lavrenov, Aidar Gabdullin, Ilya Afanasyev, and Evgeni Magid. 3d modelling and simulation of a crawler robot in ros/gazebo. In *Proceedings of the 4th International Conference on Control, Mechatronics and Automation*, pages 61–65, 2016. 9

[48] Danilo Tardioli, Ramviyas Parasuraman, and Petter Ögren. Pound: A multi-master ros node for reducing delay and jitter in wireless multi-robot networks. *Robotics and Autonomous Systems*, 111:73–87, 2019. 27

[49] Victor Vaquero, Ely Repiso, and Alberto Sanfeliu. Robust and real-time detection and tracking of moving objects with minimum 2d lidar information to advance autonomous cargo handling in ports. *Sensors*, 19(1):107, 2019. 3, 4

[50] Trung-Dung Vu, Olivier Aycard, and Nils Appenrodt. Online localization and mapping with moving object tracking in dynamic outdoor environments. In *2007 IEEE Intelligent Vehicles Symposium*, pages 190–195. IEEE, 2007. 4

[51] Shiyong Wang, J. Wan, D. Zhang, D. Li, and Chunhua Zhang. Towards smart factory for industry 4.0: a self-organized multi-agent system with big data based feedback and coordination. *Comput. Networks*, 101:158–168, 2016. 1

[52] Eric N Willcox III. Forward perception using a 2d lidar on the highway for intelligent transportation. Thesis, Worcester Polytechnic Institute, 2016. 4

[53] Yilu Zhao and Xiong Chen. Prediction-based geometric feature extraction for 2d laser scanner. *Robotics and Autonomous Systems*, 59(6):402–409, 2011. 5

[54] Lanxiang Zheng, Ping Zhang, Jia Tan, and Fang Li. The obstacle detection method of uav based on 2d lidar. *IEEE Access*, 7:163437–163448, 2019. 3