Enhancing the Accuracy and Robustness of LiDAR Based Simultaneous Localisation and Mapping

Weichen WEI

Supervisor: Prof. B. Shirinzadeh

Department of Engineering

Monash University

A thesis submitted for the degree of

Doctor of Philosophy

February 2021

Copyright Notice

Abstract

In the past decades, the robotic system autonomy has became essential in many research and industrial fields. Studies have been focused on the simultaneous mapping and localisation (SLAM) of robotic systems in unknown environments. The application of LiDAR-based SLAM approaches has been challenged by the flexible motions of robotic systems, the specifications of LiDARs and the dynamic changes of the environments. To meet this demand, further investigation of LiDAR-based SLAM accuracy, with a focus on the system robustness, is required.

This work has studied two aspects of the LiDAR SLAM systems: the mapping and localisation error accumulation in SLAM processes; and pose estimation through feature points extraction. Emphases are placed on the study of system failure recovery and odometry stabilisation in 2D and 3D LiDAR-based SLAM systems.

State-of-the-art 2D and 3D LiDAR-based SLAM approaches are investigated, with a focus on their error accumulations in mapping and localisation. Conventional SLAM approaches generate low-quality pose estimation during temporary system degradation. Errors in the system pose are accumulated and eventually cause mapping failure. In this work, studies were conducted to evaluate the system degradation scenarios and the reasons for pose estimation errors. Modelling of the mapping error was studied, as well as a method to recover system states which utilises a supplementary trajectory. The iterative trajectory matching (ITM) approach is presented, which applies iterative-closest-points (ICP) algorithm to trajectories to model their geometrical relationship, and thus identify the accumulated drift of a SLAM system. Experiments were conducted using 2D SLAM algorithms to evaluate the efficiency of the ITM algorithm. Then the study was extended using a 3D LiDAR-based SLAM algorithm to validate its effectiveness.

Through the combination of the 2D and 3D LiDAR-based SLAM algorithms, a dual-LiDAR

2D-3D mixed SLAM approach was developed in this study to improve the efficiency of a Solid-State-LiDAR (SSL). Conventional SSL SLAM approaches are limited by the narrow Field-Of-View (FOV) of the SSL. In this work, the described dual-LiDAR SLAM design is investigated to enhance the feature points extraction of a SLAM system. Significant robustness and stability were demonstrated through the experiments using the developed algorithm in various scenarios.

Using the developed methodologies, novel LiDAR SLAM systems were designed, developed and characterised, including two 2D handheld LiDAR SLAM devices and a 3D SLAM unmanned ground vehicle (UGV) that facilitate the 2D-3D mixed LiDAR mapping unit. System evaluations were performed through mapping tasks in small to medium size indoor and outdoor scenarios which demonstrated enhanced robustness, and hence increased accuracy of LiDAR-based SLAM algorithms. Through the development of these LiDAR-based SLAM systems, the works contained in this thesis have expanded the ability of users to reliably and precisely perform SLAM tasks.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Weichen WEI

February 2021

Acknowledgements

Thank you to the many people who supported this work, both in academic contributions, and personal support.

To my supervisor, Bijan Shirinzadeh, thank you for instigating and developing the field of research that has provided endless mental stimulation and problems to solve. Thank you for your guidance and supervision throughout my research study. Thank you for the time you've invested in developing my skills as a researcher.

To all my dear friends from the RMRL, particularly Ali, Tilok, Ammar, David, Shen, Dr. Rohan, Dr. Josh and Armin. They have provided me with intellectual support, technical guidance, and motivation. Thank you for all the accompanies and conversations. Those late nights we spent in the lab and those extended lunch sessions have formed a substantial component of my university life. Special thanks to Ali and Shen, the strongest boys in the lab and the smartest guys in the gym. I will miss those gym days, especially with the COVID-19 and the lockdowns.

Thank you immensely to my family, who have provided me with their unconditional love. Thanks to my parents, who always support me in any way possible. Your commitment to the family and career have motivated me to keep making improvements in myself. Special thanks to my wife, Yang Jiaorao, who always reminds me that running a food truck is not a bad idea if I get exhausted with the academic work, which I doubt.

To my journey here in Australia, which shaped me into what I am today. Thanks for everyone I came across and all the help I have received. Thanks to my friends in Canberra, Sydney and Melbourne. It would be a difficult time for me as a young international student without you. After over a decade, the memory from that time is still exciting to me.

Finally, I would like to acknowledge the primary funder of this research, the Australian

Government, who supported me personally through an Australian Government Research Training Program (RTP) Scholarship, and further funded a significant proportion of the research at RMRL through Australian Research Council (ARC) Linkage Infrastructure, Equipment and Facilities (LIEF), and ARC Discovery grants. Similarly, I would never have commenced this work without the financial support of the Monash University through an Graduate Research Completion Award (GRCA).

First author journal articles resulting from thesis

[J1] W. Wei, B. Shirinzadeh, R. Nowell, M. Ghafarian, M. M. A. Ammar and T. Shen, "Enhancing Solid State LiDAR Mapping with a 2D Spinning LiDAR in Urban Scenario SLAM On Ground Vehicles," *Sensors*, Submitted, 2021.

[J2] W. Wei, B. Shirinzadeh, R. Nowell and M. Ghafarian, "Posture and Map Restoration in SLAM Using Trajectory Information," *Journal of Field Robotics*, Submitted, 2021.

First author conference publications resulting from thesis

[C1] W. Wei, B. Shirinzadeh, S. Esakkiappan, M. Ghafarian and A. Al-Jodah, "Orientation Correction for Hector SLAM at Starting Stage," *2019 7th International Conference on Robot Intelligence Technology and Applications, RiTA 2019*, pp. 125–129, 2019, ISSN: 2340-9711. DOI: 10.1109/RITAPP.2019.8932722.

[C2] W. Wei, B. Shirinzadeh, M. Ghafarian, S. Esakkiappan and T. Shen, "Hector SLAM with ICP trajectory matching," *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, vol. 2020-July, no. 1, pp. 1971–1976, 2020. DOI: 10.1109/AIM43001.2020.9158946.

[C3] W. Wei, B. Shirinzadeh, R. Nowell, M. Ghafarian, M. M. A. Ammar and T. Shen, "Multi-LiDAR LOAM for Improving Mapping Robustness of Narrow Field-of-View LiDAR on Ground Vehicles," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2021, Submitted.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Simultaneous localisation and mapping (SLAM) is a computation problem that has been studied for more than thirty years. Recently, the technology has been applied to many rapid growing industries, such as autonomous systems and augmented reality (AR) devices. As the term stated, the problem a SLAM system is trying to address can be divided into two parts. The first part being accurately mapping the environment surrounding the robotic system during its movement. The second part is correctly locating the robotic system while it is travelling through the environment. In other words, a SLAM system is a system which uses previous recorded map and system state to estimate the current surrounding map and system state.

Traditionally, the mapping and localisation of a robotic system have relied on the pre-knowledge about the environment. For example, the Global Positioning System (GPS) requires existing modelling about the earth coordinate system. Similarly, the inertial measurement unit (IMU) have relied on knowledge about the earth magnetic field. With the location information about the antenna tower, some approaches use the Doppler effect to estimate the distance between the sensor and the signal source. In summary, all these traditional approaches have a common requirement for existing knowledge about the environment, which significantly restricts the application of the robotic system.

Evolving from the traditional mapping and localisation methods, the recent development of SLAM approaches emphasises the ability of mapping and localising in an unknown environ-

Table 1.1 Different kinds of imaging sensors with their features listed

Sensor/Impact	Lighting	Colour	Depth	Range	Accuracy	Cost
Spinning-LiDAR	Low	Low	Yes	High	High	High
Solid-State-LiDAR	Low	Low	Yes	High	High	Low
Camera	High	High	No	Medium	Low	Low
Depth-Camera	High	Low	Yes	Low	Medium	Medium
mmWave	Low	No	Yes	High	Medium	Medium
Sonar	Low	No	Yes	Medium	Low	Low

ment with no existing knowledge. To achieve this, modern SLAM approaches use a so called scan-matching process which scan the environment and match the current measurement with previous measurements, thus estimate the transformation from its previous system state to the current system state. The scan-matching process of a SLAM method is generally based on the measurements from the primary imaging module. The choice of imaging sensors defines the application of the SLAM approach. The most common imaging sensors used in SLAM methods are listed in Table 1.1, with their advantages and disadvantages provided.

However, only a imaging sensor alone is not enough to provide accurate mapping results. The performance of a SLAM method depends on four fundamental components:

- **The chassis of the moving platform**, including its performance, driving mode and control algorithm. As the requirements of a SLAM approach varies, the implementation can be air-based, ground-based or underwater-based. Different chassis design defines the method of sensor installation and the platform moving pattern.

- **The sensors equipped**, such as IMU and GPS. A SLAM approach does not necessarily depend on one sensor. Instead, sensor fusion is a common approach for high-performance SLAM systems. The performance of a SLAM implementation is related to the cooperation of all sensors installed in the system.

- **The software system**, for example, an embedded system. Internal communication and computation of a SLAM device with multiple components is critical to its mapping performance. A high-performance SLAM device requires the ability of real-time processing and high-speed data transmission.

- **The mapping algorithm**, which defines the scan-matching process of the SLAM method. With all the sensors and computers installed, the core of a SLAM method is the mapping algorithm that takes the sensor measurements and calculates system state estimations.

Developed according to the above four major aspects, the performance of a SLAM system is related to a range of components. Studies addressing SLAM performance under different scenarios have been extensively discussed in recent years. The design, modelling and development of advanced three to six degree of freedom (DOF) SLAM algorithms have formed the basis of the recent investigations in SLAM methodologies.

While a SLAM device can use cameras, LiDARs and other sensors as its primary sensing module, LiDAR offers an outstanding balance between accuracy, scan rate, robustness and cost among all other vision sensors. The systems adopted LiDAR SLAM algorithms can generate accurate 2D and 3D maps and localise the robots in millimetre-level accuracy. LiDAR sensors use laser beams as the measurement technique, which is naturally robust to the environmental light. Additionally, LiDARs directly provides distance information in the measurements, thus have lower computational cost than some other vision sensors.

Unfortunately, the studies of LiDAR-based SLAM algorithms do not readily extend to dynamic scenarios that are more related to real-world applications. The foundation of the state-of-the-art LiDAR SLAM approach is based on the assumption of a smooth and continuous system motion in a feature-rich environment. Rapid movement or motions on unsupported DOF can cause significant errors in the system state estimation process. To ensure the accuracy of the mapping process, many approaches are running in a controlled environment with strictly limited applications.

As the mapping environment required by SLAM applications expanded from room-size indoor spaces to city-scale indoor-outdoor 3D maps, the ability to handle the mixed environment in SLAM approaches and correct accumulated mapping errors will become the next milestone in the study of the SLAM techniques. Thus, the methods enhancing the robustness of SLAM approaches are required. The scan-matching algorithms for single LiDAR-based SLAM approaches and multi-sensors based SLAM approaches should be investigated to systematically improve the performance of a SLAM method under challenging environments.

1.1 Research aims

The objective of this research is to investigate and identify the requirements for high-robustness LiDAR-based SLAM, and thus design and develop methodologies to enhance the stability of the mapping and localisation process the LiDAR-based mapping devices in complex scenarios. The specific research aims of this study included:

- To investigate the instability of 2D and 3D LiDAR-based SLAM approaches during their initialisation stages, with a focus on the pose estimation errors due to the incomplete map recorded by the algorithm, thus establishing methodologies which identify, estimate and correct localisation errors accumulated in the starting stage of a LiDAR-based SLAM approach.

- To identify the sources of mapping error accumulation in mainstream LiDAR SLAM algorithms, including errors caused by kidnapping (unexpected movements), rollover, sharp turns and environment change, and thus establish methodologies and algorithms to enable mapping and localisation error correction for LiDAR SLAM methods.

- To examine and characterise the scan-matching process of 3D LiDAR SLAM algorithms, with the consideration of different laser scan patterns and field-of-view (FOV) from various LiDAR models, including feature selection method, motion blur correction, pose estimation and map updating.

- To formulate strategies to systematically enhance 3D LiDAR SLAM systems, including possible improvements in sensor layout and mapping algorithms, and thus establish mapping methodologies and develop a 3D LiDAR-based SLAM system that demonstrates robustness in challenging SLAM scenarios.

1.2 Principle contributions

The expanding implementation of SLAM methods in various terrain scenarios will almost certainly create a strong demand for more robust SLAM approaches in terms of both mapping quality and stability. The critical challenges faced by the current SLAM approaches include the system degradation in the rapid motion, complex scanning environment and error accumulation during the life span of the SLAM process. This thesis described multiple approaches to improve the quality of a LiDAR SLAM process when facing SLAM system degradation. A novel multi-LiDAR SLAM approach is also presented in this work which uses a 2D LiDAR to enhance the 3D LiDAR mapping system to capture a wider range of feature points than previously studied approaches. The principal contributions of the research described within this thesis can be summarised as follows:

1. An approach to identify and model the position drift during the initialisation stage of the Hector SLAM, which enables pose correction in 2D SLAM approaches based on a grid map. The developed approach is analysed and tested, with software provided for implementation, and experimental validation.

2. A timestamped grid map structure is proposed, which allows time-based map selection and correction. The proposed method is tested with a detail analysis focus on its performance, operation methods and real-time efficiency.

3. A Spherical-linear-interpolation-based (Slerp) 3D SLAM pose correction approach is described enabling smooth and real-time pose correction. A 3D LiDAR SLAM approach is developed based on the Slerp pose correction method.

4. Two approaches to estimate and correct errors accumulated in the SLAM states are proposed, with both poses and mapping results improved accordingly. By comparing the SLAM system trajectory and a reference trajectory, the proposed method can identify errors recorded during the process, thus corrected by the developed method. The proposed approaches are applied to both 2D grid map based SLAM approaches and 3D point cloud

based SLAM approaches which demonstrate its ability to be extended to most existing SLAM approaches.

5. A multi-LiDAR SLAM design, aiming at enhancing the FOV of traditional LiDAR SLAM approaches, is designed and developed. The mapping module features low cost and large Field-of-View (FOV), enabling a SLAM algorithm to extract more feature points in a challenging environment.

6. A 3-DOF interpolated 6-DOF LiDAR odometry has been developed, which allows stable odometry updates in challenging scenarios. The performance of the developed odometry is investigated. Compared with existing approaches, the proposed method is able to reduce localisation errors by interpolating odometry updates.

7. A 2D-3D mixed LiDAR SLAM approach is described, with details in both software and hardware. The developed approach demonstrated its application through testing scenarios. Through this novel approach, the SLAM system demonstrates significant robustness in challenging environments, includes sharp corners and long corridors. Consequently, the stability of the SLAM process in a mixed environment is enhanced.

1.3 Thesis organization

The organisation of this thesis follows the time frame and workflow of the development of the project. In particular, this thesis provides a background review of the studied areas, the description of the proposed methods. The conclusions and future works are discussed in the final part of the thesis.

- Chapter 2 presents a review of the background and related works for this study. It describes the development of current SLAM approaches, with a focus on the relationship between sensor, algorithm and mapping quality. The mapping performance of SLAM approaches in different terrain scenarios is extensively reviewed, along with the methodology of the multi-terrain SLAM systems currently under research. The review covers the environmental challenges faced by SLAM approaches and some of the existing approaches to improve mapping performance in complex terrain scenarios, with comparison in their stability, efficiency and flexibility.

- Chapter 3 discusses the cause of mapping error during the initialisation stage, with detailed examples based on Hector SLAM. A method of observing two separate sets of pose estimation from a single SLAM approach is described, along with the proposed system pose correction method. A SLAM system is constructed by combining the proposed method into Hector SLAM. The mapping performance of the constructed SLAM system is evaluated by comparing its mapping results with the original Hector SLAM.

- Chapter 4 details the design, implementation and evaluation of a method for the correction of real-time system state and mapping results. The study included in this chapter is based on the work presented in Chapter 3, with extensive investigations in mapping correction and pose recovery. These include studies of the design, analysis and experimental works to establish a map correction mechanism, a 3 DOF SLAM state correction approach and its extensive application in 6 DOF SLAM system.

- Chapter 5 outlines the development of a 2D reinforced 3D LiDAR SLAM system for complex urban mapping scenarios. Experimental work is performed to evaluate the performance of the proposed 2D reinforced 3D mapping model. Finally, a ground vehicle based robust mapping platform is introduced with the integration of the proposed SLAM approach described in Chapters 3 and 4.

- Chapter 6 summarises the progress and achievements of the research studies, and identifies the key contributions in each part of the project. The future direction of the project is also investigated, with the consideration of the research gap, possible approaches and expected outcomes.

Chapter 2

Background

2.1 SLAM systems and improving mapping and localisation performance

The problem a SLAM is trying to solve is to perform mapping and localisation at the same time. However, these two questions are often mutually exclusive. Many developed SLAM systems nowadays could provide high-quality mapping results in different terrain scenarios. This study is aiming at improving the stability and robustness of a LiDAR SLAM system. In this chapter, we review some of the current mainstream SLAM algorithms to identify their limitations. While comparing the performance of the state-of-the-art SLAM algorithms, the review also covers some related works focusing on the robustness of a mapping system, including sensor fusion, map correction and environment classification techniques.

2.2 State-of-the-art SLAM approaches

The rapid development of SLAM technologies has dramatically changed the performance of the mapping systems in recent years. Form 2D localisation, the technology quickly developed to 3D mapping with a wide range of sensors available. The key problem a SLAM system addresses is stated in Equation (2.1) where at time k the SLAM system state vector $\mathbf{x}_k$ can be described using the control input u_k, robot pose $x_{v_{k-1}}$ and map m.

$$\begin{aligned}\mathbf{x}_k &= \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k)\\ &= \begin{bmatrix} \mathbf{f}_v\left(\mathbf{x}_{v_{k-1}}, \mathbf{u}_k\right) \\ \mathbf{m} \end{bmatrix}\end{aligned} \tag{2.1}$$

This chapter reviews some of the SLAM approaches and algorithms that related with the research question proposed by this thesis. Since this study focuses on LiDAR sensors, the review mainly covers state-of-the-art 2D and 3D LiDAR SLAM methodologies. Some related camera-based approaches are also included in the review for comparison purposes.

2.2.1 Two-dimensional LiDAR SLAM

2D LiDAR SLAM approaches are well-adopted solutions for the Unmanned Ground Vehicle (UGV) systems on the market. They offer reliable and affordable solutions for most of the ground moving platforms. As well-established SLAM methodologies, 2D LiDAR SLAM approaches are based on LiDAR sensors which use a rotational laser beam to sample a 2D fraction of the surrounding environment. Since 2D SLAM can only generate 2D maps, these systems prioritise localisation over the mapping details.

2.2.1.1 Graph-based scan matching algorithms

Iterative Closest Point (ICP) is a method initially proposed for computer graphic systems [3] [4] [5] [6]. The main idea of ICP is to find the rotation and translation between two sets of points which minimise the their Euclidean distance. The method iteratively calculates the distance between each pair of points in the point sets. ICP is a gradient-based method. With all points in two point sets paired, it uses gradient descent, Gauss-Newton or Levenberg–Marquardt algorithms to find the best transformation information.

Consider q_i and p_i are a pair of points in two registered point sets. the cost function of a point-to-point ICP is

$$\mathrm{E(R,T)} = \mathrm{argmin}_{R,t} \sum_{i=1}^{n} ||Rp_i + T - q_i||^2 \tag{2.2}$$

Where the rotation R and translation T are the targets of the optimisation process.

From Equation (2.2), the ICP method requires the two point sets have the same number of points. Downsampling or upsampling is required to balance the number of points in both groups. However, even every point in the sets is paired, the outliers in the data set significantly affect the accuracy of the algorithm. FAST-ICP improves the performance of the ICP process by filtering point sets and assigning point pairs with different weights [7]. While the original ICP algorithm takes all points into the calculation, FAST-ICP removes some of the points that could slow the speed of convergent. Another attempt to improve the performance of the ICP method included the use of the semantic information of the point set. NICP [8] takes the normal vector of the target points and its curvature into consideration during the calculation. To pair two points, they need to be the closest to each other and have their normal vectors in the same direction. Figure 2.1 illustrates the point distribution according to their curvature.

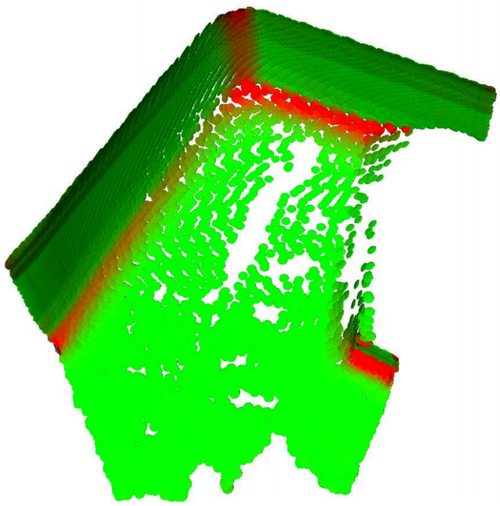


Figure 2.1 NICP point selection, with points colour corresponding to its curvature value. © [2015] IEEE. Reprinted, with permission, from [Jacopo Serafin, NICP: Dense normal based point cloud registration, International Conference on Intelligent Robots and Systems (IROS), September 2015]

Using the point-to-line distance instead of point-to-point could also improve the stability of the optimisation, especially when points are less accurate and have large noise distribution [6]. As seen in Equation (2.3), instead of finding the minimum Euclidean distance between two sets of points, PL-ICP is targeting the minimum distance between the point and its closest plane in the target point set. The comparison between PL-ICP and ICP is shown in Figure 2.2, with original ICP shown in (b) and PL-ICP in (c).

$$\mathrm{E(R,t)} = \mathrm{argmin}_{R,t} \sum_i ((R \cdot x_i + t - y_i) \cdot n_i) \tag{2.3}$$

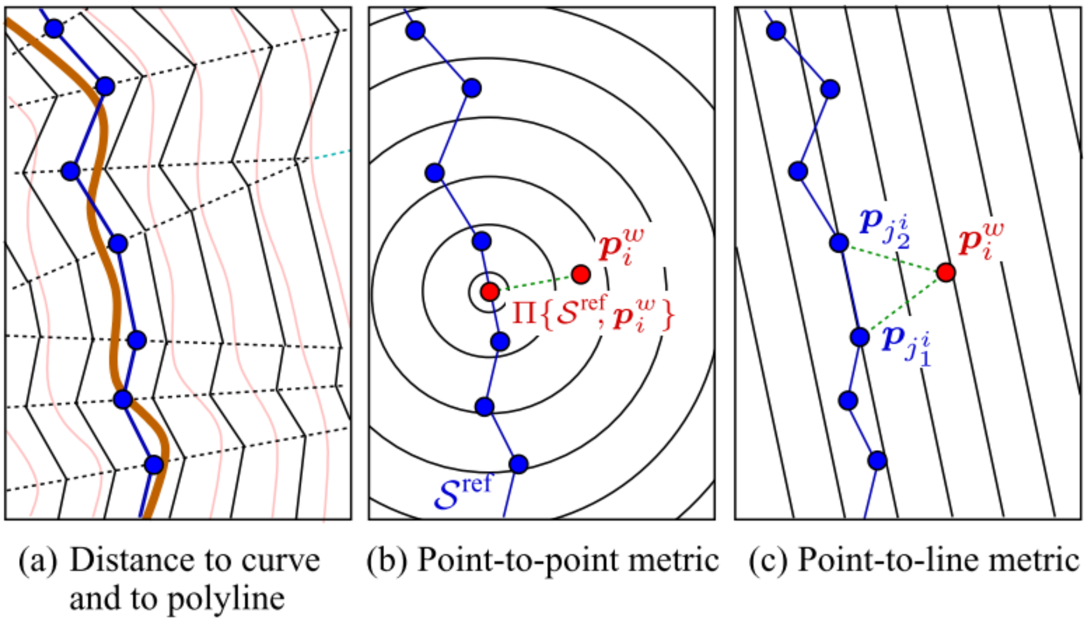


Figure 2.2 Examples of frame-to-frame ICP iteration. © [2013] IEEE. Reprinted, with permission, from [R. Tiar, ICP-SLAM methods implementation on a bi-steerable mobile robot ,2013 IEEE 11th International Workshop of Electronics, Control, Measurement, Signals and their application to Mechatronics, June 2013]

Early studies directly applied ICP algorithm to two consecutive LiDAR scans [9][10][11][12]. However, high-performance LiDAR will produce a large number of points in each scan which notably affects the computationally complexity. Skipping LiDAR frames and using FAST-ICP instead of original ICP could help improve performance [13]. Researchers have also used rotation-invariant descriptors to enhance the robustness of the system further [14].

Hector SLAM is one of the most well-known 2D LiDAR SLAM approach among all ICP based SLAM approaches. Hector SLAM was proposed in [15]. During its first introduction in Robocup 2011, the author used a handheld device with only a 2D LiDAR to map a simulated rescue scenario. The system demonstrated its remarkable performance and still considered a state-of-the-art method at the time of writing this thesis. Hector SLAM can be described in two

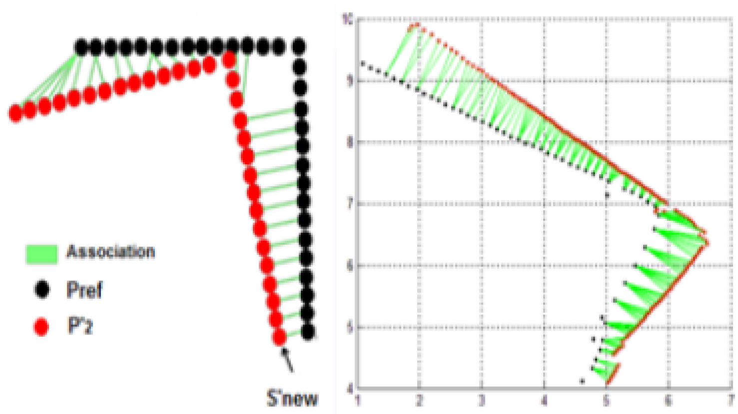


Figure 2.3 ICP and PL-ICP point selection metric comparison. © [2008] IEEE. Reprinted, with permission, from [Andrea Censi, An ICP variant using a point-to-line metric, IEEE International Conference on Robotics and Automation, May 2008]

parts: the first part being its frontend scan matching algorithm, the second being the backend mapping module.

The sensing module of the Hector SLAM uses a 2D LiDAR as the input information. After initialisation, each sweep of 2D LiDAR scan is recorded on the map using bilinear interpolation (Figure 2.4). The system compares the grid cells of the current laser scan with the established map grid cells. If the two sets of cells can be matched, then the translation between them is the pose update between the last system pose and current system pose.

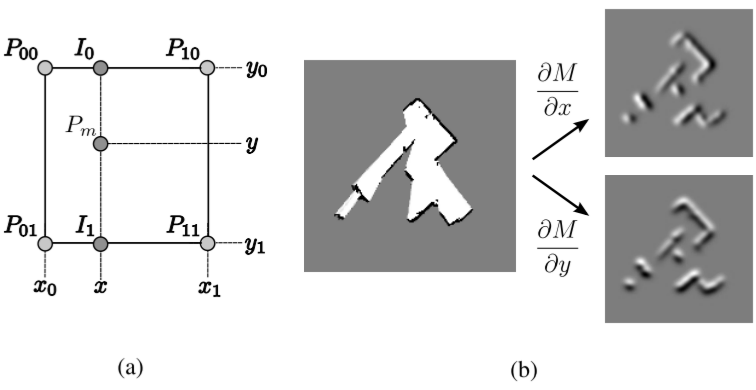


Figure 2.4 Bilinear interpolation in Hector SLAM. © [2011] IEEE. Reprinted, with permission, from [Stefan Kohlbrecher; Oskar von Stryk; Johannes Meyer; Uwe Klingauf, A flexible and scalable SLAM system with full 3D motion estimation, 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics, 1-5 Nov.2011]

he mapping module of the Hector SLAM maintains the mapping results where multi-level resolution is applied to avoid the local minimum problem. The mapping module also features a trajectory server and other maintenance tools.

One of the most important contribution of Hector SLAM is the scan-matching process in the sensing module. After transform fresh laser scan into grid map coordinate system, Hector SLAM uses ICP to iteratively find the most suitable translation and rotation between the map the scan. In the original Hector SLAM, the author used a Gauss-Newton method with rotation and translation as the cost functions to find the current pose update. However, since the corresponding grid cells are bilinear interpolated, it is possible for the algorithm to yield errors if the interpolation results in the middle of four map grids. This problem is improved later using trilinear interpolation [16].

Although Hector SLAM relies on only a LiDAR sensor to work, the system could use an additional IMU to enhance its pose estimation. Fuse the IMU reading to a Kalman filter helps the systems to handle more complex and rapid movement. Additionally, the IMU sensor also provides information on z-axis. The z-axis motion could be used to build 3D mapping with only a 2D LiDAR.

2.2.1.2 Probabilistic SLAM algorithms

In addition to least-squares-based approaches, a SLAM problem can also be described as a Bayesian probability problem where the current pose confidence coefficient of a robotic system is calculated based on the posterior probabilities of the previous pose confidence coefficient, observed map and control signal [17]. Most of the approaches under this category adopt Extended-Kalman-Filter (EKF) or Particle Filter (PF).

Due to the non-linearity nature of the SLAM system, early EKF-based algorithms suffered from inconsistency in the mapping process [18] [19] [20] [21]. Researchers extensively discussed the inconsistency and error accumulation problem in the EFK-based SLAM systems [22]. Approaches such as IEFK [23] and UFK [24] improved the convergent speed of Kalman filtering for non-linear problems, but did not fundamentally address the inconsistency issue.

The computational complexity of EKF grows exponentially with the number of feature marks recorded on the map. Some works uses the Extended Information Filter (EIF) and Sparse

Extended Information Filter (SEIF) to simplify the complexity growth to a linear problem [25][26].

Particle filter based algorithms also demonstrate a decent performance in SLAM systems. Rao-Blackwellized Particle Filter (RBPF) assumed features on the map are independent to each other and only connected by the trajectory of the robot. Similar to Kalman filter approaches, let the current state of a robot be described as (x, y, θ). In a map space $\{0,1\}^{MN}$ where M and N are the coordinates of the girds, the problem a RBPF is trying to solve can be described as:

$$w_t^i = \int p\left(z_t \mid x_t^i, m_t\right) p\left(m_t \mid z_{1:t-1}, x_{1:t-1}^i\right) dm_t \tag{2.4}$$

The most significant contribution of the RBPF algorithm included separating the probability of SLAM into two questions. The current state of the SLAM system can be represented by the product of the posterior probability of the trajectory state and the posterior probability of the map state. The separation of the calculation significantly improved the performance of the RBPF-based SLAM systems [27][28].

As an 2D RBPF-based LiDAR SLAM, Gmapping is proposed in [29]. Gmapping depends on supplemental odometry as the source of its pose prior probability. While the robot is moving through the environment, Gmapping uses the observation from the LiDAR sensor to lower the uncertainty of its position in the room. It uses resampling to update the system map to avoid imbalanced weight in particles. Figure 2.5 shows the probability distribution of the robot position in an open-end corridor, close-end corridor, and open space.

Gmapping is accurate in small spaces and generally outperforms Hector SLAM during sharp motions. However, since each particle carries a copy of the maps, the computational complexity dramatically increases with the size of the map. Additionally, Gmapping requires additional odometry to access its pose prior probability (Figure 2.6). In practice, wheel odometer is often selected as a cost-efficient solution. Extra odometry increases the complexity of the system. However, the prior probability of the pose information also allows Gmapping to tolerate low-performance LiDARs.

Another approach called Cartographer also uses particle filter to estimate its system pose

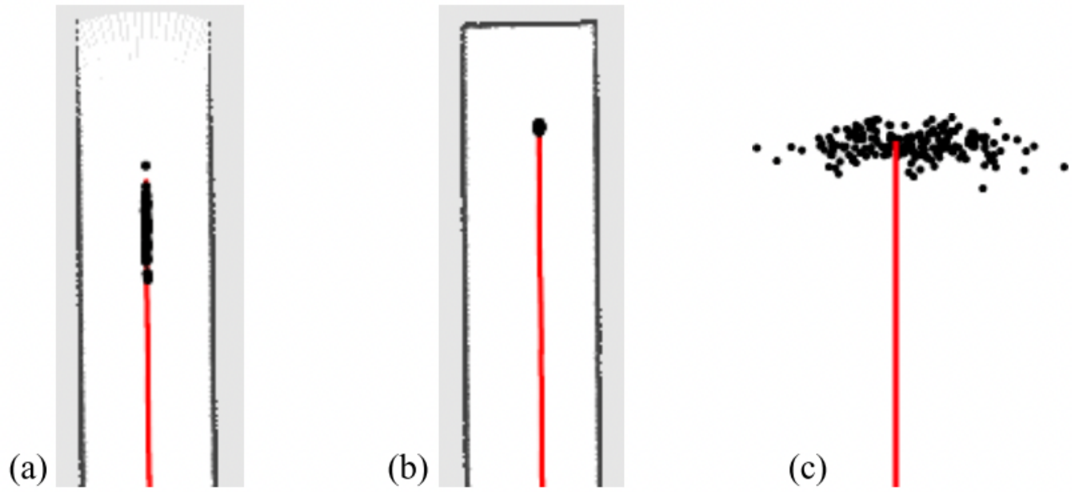


Figure 2.5 RBPF probability distribution of the robot position in (a) open end corridor, (b) close end corridor, (c) open space. © [2011] IEEE. Reprinted, with permission, from [Giorgio Grisetti, Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters, IEEE Transactions on Robotics, 2007]

[30] [31]. Besides the particle filter, the system uses the pose graph to limit the complexity of the calculations. This specially suits large scale mapping scenario where an excessive amount of particle would be generated for pose estimation.

2.2.1.3 Grid map

Occupancy grid is the primary method used by 2D SLAM system to represent the mapping result. 2D SLAM methods such as Hector SLAM [15], GMapping [29], Cartographer [16] and KrtoSLAM [32] match the scans with the established maps to update the grid through a linear interpolation. Each grid cell represents the possibility of having obstacles in the area. Since a grid cell is the unit on the map, it also indicates the resolution of the map. Due to the nature of grid maps, most of the approaches have the potential to be transformed into 3D methods [33] [34]. However, the computational cost increases dramatically as the resolution and map size expands [31].

Grid map-based approaches mainly feature navigation. There is a lack of details on the map. Increase the map resolution helps generate high density map. However, a high-resolution map

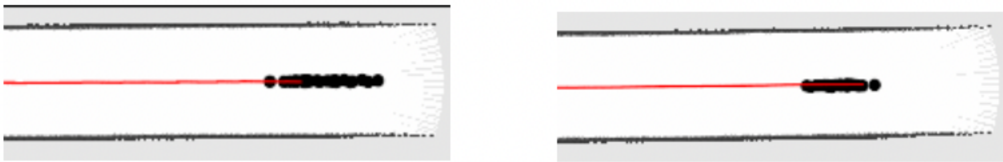


Figure 2.6 RBPF probability distribution with (Left) and without (Right) wheel odometer initialisation. © [2011] IEEE. Reprinted, with permission, from [Giorgio Grisetti, Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters, IEEE Transactions on Robotics, 2007]

often causes the gradient-based scan-matching algorithms being trapped in a local minimum. Another approach uses a multi-level resolution grid map to balance the readability of the map and the stability in scan-matching process [15].

2.2.2 Three-dimensional LiDAR SLAM

Most of the 2D SLAM methods have the ability to extend their function to 3D mapping [35][36]. However, the concept of using grids to store the map and register scans makes the system computationally inefficient as both the LiDAR resolution and mapping details will have improvement over time. Instead of using grids, with the recent trend of directly processing point clouds, Zhang and Singh proposed a new LiDAR odometry and mapping (LOAM) approach which takes the advantage of certain features between scans to identify the transformation of the system [37][38].

LOAM first classifies the point cloud into plane points and corner points using their curvature. An example of feature points extraction is illustrated in Figure 2.7, with corner points coloured in yellow and plane points in red. The calculation of translation and rotation is based on the idea that the relative movement of the robotic system from the previous posture to current posture is the displacement of feature surfaces ad edges between two consecutive LiDAR frames. As illustrated in Figure 2.8, let l be the feature point in the current scan sweep, j be its closest neighbour. The method of pose estimation with corner feature is shown in Figure 2.8(a), where a corner feature point in current scan should close to the corner feature points in the previous scan. Similarly, in Figure 2.8(b), the plane feature points in current scan should be close to the plane feature points in the previous scan [37].

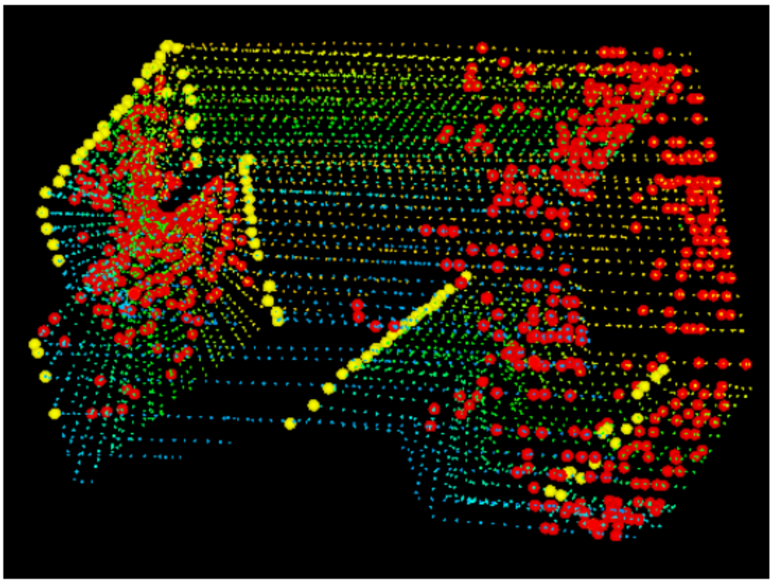


Figure 2.7 Corner and Plane feature extraction with LOAM. © [2011] RSS Foundation. Reprinted, with permission, from [Zhang, Ji, LOAM: LiDAR Odometry and Mapping in Real-time, Robotics: Science and Systems, 2014]

Rotation in LOAM is described using Rodrigues formula [39] which later becomes the most adopted method in 3D LiDAR SLAM to estimate frame-to-frame rotations. LOAM also use timestamp interpolation to smooth the motion blur within each LiDAR sweep. Researchers also described multiple approaches to improve the motion blur issue, thus, to adapt the LOAM algorithm to different LiDAR models [37].

Variations of LOAM were developed by researchers to suit different scenarios [40][41][42]. LeGO-LOAM was developed based on LOAM with optimisation for ground vehicles [43]. Compared with original LOAM, LeGO-LOAM offers filtering functions to remove the ground surface from the point cloud. Moreover, it assigns points into clusters. Clusters with not enough points are considered as moving objects, and therefore removed from the frame. With less interference from the ground points and the moving obstacles, LeGO-LOAM is able to achieve better performance with less computational complexity than the original LOAM. However, the assumption of a flat ground plane limits the application of the algorithm.

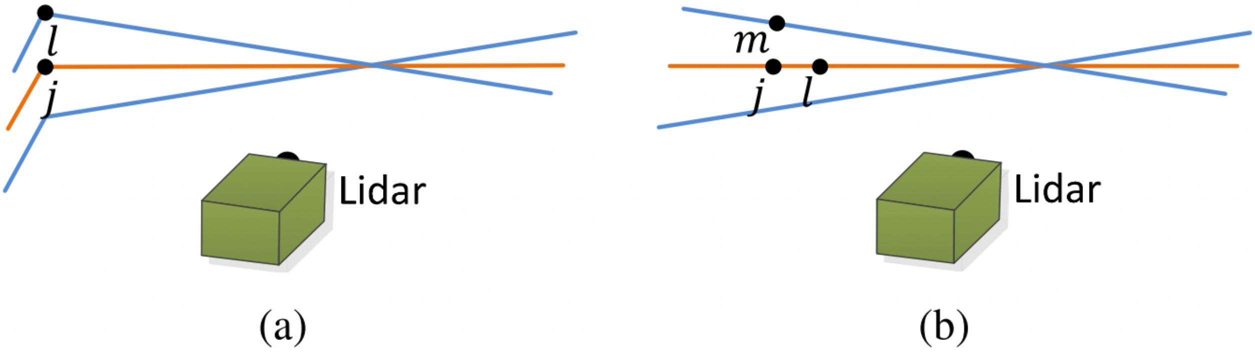


Figure 2.8 Frame-to-frame pose estimation in LOAM (a) corner features, (b) plane features. © [2011] RSS Foundation. Reprinted, with permission, from [Zhang, Ji, LOAM: LiDAR Odometry and Mapping in Real-time, Robotics: Science and Systems, 2014]

2.2.2.1 Point cloud map

As described in Section 2.2.1.3, a 2D SLAM mainly uses an occupancy grid as the map representation method. However, a grid map is not applicable for a high-density map representation. As a result, most of the 3D SLAM approaches directly use a point cloud representation. Compared with grid maps, point cloud maps are more accurate, informative and flexible. As a trade-off, point cloud maps can be storage consuming. The discussion of effectively storing the point clouds firstly started in the geographic information system (GIS) industry where information needs to be extracted from city-scale maps when complex 3D structures involved. Researchers use quad-tree and oct-tree structures to simplify the indexing process of the map [44] [45] [46] [47] [48]. Using a tree structure also allows filtering the point with their structural features. With points recorded in the tree structure, methods such as VoxelGrid filter, K-nearest-neighbour, kernel convolution, intensity weighted, density weighted or other statistic-based methods can be applied to filter the points on the map [49] [50] [51] [52] [53] [54] [55] [56]. Partial Differential Equations (PDEs) can be used to extract directional information from subsets of a point cloud, which supports semantic segmentation of the point cloud [57] [58] [59].

2.2.3 Visual SLAM and other approaches

This thesis focuses on LiDAR SLAM systems. However, other SLAM approaches are reviewed in this section for comparison purposes. Visual SLAM typically means a SLAM system with cameras as its major sensing module. To be more specific, camera SLAM systems include systems built on top of mono-cameras, stereo-cameras and RGB-D cameras. Comparing with LiDAR, cameras have significantly lower cost and more dense scanning results. However, cameras often have limited field-of-view and strongly affected by environmental lighting [60].

Like LiDAR SLAM systems, visual SLAM systems can be described in four components: visual odometry, backend optimisation, mapping unit and loop closing. Depending on the processing of camera readings, visual SLAM systems are categorised into direct methods and indirect methods.

Indirect visual SLAM has many common features with LiDAR SLAM. Instead of directly using the image readings from the camera module, an indirect visual SLAM system focuses on extracting features from the images. These features include shapes such as lines and corners, bright points, curvatures and optical flows. An indirect visual SLAM solves the SLAM problem as a series of geometric problems.

As one of the least hardware-dependent SLAM systems, MonoSLAM approach uses only a mono-camera as the input source [61]. The system extracts corner points using Features from Accelerated Segment Test (FAST), which takes the grey-scale information as the reference to find the most distinct points in the region. With EKF, the system state of the MonoSLAM is represented by the feature points and the camera pose. The MonoSLAM only focuses on odometry and does not offer mapping functions.

PTAM is a visual SLAM system which performs mapping and localisation at the same time. The system firstly uses the FAST algorithm to extract feature points from four layers of Gaussian Pyramid filter [62]. It then calculates the Shi-Tomas score [63] of each feature and filters the features based on their score ranking. Linear triangulation method is adopted to recover the depth information from the 2D images. Bundle Adjustment (BA) in PTAM is based on the Levenberg-Marquardt (L-M) algorithm. It is the first visual SLAM system that uses least-squares error optimisation instead of EKF methods.

ORB-SLAM is one of the most adopted visual SLAM systems [64]. The most significant contribution of ORB-SLAM is that all modules in the system are based on the same collection of Oriented FAST and Rotated BRIEF (ORB) features [65][66]. The consistency between modules enhances the stability of the system as a whole. Oriented FAST uses a vector from the centre of the FAST selection to the mass centre of the grey-scale value to assign an orientation value to each feature point, thus ensuring the rotation invariance of the system.

A direct visual SLAM algorithm does not preprocess information observed from the camera module. Algorithms that belong to this category rely on comparison between picture frames and have less in common with LiDAR SLAM systems. DTAM is a method which uses energy minimisation method to recover depth information from 2D images [67]. Directly comparing the geometrical features in the depth graph provides an estimation of the pose update. The geometric and scale information on a series of 2D images could also provide contributions to the trajectory recover [68]. Without feature extraction, most of the direct visual SLAM algorithms are GPU intensive. Instead of the entire image, Semi-direct visual Odometry (SVO) only uses the pixels around feature points for pose estimation, which notably optimises the system's response speed [69].

2.3 Enhancing robustness and stability of SLAM methods

Most of the LiDAR-specified SLAM approaches are proposed for a specific scenario. Generalising these systems for a broader range of applications is an important research field. The problem these studies focuses on is to stabilise the SLAM algorithms by handling challenging situations in dynamical scenarios.

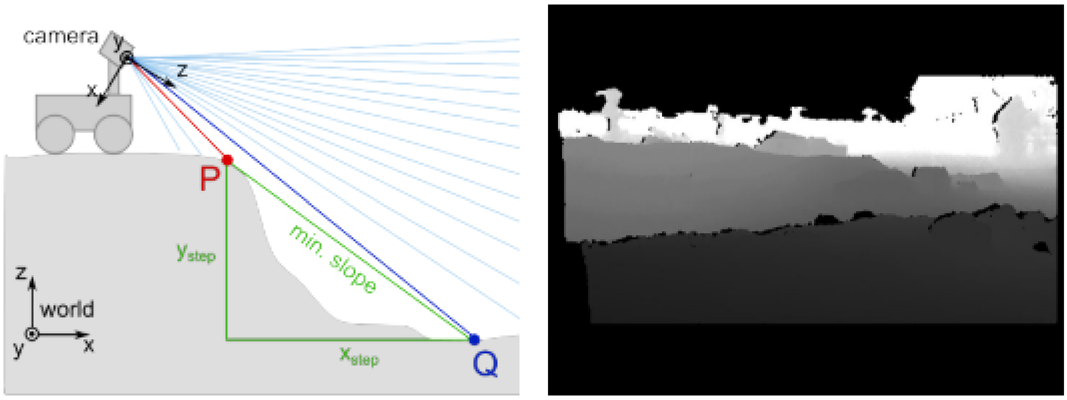
2.3.1 Sensor fusion

Among all approaches, sensor fusion is the most well-adopted approach for SLAM systems to improve the mapping performance [70] [71] [72]. For example, researchers uses wheel odometry and Extended Kalman Filter (EKF) to improve the robustness of the system [73] [74]. These studies are based on LiDAR-only solutions. By adding wheel odometry to the system, these methodologies outperform the original algorithms in specific circumstances, such as a long corridor. Similarly, other studies have also used Magneto-Inertial Measurement Unit (MIMU) or Inertial Measurement Unit (IMU) to improve the mapping accuracy during sharp motions [75]. Comparing with wheel odometry, IMU offers more flexibility for the system. It is worth noting that studies have shown using a MIMU could significantly reduced the system error accumulation [76].

2.3.2 Environment classification

Fatal errors most likely appear during an environment change. Most of the SLAM algorithms are environment specified. Configurations of a algorithm and the selection of sensors are designed especially for particular types of surroundings. Once the feature changes, the pre-configured set-up would not successfully identify the environment, and therefore leads to fatal errors. Environment classification is an approach to estimate possible environment change for SLAM process[77][78] [79] [80] [81] [82]. Researchers use a stereo camera to identify the slopes in the front of the robot [83]. As shown in Figure 2.9, a system changes its odometry algorithm when the vehicle is entering an area with different slop. Once a slope is detected, the system uses a different set of odometry to estimate the posture of the robot. Indoor and outdoor detection is also common for robotic systems to quickly swap between algorithms. Foe example, GPS signals were used to classify indoor or outdoor environments for a hexacopter camera mapping system [84]. When no GPS reception is detected, the hexacopter will switch to an indoor mode where a LiDAR and an IMU are used to locate itself in the building.

Collier and Ramirez-Serrano [85] uses an Artificial neural network (ANN) based classifier to evaluate whether the robot is currently in an indoor or outdoor environment. The neural network is trained based on the existing data. The system integrates a monocular camera dedicated to classifying the state of the environment. The ANN decides if the robot should rely on a LiDAR-based SLAM system or a stereo camera-based terrain mapping system. A more straight



(a) schematic (b) depth image

Figure 2.9 Using stereo camera to detect slope in the environment. © [2011] RSS Foundation. Reprinted, with permission, from [Christoph Brand, Stereo-vision based obstacle mapping for indoor/outdoor SLAM, 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sept. 2014]

forward approach to distinguish different environment is taking advantage of GPS sensor signal strength. Dill, De Haag, Duan, Serrano and Vilardaga [84] used the number of satellites a Global Navigation Satellite System (GNSS) is communicating with to obtain the current situation of the system. Stable GNSS readings will lead the robot to use GNSS and IMU to estimate its posture on the map. On the other hand, if GNSS lacks satellite connection, then the robot will rely on LiDAR and RGB camera to locate itself. Elevation information could also be used to determine terrain condition during a SLAM process [86]. A diagonally installed LiDAR gives elevation readings of the obstructions in front of the robot. Based on different elevation readings, the terrain can be classified into the ground, obstacle and overhanging objects. The classification results further allow the system to decide if it is entering or exiting from an indoor space.

2.3.3 Submaps

Constructing a map in one piece is risky. Instead, researchers found dividing the map into many submaps helps isolating, identifying and correcting mapping errors. Submapping can be seen in early SLAM studies [87] [88] [89] [90], where the relative translation between two submaps are re-evaluated after the mapping process. Time, distance and landmarks could be used as an indicator to trigger the segmentation. In another work, a new submap is created every 7 meters the robot travels [77]. It is worth mentioning that the sub-mapping in this work also contributed to reducing the impact of long corridors and other featureless scenarios in the mapping results. Brand, Schuster, Hirschmuller and Suppa [91] adopts a similar approach. Instead of 7 meters, their approver creates submaps in every 2.5 meters or after a significant rotation. However, in this approach, not all submaps are integrated into the global map in the world frame. Discarding low-quality submaps in a matching process helps to keep the accuracy of the results. Submapping helps to improve the efficiency of traversing through local mapping results. Moreover, it contributes to loop closing during a SLAM process, as matching submaps evaluate the likelihood of revisiting the same spots.

2.3.4 Loop closing and drift correction

Closing the loop in a SLAM system helps to reduce drifts accumulation and corrects existing errors on the map. ICP-based loop closing approaches directly use the geometrical information of the map sections to identify the overlapped area in a SLAM process [92].

However, the challenges of loop closing in a SLAM process come from three major aspects. Firstly, landmarks and feature points, such as chairs, building corners and trees, are largely repetitive in the environment. The difficulty of distinguishing landmarks from each other burdens the efficiency of loop closing. Secondly, the current reading of a SLAM system can only reflect a fraction of the mapped space. Restricted vision limited the number of comparison candidates in the loop closing process. Finally, the search scope of the loop closing targets grows with the size of the map recorded. Traverse through a massive amount of candidates in large-scale SLAM processes dramatically slows the loop closing process.

To address the above challenges, Researchers use information entropy to select landmarks in loop closing [93]. Similarly, another work used Gabor-Gist pattern to define feature points [94]. Additionally, this work adopted Principal component analysis (PCA) to simplify the features. The scale-invariant nature of features could also help identify landmarks on the map [95]. Ramos, Fox and Durrant-Whyte [96] used Conditional Random Fields (CRF) to evaluate the similarity of related features, whereas Campos, Correia and Calado [97] used image clusters to identify the most significant keyframes in the scans.

Some recent works adopted bag-of-words (BoW) methods which feature a k-means cluster [98]. Using Lisbon Zoo as an example, Caballero, Pérez and Merino [99] integrated 3D LiDAR and cameras to map a large proportion of the zoo continuously. This work used 3D LiDAR to record the environment as point clouds. A stereo camera is equipped to take keyframes during the operation of the SLAM process. The method extracts BoW features from keyframes periodically. Each BoW feature is then stored into a database with posture information recorded at the moment of that frame. Similarly, a match in BoW between scans would provide the system information to recover from cumulated localisation errors [64] [100] [101].

2.4 Multiple LiDAR cooperation in SLAM systems

Combining multiple LiDARs in a mapping unit often aims at enhancing the performance of a SLAM system. Despite the fact that multi-LiDAR system requires extra efforts to merge the readings before processing, adding an extra LiDAR to the system directly enlarges the FOV of the sensing unit. In most of the multi-LiDAR systems, LiDARs are horizontally aligned to ensure the mutual coverage of their scanning FOV [102] [103] [104]. In these works, the number of LiDARs directly amplifies the scanning field. In Sualeh and Kim [104], five LiDARs are mounted on each side of a car as illustrated in Figure 2.10, with their scan direction parallel to the ground. This work used 16-line LiDARs to perform object detection in a merged point cloud. These works emphasised merging multi-LiDARs to generate an enormous point cloud, which requires calibration during initialisation.

Calibration of multiple LiDARs aims at finding the transformation matrix between the LiDARs and the robotic system odometry. Kim and Park [105] used reflective conic shapes appear in both LiDAR scan results to calculate the displacement and rotation between them (Figure 2.11(b)). Checkerboard calibration methods are effective for calibrating mixed types of sensors [106] [107] [108] [109]. Figure 2.11(a) illustrates a LiDAR-Camera calibration method

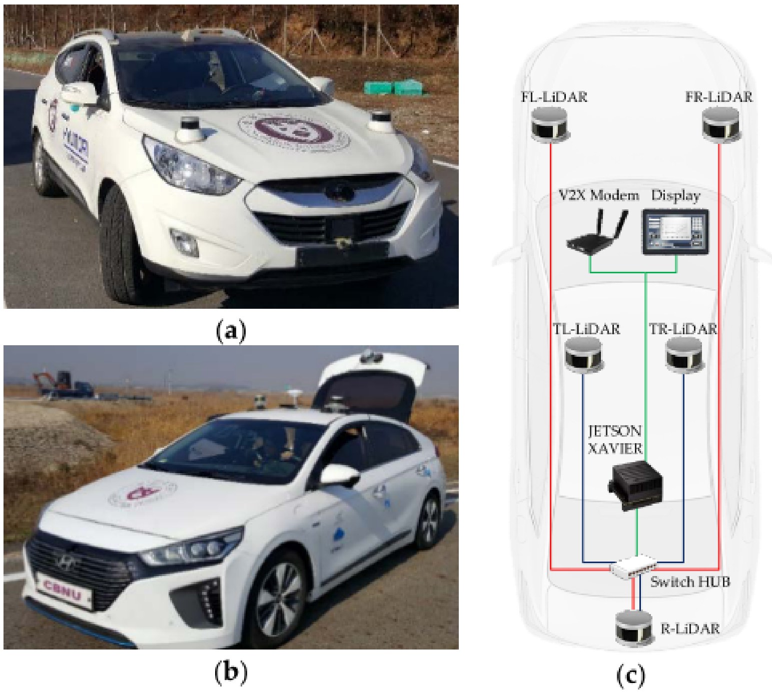
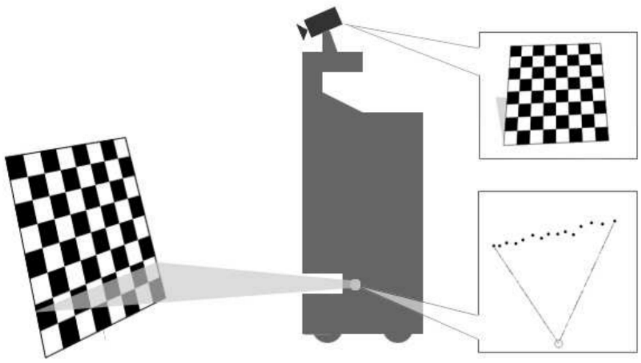


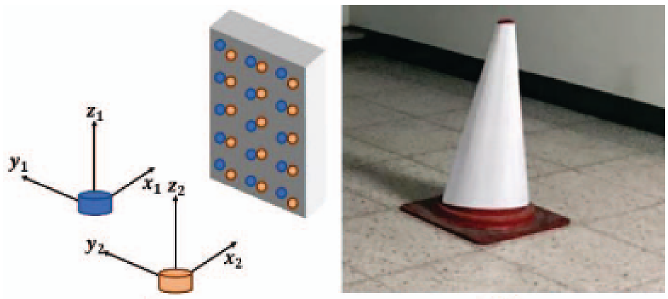
Figure 2.10 LiDAR Installation in [104] (a) Front LiDARs, (b) Side LiDARs, (c) Top view LiDAR Orientations. © [2017] IEEE. Reprinted, with permission, from [Taehyeong Kimd, Calibration method between dual 3D lidar sensors for autonom-ous vehicles, 2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE), Sept. 2017]

proposed in [110] where the relationship between the two sensors is calculated based on the deformation of the board length appearing in the reading. Changing the shape and pattern on the calibration board allows different sensors to identify their transformation matries to the checkerboard, and thus with respect to other system components. Other than checkerboard, Pereira, Silva, Santos and Dias [111] used a spherical shape item to calibrate both cameras and LiDARs on an autonomous vehicle. However, the sensor position is constantly changing in real-world scenarios. Some calibration methods are not relying on a pre-known target. Calculating the geometrical relationship between two sets of point clouds only requires the LiDARs to have overlapped scanning area [112] [113]. When processing high-frequency LiDAR readings, synchronising the readings and minimising the time gap between the LiDARs is critical to the merging process. Researchers also discussed the effects of timestamp and synchronisation in multi-LiDAR calibration [114] and [115].

Instead of finding the nearest neighbour between two sets fo point clouds, some studies



(a) Camera-LiDAR calibration with checkerboard. © [2004] IEEE. Reprinted, with permission, from [Qilong Zhang, Extrinsic calibration of a camera and laser range finder (improves camera calibration), 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2004]



(b) Dual LiDAR calibration with reflective cone shape. © [2017] IEEE. Reprinted, with permission, from [Taehyeong Kim, Calibration method between dual 3D lidar sensors for autonomous vehicles, 2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE), 2017]

Figure 2.11 LiDAR-Camera and LiDAR-LiDAR calibration

compared only the trajectories generated from different sensors to identify the transformation between them, which significantly reduces the computation complexity of the calibration [116] [C2].

2.4.1 Cross-dimensional feature extraction from LiDAR data

Unlike RGB cameras, which use complementary metal–oxide–semiconductor (CMOS) to generate a 2D pixel matrix, LiDARs use a moving laser beam to sample the environment. The mechanical nature of the LiDAR sensors makes the output point cloud contain strong geometrical information. Researches had use this geometrical feature of the LiDAR sensor to create high-dimensional images from low-dimensional LiDARs [117] [70] [106]. In another work, a 2D LiDAR is attached to the top of a voice coil to perform z-axis motion Yang, Yang, Tian, Zheng, Li and Wang [70]. A $10mm$ displacement generated from the voice coil allows the system to produce 2.5D maps with only a 2D LiDAR. Instead of linear motions on z-axis, rotating along the x-axis is also a common approach to produce 3D reading from 2D LiDARs [37] [118]. Pfrunder, Borges, Romero, Catt and Elfes [119] uses an inclined 2D LiDAR to scan through space with the 6 Degree-of-Freedom (DOF) motion of the ground vehicle recorded by other sensors. Similar approaches can also be found in other studies [120] [121] [122].

On the other hand, compressing 3D point clouds into lower-dimensional format contributes to the transmission and storage of the generated map [123] [124] [125]. The feature compressing methods significantly improved the mapping system's performance in the urban environment for two reasons. Firstly, urban synthetic scenes are often perpendicular to the ground [126] [127] [128]. Downgrading 3D map into 2D 'bird's-eye view' maps had little effect on the navigation system [124]. Secondly, the compressed data stream improved the connectivity of a robotic system in the network [123].

2.4.2 Solid-State-LiDAR SLAM

Spinning multi-line LiDARs occupy a large share of the LiDAR market, both in research and industry. The spinning mechanism ensures the laser beam repeatedly covers the same area in different scans. However, in recent years, solid-state LiDAR (SSL) have become common on the market. Compared with traditional spinning LiDAR, SSLs have less moving parts, more compact design, low power consumption and higher reliability [129]. More importantly, SSLs generally cost less than the spinning multi-line LiDARs [130].

Although SSL development is having a promising future, directly applying spinning LiDAR algorithms on them can be difficult. Most SSLs have irregular scan patterns, such as 'Z' shape, petal shape or ellipse shape. Researchers have re-engineered the feature matching algorithms to adapt to different kinds of scan patterns [131] [132].

Additionally, SSLs often have lower sampling rate compared with the traditional spinning LiDAR. With a rotating motor, a spinning LiDAR can easily maintain its scan frequency above $20Hz$. However, many SSLs could only provide 10 to $15Hz$ scan frequency, which requires more robust motion blur methods [129].

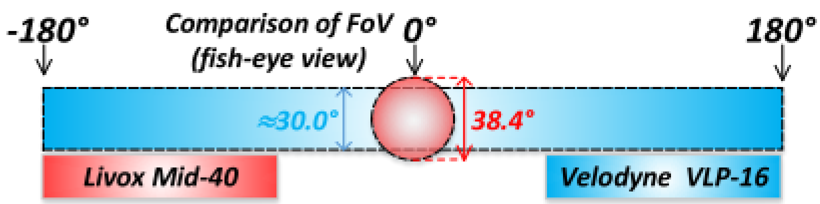


Figure 2.12 Livox Mid-40 FOV compare with Velodyne VLP-16. © [2011] IEEE. Reprinted, with permission, from [Jiarong Lin, Loam livox: A fast, robust, high-precision LiDAR odometry and mapping package for LiDARs of small FoV, 2020 IEEE International Conference on Robotics and Automation (ICRA), May 2020].

Furthermore, since most of the SSLs are manufactured with Micro-electromechanical Systems (MEMS), the optical mechanism restricts their field-of-view (FOV) [133]. Using Velodyne Velarray M1600 3D LiDAR as an example (Figure 2.12), the sensor only has a 120° horizontal FOV and 35° vertical FOV. Without fusing with other sensors, such a limited rectangle shape view window challenges the performance of the SLAM system [134]. Researchers have improved this problem by using extra information for feature matching [132] [135]. Besides depth information, the authors also use intensity as the supplemental data for feature matching.

2.5 Summary

LiDAR-based SLAM methodologies with 3 to 6 DOF have been comprehensively studied and applied to various applications to address localisation and mapping challenges faced by robotic systems. The performance of the SLAM algorithms is directly related to the stability of the mapping process, especially during challenging scenarios. Consequently, the ability to overcome mapping errors will be directly reflected in the final mapping results.

Various techniques, including sensor fusion, environment classification, submapping and loop closing have been investigated to improve the stability and robustness of the LiDAR SLAM systems. However, measuring errors in the system states during a SLAM process is outside the capabilities of existing SLAM methodologies. For this reason, little work has been performed in a complex environment LiDAR-based SLAM with flexible motions. The development of techniques that enhances SLAM stability and enables system states recovery is therefore mandated. Such a technique can be a novel SLAM approach or could build upon existing SLAM methodologies. Studies are required in both cases to propose, develop and evaluate possible solutions to provide the required techniques.

Chapter 3

Pose Correction Using Trajectory Information[1]

3.1 Introduction

Most of the common SLAM approaches use a preset value to initialise their starting system pose. In 2D mapping, this value includes the system's starting coordinates on the map and the orientation angle. Presetting the starting pose does not affect the mapping results visually as it only provides a reference for the algorithm to start the iteration. However, in most cases, the starting system pose is primarily related to the direction of the map relative to the 'world' coordinates frame.

Nevertheless, presetting the initial system pose does not always work as expected. Since most of the SLAM approaches use an established map recorded by previous LiDAR scan to estimate the current system pose, a proper estimation requires a well-established map. If the map recorded is not appropriate to support the optimisation algorithms to find a good estimation, the algorithm will likely yield a misaligned move in the next scan period. The misalignment is especially a problem during the starting phase of a SLAM approach as very little map information is recorded by the algorithm. The problem is self-limited as the map accumulates. However, the large error

[1]The works contained within this chapter have previously been published in: [C1]

distribution in the starting phase of a SLAM process defeats the propose of setting the initial pose. Often, after the starting period, the algorithm accumulated significant drift compared with its preset initial pose.

This chapter presents the design, analysis and experimentation of high robustness SLAM approach using the proposed trajectory matching approach, targeting at compensating the drift accumulation of system pose during the starting phase of a SLAM process. The chapter addresses two major research challenges. The first challenge is to improve the instability of a SLAM approach during the initialisation stage. The second challenge is to identify and correct the system pose during a SLAM process.

Two 2D SLAM algorithms were fully investigated to understand the cause of the mapping errors, including the Hector SLAM and Gmapping. The methodologies of both mapping approaches were studied along with the approaches of improving mapping results. The contribution of this study has two parts: the first part is to establish a novel approach to reduce the instability of a SLAM approach during the starting stage, the second part is to validate the capability of adapting the proposed trajectory matching algorithm into existing mapping approaches.

3.2 Study of Hector SLAM and system pose recalibration during initialisation

Hector SLAM is a classic 2D SLAM approach. The method uses the measurements from a LiDAR sensor as its minimum requirements to locate a robotic system in an unknown environment. Overall, Hector SLAM takes each sweep of the LiDAR scan and matches it with the existing map recorded by previous scans. The translation and rotation from the current scan to the map is the translation and rotation of the robot from the last system state to the current system state.

The mapping function is continuous as each pose estimate is triggered by the arrival of a new scan measurements. Hector SLAM is highly modularised. The data flow and processing between different system components are briefly introduced in Figure 3.1.

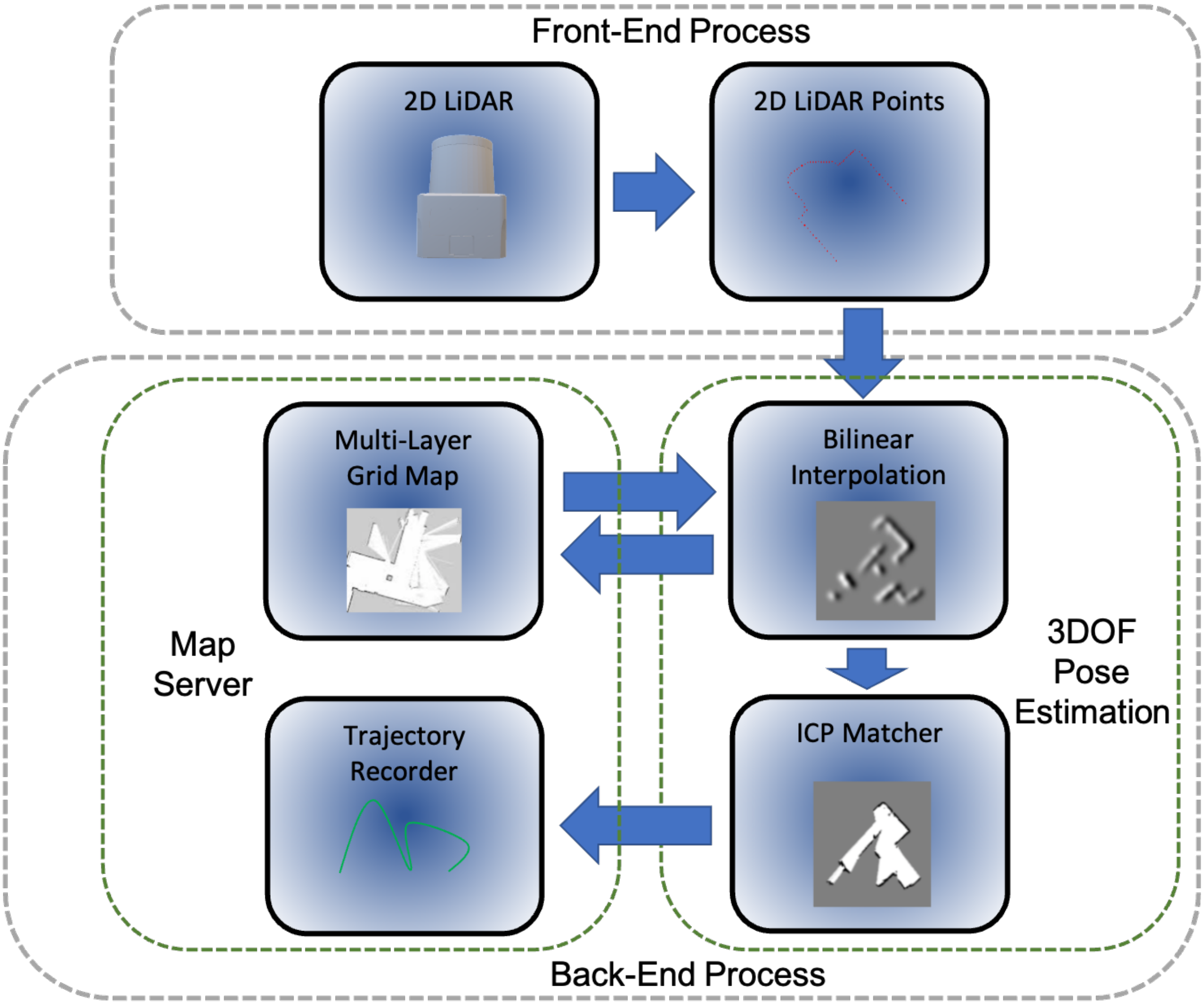


Figure 3.1 Hector SLAM system map.

As shown in Figure 3.2, Hector SLAM uses a preset $/initialpose$ to define its starting location and orientation relative to the world coordinate frame.

3.2.1 Pose estimation of Hector SLAM

Understanding the mapping mechanism in the Hector SLAM is required to improve the pose estimation. The map of Hector SLAM is recorded on an occupancy grid. Each grid cell represents the possibility of having an obstacle in the cell location. As shown in Figure 3.3, the map uses greyscale to represent the possibility.

The Hector grid map uses cells with its unit length set to 1. The resolution of the map controls by the scale of the coordinates. The higher the resolution, the smaller the area each cell

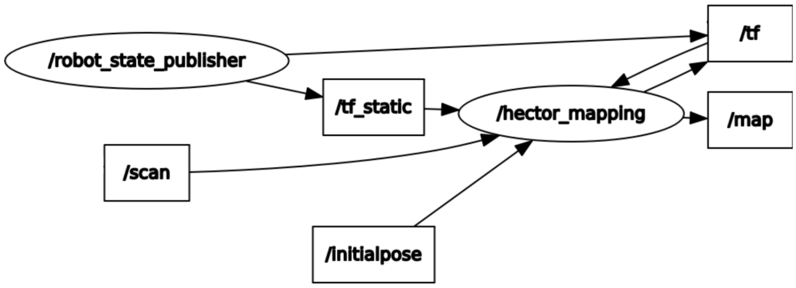


Figure 3.2 A software architecture of Hector SLAM under ROS framework.

Figure 3.3 Grid map generated using Hector SLAM.

represents. The map is updated with scan information from each LiDAR sweep. Hector SLAM uses bilinear interpolation to update the corresponding grid cells.

Using Figure 3.4 as an example, let M be the grid map, and P_m be the individual coordinate. Using bilinear interpolation, the gradient of a given point (x,y) on the map $\nabla M(P_m)$ can be written as:

$$\nabla M(P_m) = \left(\frac{\partial M}{\partial x}(P_m), \frac{\partial M}{\partial y}(P_m)\right) \tag{3.1}$$

Therefore, the grids surrounding a LiDAR reading coordinate are $(P_{00}, P_{01}, P_{10}, P_{11})$. According to Equation (3.1), the derivative of any coordinate on the map can be written as:

$$\begin{aligned}\frac{\partial M}{\partial x}(P_m) \approx & \frac{y-y_0}{y_1-y_0}\left(M(P_{11}) - M(P_{01})\right) \\ & + \frac{y_1-y}{y_1-y_0}\left(M(P_{10}) - M(P_{00})\right)\end{aligned} \tag{3.2}$$

$$\frac{\partial M}{\partial y}(P_m) \approx \frac{x-x_0}{x_1-x_0}\left(M(P_{11}) - M(P_{10})\right) + \frac{x_1-x}{x_1-x_0}\left(M(P_{01}) - M(P_{00})\right) \tag{3.3}$$

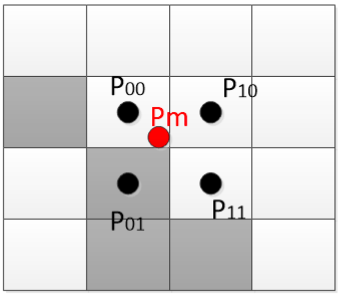


Figure 3.4 Locating Point P_m On the grid map.

With the map as M, a pose on the map at point P_{xy} can be written as:

$$\xi = (p_x, p_y, \psi)^T \tag{3.4}$$

The transform of the pose is calculated during the scan matching process. Future pose, $\Delta\xi$, is the pose that provides a minimum error in the scan matching when comparing the map with the scanned occupancy. Following is the scan matching process where n is the number of scan readings in each sweep.

$$\sum_{i=1}^{n} [1 - M(S_i(\xi + \Delta\xi))]^2 \to 0 \tag{3.5}$$

Let H be the Hessian matrix where

$$H = \left[\nabla M(S_i(\xi)) \frac{\partial S_i(\xi)}{\partial \xi}\right]^T \left[\nabla M(S_i(\xi)) \frac{\partial S_i(\xi)}{\partial \xi}\right] \tag{3.6}$$

With the previous equation,

$$\frac{\partial S_i(\xi)}{\partial \xi} = \begin{bmatrix} 1 & 0 & -\sin(\psi)S_{i,x} - \cos(\psi)S_{i,y} \\ 0 & 1 & \cos(\psi)S_{i,x} - \sin(\psi)S_{i,y} \end{bmatrix} \tag{3.7}$$

After the first order Tayler expansion, solving the minimisation problem of $\Delta\xi$ with Gauss-Newton equation gives:

$$\Delta\xi = H^{-1} \sum_{i=1}^{n} \left[\nabla M\left(S_i(\xi)\right) \frac{\partial S_i(\xi)}{\partial \xi} \right]^T \left[1 - M\left(S_i(\xi)\right)\right] \tag{3.8}$$

This can yield a step of $\Delta\xi$ towards the minimum error value. Same as other gradient descent methods, Hector SLAM suffers from local minimum problem, as the above process is a non-smooth linear approximation. It uses an optional multilayer map with different resolutions to reduce the chance of locking in local minimums.

Hector SLAM can use up to three layers of maps with different resolutions to perform localisation at the same time. The multi-layers map is particularly used in the situation where a local minimum in gradient descent is filtered out on a lower resolution map. Illustrated in Figure 3.5, with less feature represented on a lower resolution, the convergence of system pose is more robust but less accurate.

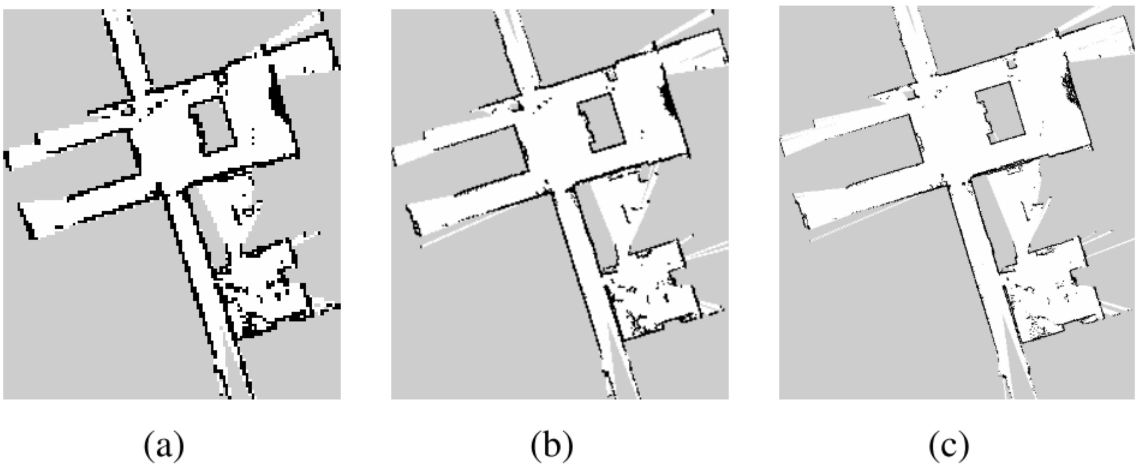


Figure 3.5 (a)Low Resolution, (b)Medium Resolution, (c)High Resolution. © [2011] IEEE. Reprinted, with permission, from [Stefan Kohlbrecher; Oskar von Stryk; Johannes Meyer; Uwe Klingauf, A flexible and scalable SLAM system with full 3D motion estimation, 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics, 1-5 Nov.2011]

3.2.2 Position instability during system initialisation

The Hector SLAM uses the existing mapping results to justify its current system pose. The algorithm needs to have a well-established map in order to produce an accurate pose estimation. Without a minimum amount of map recorded by the algorithm, Hector SLAM, as well as most

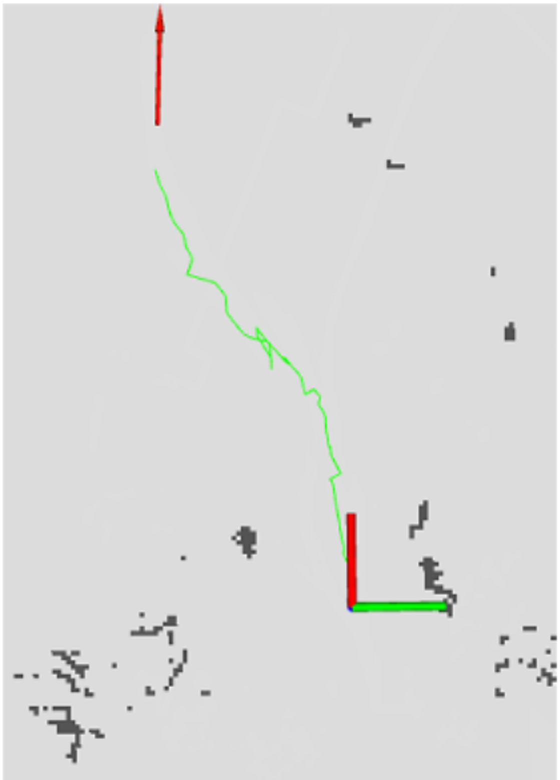


Figure 3.6 Joggling of the system pose of a Hector SLAM at starting phase.

of the other SLAM approaches, will provide unstable pose estimation. This problem gets severe during the starting stage of a SLAM process. The instability is due to the gradient descend based position optimisation not being able to find a reliable convergent. As shown in Figure 3.7, with the algorithm starting in identical scenarios, Hector SLAM provides slightly different pose estimation in four attempts. This problem is self-limited during mapping and generally will not keep affecting the mapping process once the occupancy map is properly recorded. However, the joggling of position and orientation (Figure 3.6) at the start of the algorithm brings significant influence to the future pose estimation of the robotic system, and results in a drift accumulation between the robot frame and the map frame.

The above problems can be improved in many aspects. Traditionally, most of the approaches are fall into two categories. The first category is increasing the sensing power. This often achieved through the increasing of sensor resolution and accuracy or fusing different sensor results to overcome a limitation of a particular type of sensor. The second category is to enhance

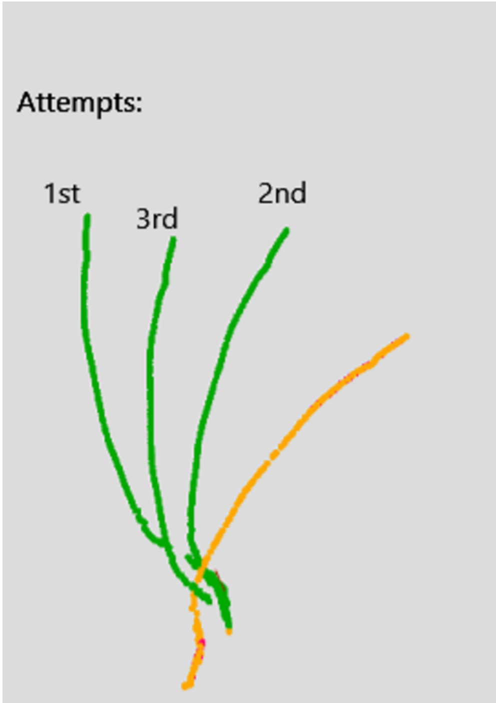


Figure 3.7 Four Hector SLAM attempts with same dataset.

readings extracted from sensors. This approach requires significant preprocessing of the sensor readings and often only suited for specific scenarios.

However, enhancing the sensing power can not eliminate the instability of the SLAM algorithm at starting stage as the algorithm still needs time to establish a map. Additionally, all these improvements require to be applied to the SLAM process in real-time or semi-real-time. This means the errors recorded on the map will be accumulated, and still affect future scans. Our study presents a different approach than those mentioned above. Instead of a real-time process, the approach proposed in this chapter is to correct the result of a SLAM process asynchronously using a supplementary trajectory.

3.3 Using trajectory information for orientation recalibration

The work present at this chapter aims to improving the instability of a SLAM algorithm at stating phase. In proposed approach, this is achieved by matching the SLAM trajectory to the fixed frame trajectory of the SLAM using their geometrical information. This is based on the fact that modern SLAM systems generally have multiple sensing modules, such as wheel odometry, GPS and ultrasound. In traditional approaches, researchers use the Extended Kalman Filter (EFK) to fuse all reading to enhance system stability. However, using EFK only improves the accuracy of the current pose estimation. The EFK methods could not identify the accumulated mapping errors and system relocations. Instead of fusing readings in real-time, this chapter propose a method which uses the trajectory of different pose estimation module to identify localisation errors.

This work define the trajectories as follows:

- **The SLAM Trajectory** is the trajectory generated by the SLAM algorithm. This work is only focusing on SLAM algorithms with a single LiDAR as input.
- **The Reference Trajectory** is the trajectory generated by a sensing unit separated from the SLAM algorithm. It can be a GPS module mounted on the robot or a laser interferometer-based tracker carried by another robotic system.

In detail, the proposed approach uses auxiliary sensors on the robotic system, such as GPS, laser tracker or Ad-Hoc networks, to generate a reference trajectory. The idea based on the fact that most of the modern SLAM systems are a combination of multiple sensors using completely different technologies. Many of these auxiliary sensors are absolute positioning sensors for which their errors are non-accumulative. Instead of sensor fusion, this work use these sensors to obtain two independent trajectories: the first trajectory being a collection of LiDAR SLAM system pose, and the second trajectory being a collection of the pose estimations from the auxiliary sensor.

The reasons this work choose generating separate trajectories over sensor fusion are the followings:

- Directly fusing absolute positioning sensors measurements to the LiDAR mapping system is problematic. It is mainly due to their large noise distribution and low update rate. LiDAR SLAM approaches has been developed into high accuracy systems. Fusing sensors measurements with larger noise distribution in real-time will most likely lower the accuracy of the system outcome.

- While LiDAR and camera based algorithms need initialisation to start providing quality pose estimation, readings from GPS or laser trackers do not affect by their up-time. These absolute positioning sensors have a non-accumulative error in Gaussian distribution. Thus, they will not lose initial pose settings during the starting period.

Instead of focusing on individual sensor readings, it is more appropriate for the algorithm to compare its pose estimation from LiDAR with the pose estimation from GPS using the graphical information of the trajectories. Consider the shapes of the two trajectories with some similarities, which provide the potential to use the Iterative-Closet-Points (ICP) algorithm to estimate the rotation and translation between the Hector SLAM trajectory and the GPS trajectory.

Figure 3.8(a) illustrates an image of a Hector SLAM trajectory with a region-of-interest (ROI) section enlarged. On the right side, in Figure 3.8(b), the blue marker points represent the trajectory recorded from a laser tracker during the same period. The overall idea of the proposed approach is shown in Figure 3.8(c), which is to use the geometrical information from both trajectories to find the rotation and translation between them.

3.3.1 Iterative Closest Point (ICP)

Many gradient-descent based optimisation method can be used to calculate the minimum square distance between two sets of trajectories. This study choose Iterative Closest Point (ICP) as the method to align trajectories. ICP is commonly used in many image processing and computer vision algorithms. It is a classic data registration algorithm used to align one set of points to another in a given space. It calculates the distance between each pair of points and iterates through to find the least square error. Figure 3.9 illustrates the matching process of the ICP algorithm.

The main idea of using ICP algorithm in this research is to calculate the minimum square error for two sets of trajectory points P and Q where $\mathrm{P} = \{p_1, \ldots, p_n\}, \mathrm{Q} = \{q_1, \ldots, q_n\}$. This can be written as:

$$E(R,t) = \frac{1}{N_p} \sum_{i=1}^{N_p} \|p_i - Rq_i - t\|^2 \tag{3.9}$$

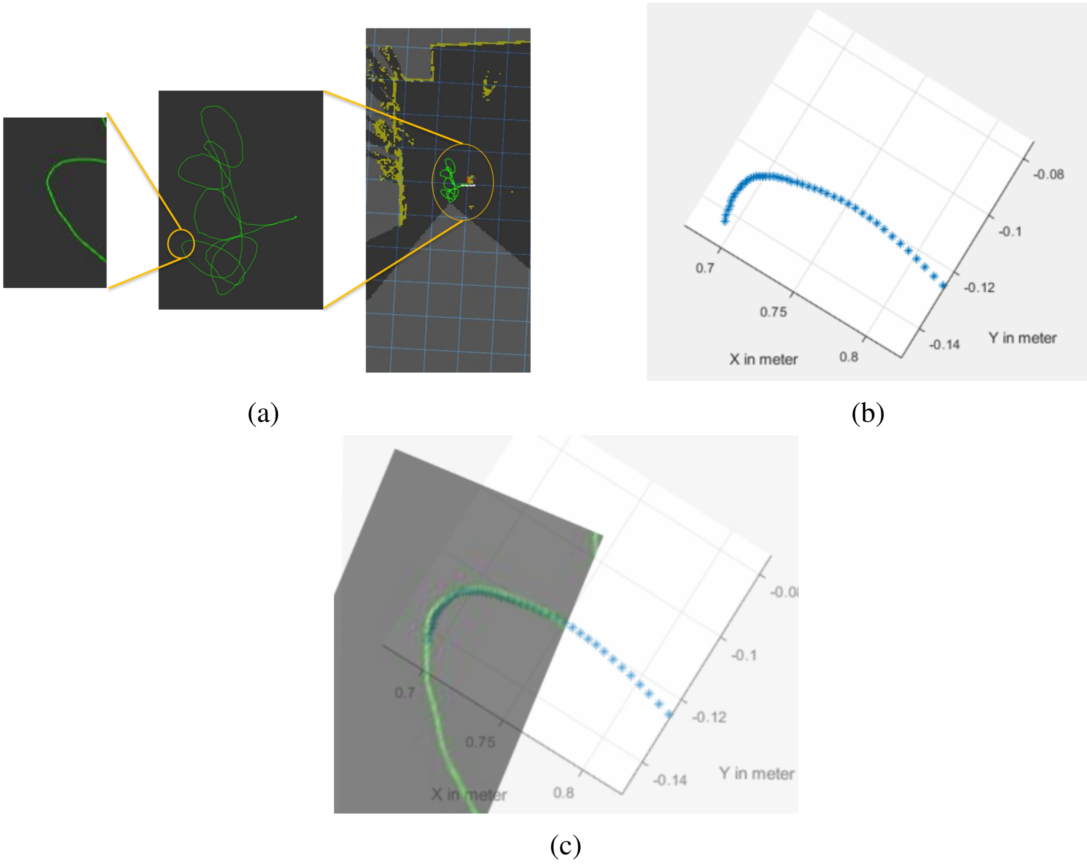


Figure 3.8 Extracting geometrical information from the trajectory of a robotic system.

Point-to-Line Iterative Closest Point (PL-ICP) works very similarly to the original ICP. The only difference here is that instead of finding the vector between two closest pair of points from two point sets, PL-ICP is attempted to find the normal vector from a point in the set to a line formed with two correspondent closest points in the target set. For each scan point in P, it allows to find two closest points in the targeting scan with index j_1^i and j_2^i. Let C_k be the point-to-segment corresponding to step k. Then, Equation (3.9) can be rewritten as:

$$J\left(q_{k+1}, c_k\right) = \sum_i \left(n_i^T \left[R\left(\theta_{k+1}\right) p_i + t_{k+1} - p_{j_1^i}\right]\right) \tag{3.10}$$

Both ICP and PL-ICP were considered in the study. This is explained in the following sections. Figure 3.10 shows an example of a match between trajectories where the Hector trajectory is fitted into the laser tracker trajectory using proposed methodology.

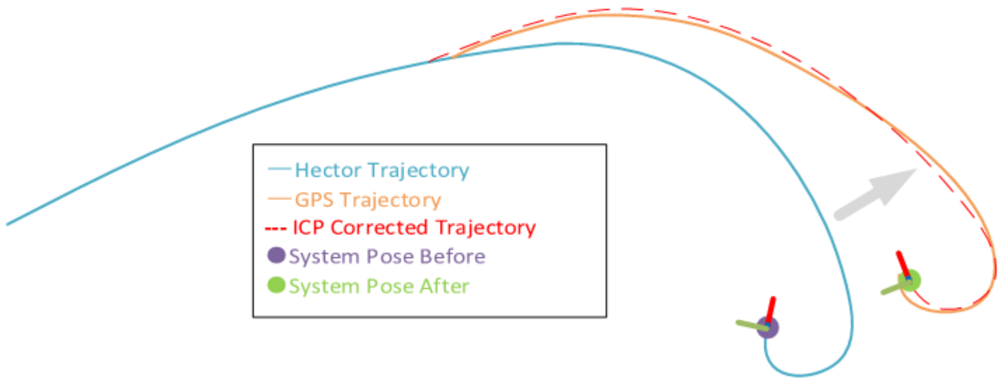


Figure 3.9 Examples of ICP Matching.

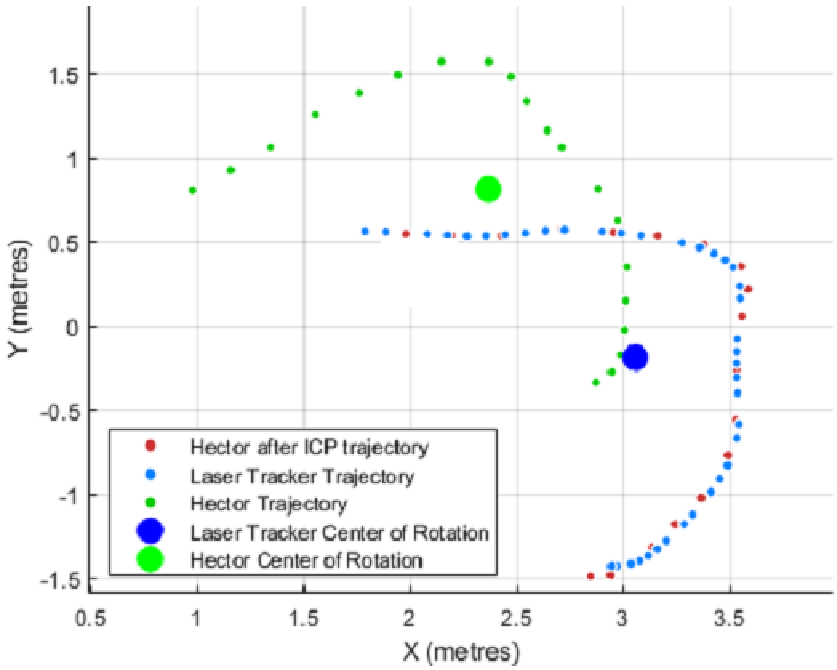


Figure 3.10 Matching Hector trajectory to the laser tracker trajectory using ICP.

3.3.2 Trajectory comparison and pose correction

Hector SLAM generates trajectories from the pose estimations between scans. A section of the trajectory is formed by a collection of recorded poses from that period of time. Each pose generated by the algorithm is a vector transferring the system from the origin of the coordinate system to its current pose. In this part of the work, the propose methodology ignore the orientation information inside pose measurements and process them as a pair of spatial coordinates. Similarly, measurements from the reference frame can be represented as a pair of spatial coordinates as well. With both pose measurements converted into coordinates, two point clouds with the pose coordinates were then constructed. Thus, ICP is used to compare two sets of point clouds and yields the translation of the rotation from it.

3.3.2.1 Extraction of orientation from geometrical information

Most of the supplemental sensors installed on a robotic system, such as GPS, are equal to or less than 3 DOF. These sensors cannot be directly used to estimate the orientation of the system. However, by grouping the trajectory points together, it is proposed to extract orientation information from the geometrical shape of the trajectory. This is achieved by estimating the rotation between two pair of trajectories.

3.3.2.2 ICP and mapping frequency

As shown in Figure 3.11, the matching results of the ICP algorithm heavily affected by the number of points in matching candidate and the target. In the current approach, this is related to the number of poses recorded from SLAM frame and the reference frame. The frequency of pose estimation in Hector SLAM is driven by the scan frequency and the system performance. As a result, the number of points in the trajectory is dependent on the scanning rate as well as computing power. The upper bound of pose update frequency of Hector SLAM has been tested on Intel Joule 570X platform and Intel i5 4250u Quad-Core @ 2.6 GHz platform respectively. The Intel Joule 570X equipped with a Quad-Core CPU running at 1.7 GHz with 4 GB of RAM, whereas the Intel i5 4250u Quad Core PC is running at 2.6 GHz with 16 GB of RAM. There are two 2D LiDAR candidates: The RpLiDAR A1 and Hokuyo UST-20LX. The RpLiDAR A1 is capable of scanning 360° with a maximum 10 Hz sweep rate. The Hokuyo UST-20LX only have a 270° Field-of-View(FOV). But it is capable of sweeping at 40 Hz. The investigation of LiDAR performances have recorded the pose estimation frequency with a different combination of configurations as shown in Table 3.1.

The update frequency of the reference trajectory was also studied. A Leica LT-500 laser tracker and a SwiftNav Piksi GPS were used to obtain the reference trajectory. In the test, Leica LT-500 is able to stream 3 DOF pose data at 100 Hz. SwiftNav Piksi is updating at a lower rate

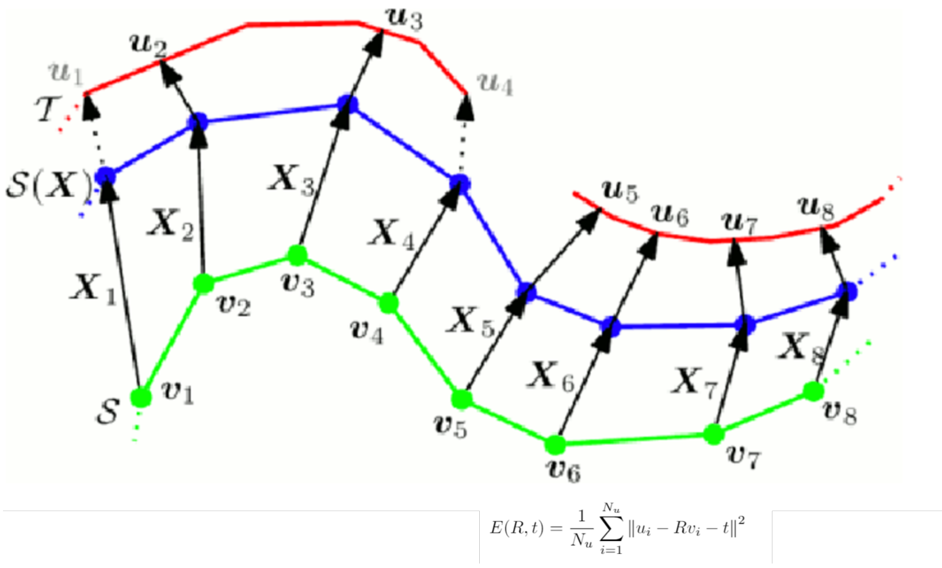


Figure 3.11 Point-to-Point ICP.

Table 3.1 Maximum stable Pose sampling rate recorded using different setups.

	Intel Joule 570X	Intel i5 4250u Quad Core @ 2.6 GHz
RpLiDAR A1	6.5 Hz	8 Hz
Hokuyo UST-20LX	29 Hz	40 Hz

at about 10 Hz. The different sampling frequency of the sensors raises the issue of unbalanced trajectory data points. This directly affects the accuracy of the ICP algorithm. Two attempts to overcome this issue were investigated:

- **Downsampling Method** The first method is to downsample the trajectory points to the same numbers in both point sets. This way, some of the information will be lost. But the point set can be directly fed into the ICP calculation.

- **PL-ICP Method** The second method is to use PL-ICP instead of the original ICP. PL-ICP uses the distance between a candidate point to a targeted line (a pair of targeted points) as the cost function of the optimisation process. It does not require to pair the points from the candidate set to the targeted set.

3.3.2.3 Pose correction in Hector SLAM

Hector SLAM estimates its system pose by matching the current scan with the existing map using bilinear interpolation [15]. Assuming S_i is the endpoint of a 2-D scan sweep. Let C_{xy} be a coordinate on the 2-D grid map $M(C)$. Let ξ be the rigid body transformation of the SLAM system where:

$$\xi = (c_x, c_y, \psi)^T \tag{3.11}$$

The minimum-square-error between scan sweep and recorded map (Equation (3.12)) will provide the best estimation of the next system pose.

$$\xi^* = \arg\min_{\xi} \sum_{i=1}^{n} \left[1 - M\left(S_i(\xi)\right)\right]^2 \tag{3.12}$$

A series of poses in a row can use to present the trajectory of the Hector SLAM process on the world frame map. Since the reference trajectory is directly obtained through GPS or laser tracker, through an ENU(East-North-Up) coordinate system, the readings can be directly used as trajectory information. Denoting two trajectories from two frames as points sets $P = \{p_1, \ldots, p_n\}$ and $Q = \{q_1, \ldots, q_n\}$. Using ICP could find the pair of translation T and rotation R that yields the minimum sum of square errors $E(R,T)$ where:

$$E(R,T) = \frac{1}{N_p} \sum_{i=1}^{N_p} \|p_i - Rq_i - T\|^2 \tag{3.13}$$

The current pose of the system should also be transformed from $P = \{p_x, p_y, \psi\}$ to:

$$P = \{(p_x + T_x^*), (p_y + T_y^*), \psi \oplus \psi_R^*\} \tag{3.14}$$

Figure 3.9 illustrates this process from the view of different trajectories. After the process, the system pose of the Hector SLAM has been aligned with the reference frame. Algorithm 1 below shows the pseudo-code for the described process. HT here refers to the trajectory generated from Hector SLAM node and RT is the trajectory from the reference frame. In actual coding, the function described here are split into a few functions.

Algorithm 1 Map Orientation Correction

Require: HT_j Trajectory from HectorSLAM since last iteration, RT_j Trajectory from Reference Trajectory since last iteration, $Iter$ number of iterations to sample the best match

Ensure: $TMatrix$ Transformation Matrix

```
MinResidual ← 1
function MATCHFINDER(TransMx,HTj,RTj)
    while i < Iter do
        TMatrix ←ORIENICP(HTj,RTj)
    end while
    ALIGNORIENT(TMatrix,Pose)
end function
function ORIENICP(HTj, RTj)
    i ← 0
    HPoints ← HTj
    RPoints ← RTj
    MinResidual ← ICP(HPoints,RPoints)
    if OrienICP.Residual < MinResidual then
        Trans ← ICP(HPoints,RPoints)
        MinResidual ← OrienICPResidual;
        return Trans
    else
        PASS
    end if
end function
function ALIGNORIENT(TransMx, Pose)
    Pose[x,y].Rotate(TransMx)
    Pose[x,y].Translate(TransMx)
end function
```

The proposed algorithm is constructed in a master-slave architecture as shown in Figure 3.12. Separating the mapping and referencing units allows the algorithm to be deployed over a robotic network. To be more specific, this system architecture allows receiving reference trajectory from another robot. The detail of the system structure is discussed in the Section 3.3.3.

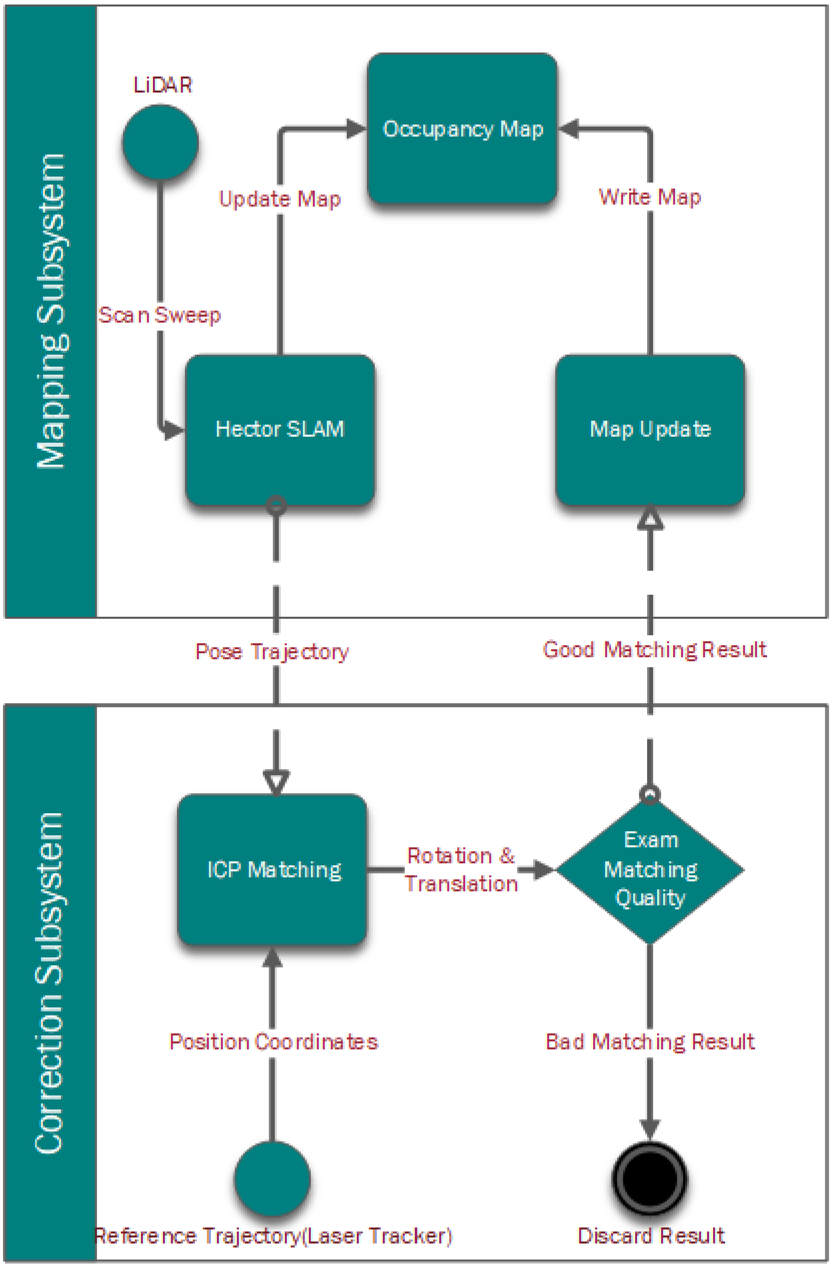


Figure 3.12 System map of the proposed method.

With the suitable results from the ICP, the translation and rotation are then applied to the Hector SLAM process. By re-orientating the robot pose with the translation and rotation from the ICP process, the system is able to align with the reference frame.

3.3.3 Simulation and experiment

This section contains two parts. In the first part, the RoboCup2011 dataset was used as a simulated environment to validate the proposed algorithm. The second part describes how to apply the proposed method on a handheld unit in an indoor mapping process.

3.3.3.1 Simulation using RoboCup2011 dataset

The concept proposed in this chapter was first evaluated by the RoboCup 2011 dataset. This data set was collected during the rescue robot challenge in RoboCup2011. It contains a GPS trajectory recorded during the real-time field test with readings from a 2D UTM-30LX LiDAR and other sensors. Only the reference trajectory and the LiDAR readings are used in this experiment. Figure 3.13 shows a complete 2D occupancy map constructed using Hector SLAM with the RoboCup2011 dataset. The dataset recorded 260 seconds of LiDAR measurements from a handheld unit travelling through a simulated rescue environments shown in Figure 3.14. The movement on the z-axis is not considered in this experiment.

Figure 3.15 compares the results of the experiments with and without our proposed algorithm. In both diagrams, yellow points represent the reference trajectory, and the green points represent the trajectory from the Hector SLAM node.

From Figure 3.15(a), the original pose of the Hector SLAM was the same as the reference. However, the orientation drifted significantly over-time due to the joggling in the starting phase. On the other hand, Figure 3.15(b) shows the mapping result using the same dataset with the ICP re-orientation activated. In this case, the two trajectories were aligned again after the initial joggling. The two trajectories overlapping on the figure indicates the effect of the correction process.

Figure 3.13 Full Mapping Result of RoboCup2011 Dataset.

3.3.3.2 Experimental study with a handheld device

Extensive experiments were conducted to evaluate the concept proposed in this work. Different from the GPS readings in the simulation, a laser tracker was used in the experiments as the reference frame. Additionally, the handheld tracking unit shown in Figure 3.16 is assembled for this experiment to perform Hector SLAM with a single LiDAR mounted on the top of the unit. The centre of the LiDAR sensor is aligned with the z-axis of the handheld unit with its front facing the x-axis. The cat-eye retro-reflector is mounted on the side of the unit with its centre aligned with the y-axis of the unit. Figure 3.17 shows the complete setup with both the LT500 laser tracker and the handheld unit.

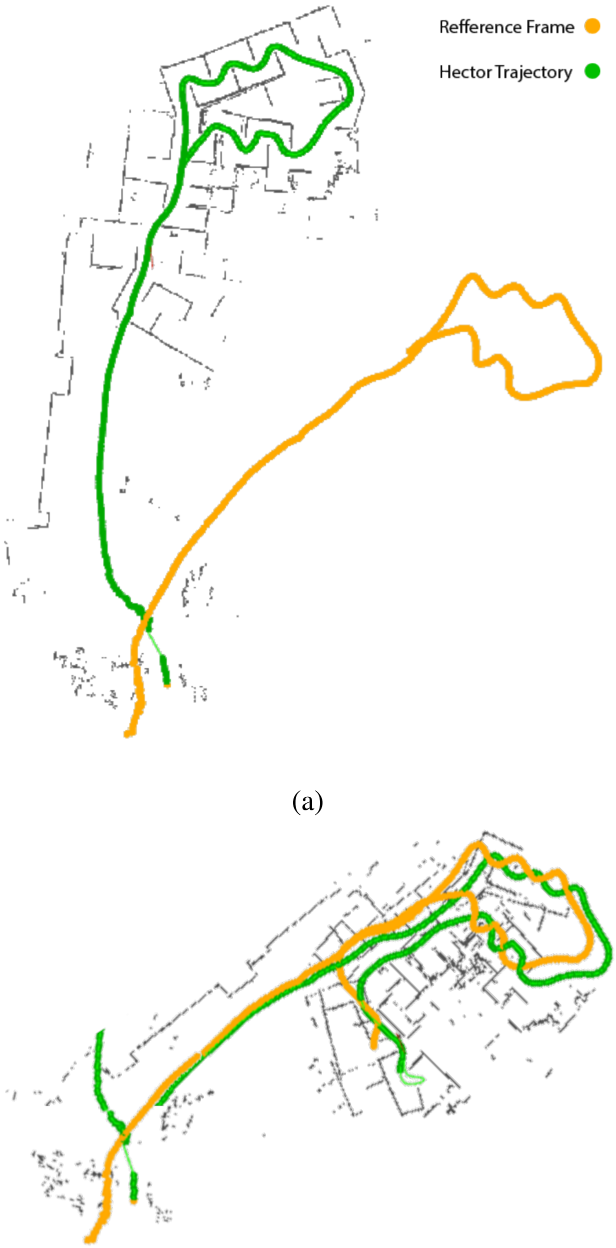
This setup is under the frame of ROS and Point Cloud Library [136] packages. Figure 3.18 shows the overall system structure of this experimental setup. Different nodes in the network are communicating with each other via Wi-Fi.

Leica LT500 laser tracker is capable of generating real-time 3-DOF coordinates of the tracking target. The position of the handheld unit is defined by its geometry centre. Since the extrinsic parameters of the retro-reflector is determined, the laser tracker measurements is used

Figure 3.14 RoboCup2011 Rescue Arena. © [2011] IEEE. Reprinted, with permission, from [Stefan Kohlbrecher; Oskar von Stryk; Johannes Meyer; Uwe Klingauf, A flexible and scalable SLAM system with full 3D motion estimation, 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics, 1-5 Nov.2011]

to represent the trajectory of the handheld scanning unit. During the experiment, the handheld unit starts scanning once locked by the tracker. By providing the initial pose to the Hector SLAM node, drifting can still be observed due to the unstable mapping in the initialisation phase of Hector SLAM. The experiment was carried out in an indoor environment in the RMRL lab. The environment is surrounded by walls, with three sides of the walls straight, and the other two sides in an irregular shape. The room is populated with obstructions with different size and shape. During the experiment, the handheld scanning unit starts the mapping process in a known position and orientation. This pose is captured by the laser tracker, and marked as the origin of the laser tracker coordinates system, with the front of the mapping unit pointing to the positive direction of the x-axis and left pointing to the positive direction of the y-axis.

Figure 3.19 shows the two trajectories recorded by the Hector mapping unit and the laser tracker. As previously explained, the two trajectories were both started in the origin of the coordinates system. However, the drift of the Hector algorithm is significant, as shown on the mapping result where the Hector trajectory (Green) is separated from the laser tracker trajectory (Yellow).



(a)

(b)

Figure 3.15 Comparison of Different Trajectories From RoboCup2011 Dataset.

The improvements is shown in Figure 3.20, where the system pose is aligned using the reference trajectory. It is worth noting that the first bit of the reference frame does not fit into the Hector trajectory as the ICP re-orientation process is iterating to find a suitable match. The experiments prove the concept proposed in this chapter which uses a reference trajectory to correct the system state during the initialisation of a SLAM process.

Figure 3.21 illustrates the detail of the trajectory matching process. Dots (Yellow) on the

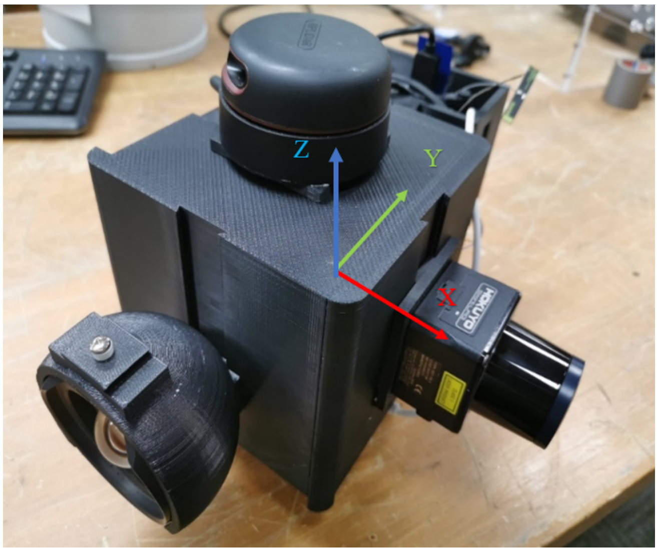


Figure 3.16 The handheld Unit for Hector SLAM and its sensor configuration.

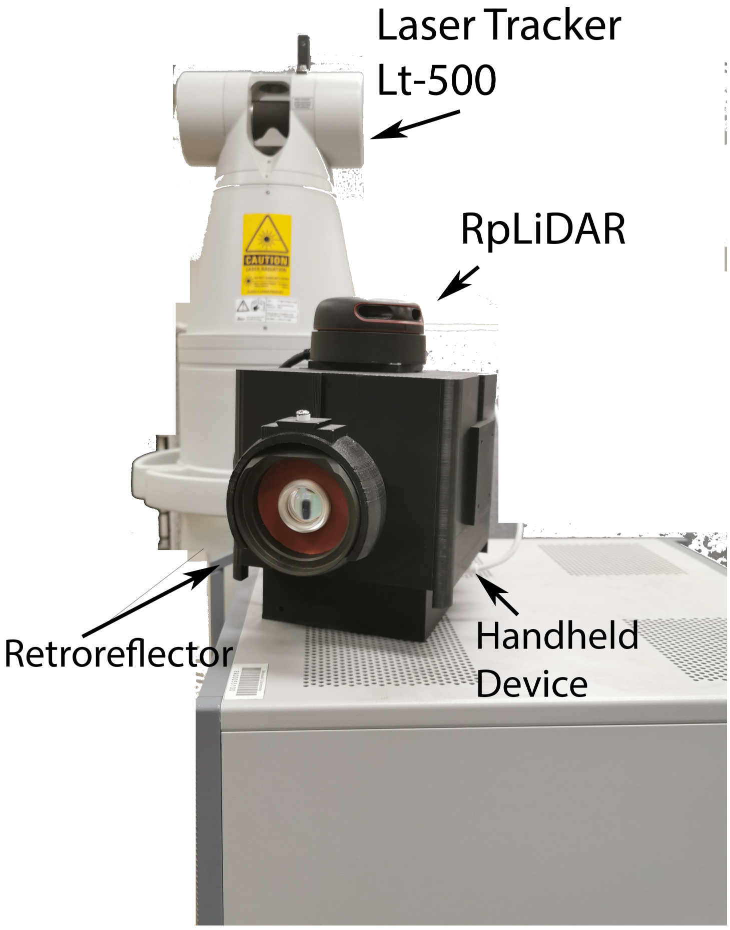


Figure 3.17 Leica LT500 Laser Tracker as a Reference Measurement System.

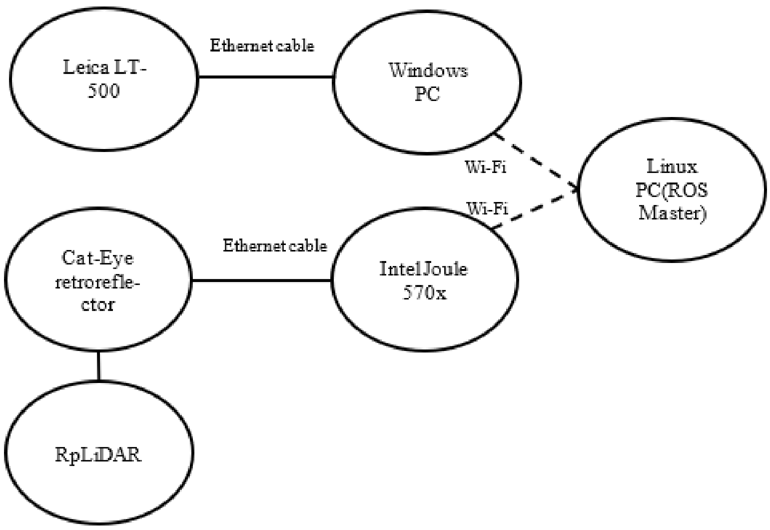


Figure 3.18 System Communication Under the ROS framework.

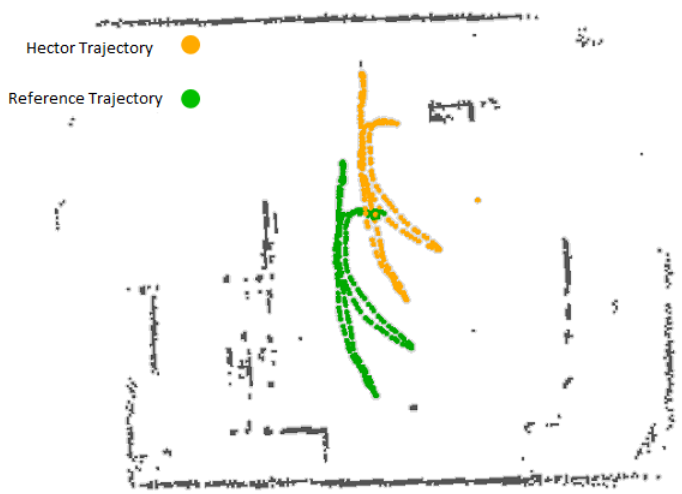


Figure 3.19 Trajectory Translation and Rotation without ICP.

figure marked the trajectory of the Hector algorithm. The lines (Blue) represents the trajectory from laser tracker. In this experiment, the Hector unit is updated pose information in 9 Hz, and the laser tracker is updating at 40 Hz. A downsampling filter is applied to the laser tracker measurements to refine the point numbers before providing to the ICP process.

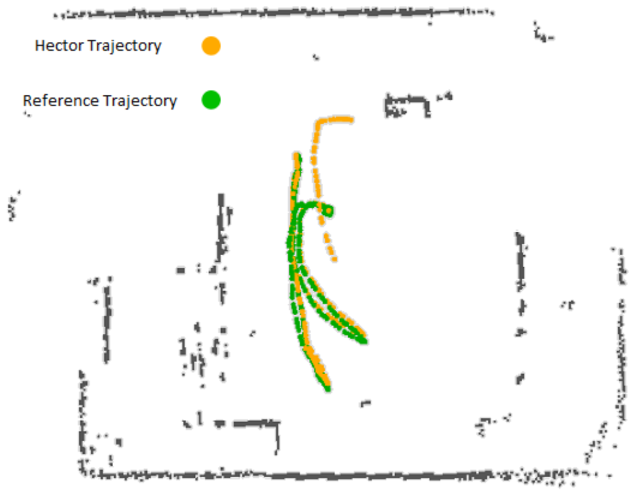


Figure 3.20 Trajectory Translation and Rotation with ICP.

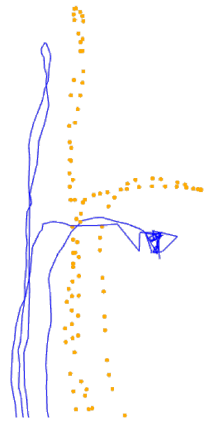


Figure 3.21 Detail of the trajectories from Leica LT-500 and Hector SLAM.

3.4 Discussion

A method which allows the Hector SLAM to identify its positioning error during the starting stage and perform correction was proposed in this chapter. An ICP based alignment technique has been developed which can estimate the displacement and rotation between the SLAM system pose and its preset initial value. A correction method has been developed to be embedded into the Hector SLAM process to correct the system pose after initialisation. The experimental results demonstrated the performance of the method with Hector SLAM. Since the proposed method directly modifies system states through the recorded grid map, it also has the potential to be applied into other SLAM approaches such as Gmapping and Google cartographer. Furthermore, the proposed method uses trajectory information to estimate the system state. The investigation have demonstrated its application in 2D SLAM. However, it is believed that this method could also be applied to 3D SLAM to solve similar problems.

The study documented in this chapter have also discovered some limitations of the proposed algorithm during the experiments. Even though the method proven to be able to correct system pose errors during the initialisation, it does not fix the mapping errors recorded on the map. The recorded mapping errors threat the stability of the mapping process in the future. Additionally, the methodology presented is a one-time process which does not help improve the SLAM process in the long term application. The correction result replaces the system pose in realtime, thus causes mapping blur that affects the mapping result. The above limitations will be further discussed and improve in the next chapter.

Chapter 4

High Robustness SLAM using Reference Trajectory Information [1]

4.1 Introduction

This chapter presents an Interactive Trajectory Matching (ITM) approach to further enhance mapping robustness of a SLAM approach. The ITM method is extended from the method proposed in the previous chapter. The previous chapter have described an algorithm to help Hector SLAM to realign its system pose according to a reference frame. This concept was evaluated using a laser tracker as the source of the reference trajectory. With the proposed trajectory matching algorithm, it is possible to identify the geometrical relationship between a reference trajectory and the SLAM trajectory. The geometrical relationship was used to improve the instability of the SLAM process during the initiating period. The ITM method presented in this chapter builds upon the trajectory matching technique. Instead of only focusing on the stability of the system pose during initialisation, the ITM method expands this concept to the entire mapping lifecycle of a SLAM algorithm. The proposed method iteratively seeks matches between a reference trajectory and the SLAM trajectory. It corrects the system state when necessary. This correction is not limited to the system pose. The ITM method is also designed to

[1] The works contained within this chapter have previously been published in: [C2]

correct existing mapping results. Furthermore, it is demonstrated that the proposed ITM method can be adapted into different SLAM approaches, in both 2D and 3D environments.

4.2 Using trajectory information for mapping correction

The stability of a SLAM algorithm is a serious concern to the industry. To partially address this problem, the recent development of Visual Odometry (VO) and SLAM approaches adopted the technique of splitting the mapping processes into different levels of mapping frequency and accuracy. These approaches are often constructed in a multi-level architecture, where the mapping state is updated in isolated steps:

- 1) **Local Frame Matching** - using the current LiDAR or camera reading to compare with the previous reading, and thus estimate the new system state based on the previous. This step is often easy to compute but suffer from high drift accumulation.
- 2) **Local to Submap Matching** - taking the current reading and comparing to a submap. The submap is chosen by a sliding window centred at the current system state. This step greatly reduces drift accumulation but is limited by its computational complexity.

The combination of these layers requires the LiDAR or camera scans the environment as frequently as possible to maintain low drifts. A higher sampling rate will generate a more accurate system orientation, as the matched features are closer between frames during movements. As a result, Global Navigation Satellite System (GNSS), Ad-Hoc beacons or other absolute localisation techniques are rarely seen in these approaches due to their high noise distribution in a short period. In addition, these sensors only provide system poses in 3 degrees of freedom (DOF). Lack of orientation estimation makes fusing these readings for the system meaningless.

However, absolute positioning systems feature high robustness. More importantly, they directly translate the system state to the global frame. These sensors do not rely on the previous system state to estimate the current system state, and therefore, the errors are not accumulative. It has been demonstrated that combining absolute positioning techniques into the SLAM process could improve the robustness of the algorithm, and helps the system to overcome significant mapping failures during initialisation. This chapter is focusing on improving the accuracy and robustness of the SLAM result throughout the mapping lifecycle. This work builds upon the concept of the multi-level map updating strategy and uses the proposed ITM method to intercept the map update lifecycle, thus correcting the system state. The main contributions of the work described in the chapter are:

- **A trajectory matching strategy** that combines the advantages of both high-frequency iterative positioning and high-robustness absolute positioning. Other than frame-to-frame accuracy, our method seeks to improve the overall mapping robustness by interpolating the system state based on readings from a secondary trajectory frame.
- **A mapping results correction methodology** which is based on the outcome of the trajectory matching process. The proposed approach took the transform information between the two frames and used it to correct the mapping results. A sliding window was used to control the size of the active area that participate in the correction, thus reducing its computational complexity.
- **Loosely-coupled mapping process** makes the proposed method insensitive to the environment. While most of the absolute positioning systems, especially GNSS, faces reception problems in mixed environments, the proposed method does not require a consecutive reading. Instead, the algorithm samples the trajectory received iteratively and only output estimations when it finds a high-quality match.
- **An adaptable structure** which does not affect the mapping process of a SLAM approach. Instead, the proposed methodology works as an add-on of a targeted SLAM algorithm. The results in the experiment section have demonstrated the ability to adapt the proposed method into different SLAM frameworks. The application of the proposed ITM method varies from 2D grid map SLAM to 3D point cloud SLAM. The study indicates that the proposed method significantly improves the quality of the mapping results in all tests.

The method proposed in this section aims to fit into an existing SLAM approach and improve its performance in view of drift accumulation. In this study, Hector SLAM and LOAM were selected as the targeted SLAM algorithms to validate the proposed methodology. The proposed ITM method requires a SLAM to generate two independent trajectories. The first trajectory being the trajectory of the SLAM algorithm, and the second trajectory being the trajectory of a reference frame, such as a GNSS or a laser tracker. The concept of the ITM method is based on the expectation that the reference trajectory has better robustness compared with the SLAM trajectory. The geometrical similarity between the two trajectories was used to examine the quality of the SLAM process. The proposed method monitors the difference between the two using an approach similar to the point-to-point ICP. Once the difference exceeds a threshold, a matching process will be triggered to align the orientation of the SLAM trajectory according to the reference frame.

Overall, the proposed ITM method uses the Mean Squared Error (MSE) between the two trajectory points as the cost function. Using the Levenberg-Marquardt (L-M) method, the convergence translation and rotation between the two trajectories as the desired transformation from the current system posture to the corrected system posture. Then, transformation is applied to posture and map in the SLAM module to complete the correction. Figure 4.1 presents the process of the proposed ITM method using a flow chart. While the ITM module is receiving posture information from both the reference frame and the SLAM frame, the algorithm is executed periodically based on a timer.

4.2.1 Trajectory alignment

This section explains the workflow of the proposed ITM algorithm. The alignment of trajectory orientations is performed using the method developed based on the previous work [C1]. The previous work was focused on finding the best match between trajectories from the starting phase of the SLAM. This process only needs to trigger once during the entire SLAM process. However, in this work, the trajectory matching is extended from a one-time process to the entire system lifecycle as an iterative process.

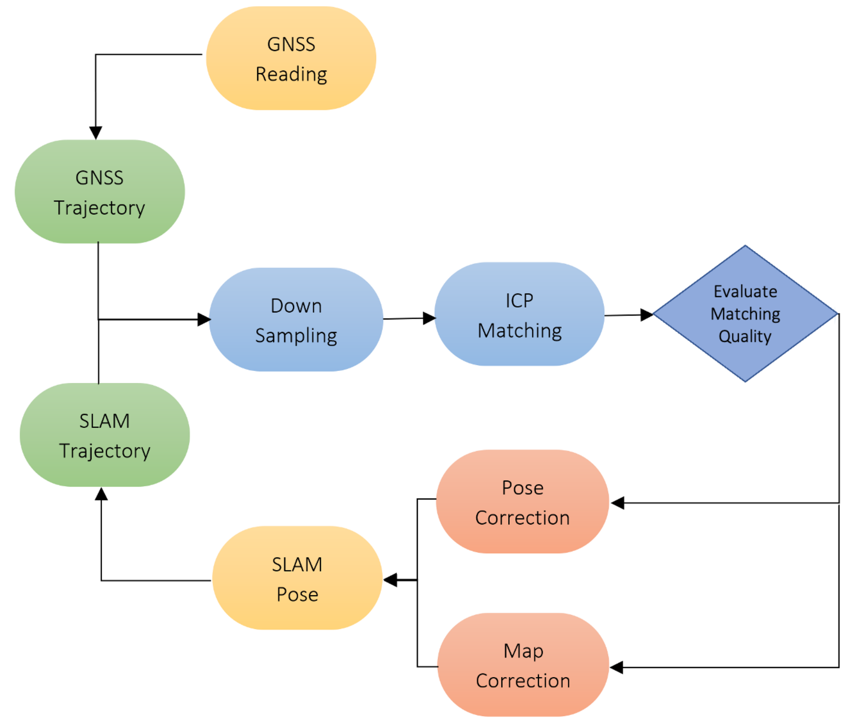


Figure 4.1 Schematic of the Proposed ITM Process.

Specifically, the trajectory matching process in this work is a periodical process which continually compares the the Euclidean distance difference between the reference trajectory and the SLAM trajectory. Iteratively matching the trajectories contribute to the ITM algorithm in two aspects. Firstly, this allows incontinuous reading from the reference trajectory. Secondly, the iteration allows continuous checking of the system state. A correction can be invoked throughout the mapping process whenever an error is detected.

Figure 4.2 shows an example of the trajectory alignment process where the SLAM trajectory is matching the reference trajectory. The ITM outcome is shown in the dashed line, which overlaps with the reference trajectory. This indicates a match where the residual of the L-M method exceeds the threshold. The rotation and translation are then applied to the SLAM system pose, and the results in the pairing of the SLAM trajectory and the reference trajectory are shown on the right side of the figure (red dash line).

The matching between two trajectories can be described by Equation (4.1). Let L and H be the two sets of coordinates from the trajectories of the reference frame and the SLAM during period T. A downsampling filter is required to ensure both trajectories have the same number of

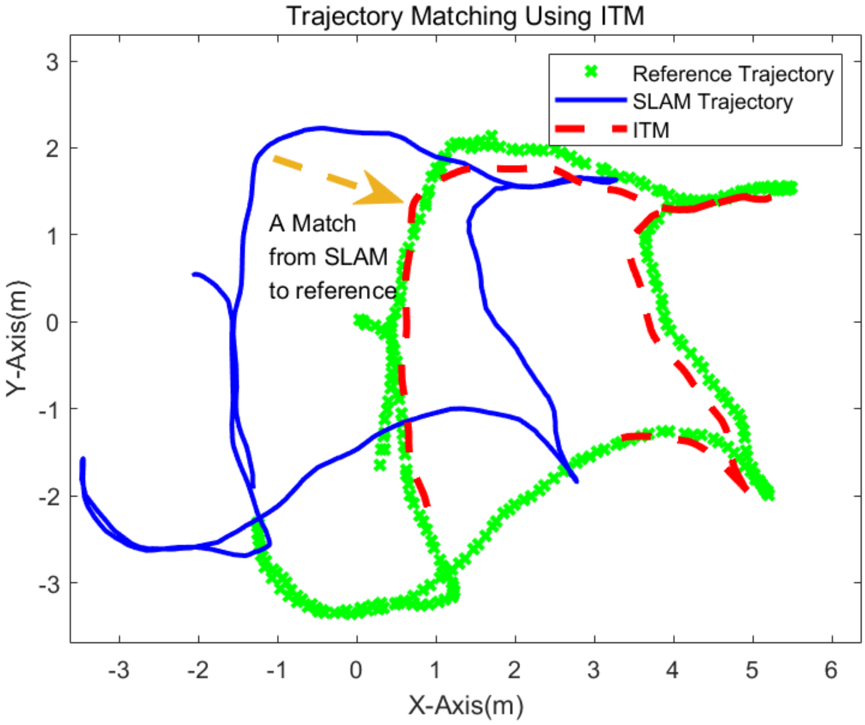


Figure 4.2 An example of trajectory matching using ITM.

points where $L = \{l_1, \ldots, l_n\}, H = \{h_1, \ldots, h_n\}$.

Setting an initial value is important for all optimisation methods. In this case, the best guess of the initial rotation in quaternion is $w = 1, x = 0, y = 0, z = 0$ and translation is $x = 0, y = 0, z = 0$. This is due to the fact that the two trajectories should be close to each other, assuming the mapping function works correctly.

$$E(q,tr) = \frac{1}{N_l} \sum_{i=1}^{N_l} \|l_i - q \cdot h_i - tr\|^2 \tag{4.1}$$

Using Equation (4.1) as the cost function, the matching provides a pair of rotation q^* and translation tr^* as the outcome if the L-M method converges with a relatively small residual. To avoid unnecessary correction, $q^*$and tr^* need to exceed a preset threshold. Thresholding here is to avoid corrections when the drift is not severe, and the correction is not necessary.

4.3 Map and posture correction in 2D SLAM

Adapting the proposed algorithm into 2D SLAM requires modification of both the system pose and the grid map updating operation. Using Hector SLAM as an example, with $q^*$and tr^* found in Equation (4.1), it is possible to correct the system pose using the approach described in the last section. The next step is to identify the current system state and the grid cells on the map that require correction. In Hector SLAM, the system posture is recorded as $P = \{p_x, p_y, \psi\}$, where ψ indicates the current system orientation. Converting the quaternion, $q*$, in Equation (4.1) to a rotation matrix R allow the methodology to directly update the system into a new posture using Equation (4.2).

$$P = \{(p_x + tr_x^*), (p_y + tr_y^*), \psi \oplus \psi_R^*\} \tag{4.2}$$

Each sweep $S_i(\xi)$ is recorded on the map using Equation (4.3).

$$I(S_i(\xi)) = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} R^* \begin{bmatrix} S_{i,x} \\ S_{i,y} \end{bmatrix} + \begin{bmatrix} p_x \\ p_y \end{bmatrix} + Tran^* \tag{4.3}$$

For a collection of grids $G = \{g_{t_1}, \ldots, g_{t_n}\}$ where $\{t_1, \ldots, t_n\} \in T$, a correction of G can be written as:

$$I(G) = R^*(G) + t^* \tag{4.4}$$

After calculating the grid location, the ITM method updates the grid map with new grid coordinates and frees the original grids. Once the map has been restored, apply R^* and T^* to the current pose will relocate the pose to a new position and orientation that is corresponding to the corrected map. From there, the Hector SLAM algorithm will estimate the next system pose based on the corrected grid map and pose.

Various parts of the original Hector SLAM need to be modified to work with the proposed ITM method. The modification involves both the map representation and the pose generation process. These modifications are explained in the following sections.

4.3.1 Threshold and correction

The proposed ITM method is executed periodically. Each execution is triggered after the algorithm have collected enough number of trajectory points. However, not all the results can be used in the correction process as the quality of the convergence varies considerably. Therefore, a threshold to judge the quality of the matching result is used.

The core of the ITM method features a Point-to-Point ICP (P2PICP) algorithm. The proposed implementation of P2PICP uses L-M algorithm to optimise the Euclidean distance between the two trajectories (Equation (4.1)). The optimisation targets are the rotation and translation from the SLAM trajectory to the reference trajectory. The optimisation is limited to 25 iterations, and the quality of the match is judged based on the value of Euclidean-distance-per-trajectory-point. Assuming the Euclidean distance between trajectories after iteration is i, the quality of the optimisation r is i/p, where p is the number of points in the trajectories after downsampling.

It was found that setting the threshold to 10 gives the best results during the laser tracker experiments. Increasing the threshold helps the algorithm to compatible with the sensors that have more extensive noise distribution. For example, increasing threshold to 15 provide better results during a single GPS module test. Map correction is only triggered if the convergence residual meets the threshold. Otherwise, if the residual did not exceed the threshold, the process will be cancelled with all the parameters reset to the default values. Every iteration of the ITM method will repeat the above steps.

It is worth noting that the ITM method does not require a constant reading from the reference frame. Instead, the algorithm starts each new iteration by collecting real-time pose points. As shown in Figure 4.3, the ITM algorithm works as an add-on to the original SLAM algorithm. This allows the algorithm to tolerate discontinued reference trajectory readings. For example, when using a GPS as the reference trajectory, ITM skips the period when the GPS reading is not stable or lost, and thus restart the matching process once the measurements can support a healthy estimation. This feature is particularly useful in urban scenarios, where the robot has dynamic access to difference sensors while travelling between the indoor and outdoor environments.

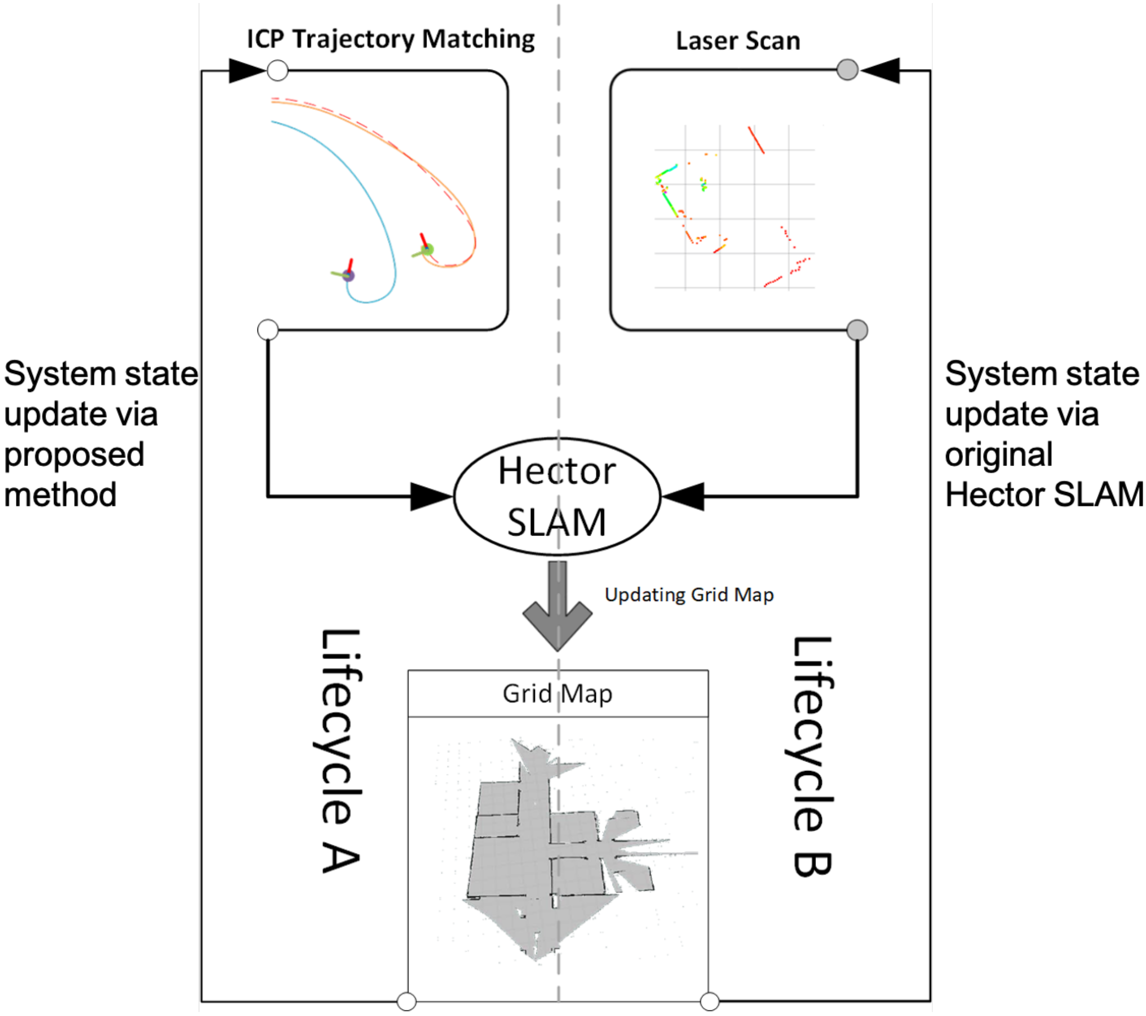


Figure 4.3 ITM lifecycle add-on to the Hector SLAM algorithm.

4.3.2 Grid map with timestamp

The regular grid map only records two attributes, the location coordinates of the cell and an occupancy value of that cell. These two attributes represent the probability of an obstacle that exists in a set of real-world coordinates. The algorithm designed in this study aims to restore a section of mapping results according to a given translation and rotation. In order to achieve this, the algorithm needs to identify those grids belong to a certain period of the mapping process. The sets of grids updated during T are timestamped with t where $t \in T$. Combining the timestamp from both the trajectory points and the grids provides the functionality of selecting grid cells that belong to a certain period of SLAM process.

The methodology in this work introduces timestamps into each grid on the map. In addition to its occupancy value, a queue of timestamps is stored within the cell. These timestamps indicates the time of the most recent updates for a grid cell. The length of the queue affects the number of timestamps saved. It can be adjusted according to the frequency of the LiDAR scanner, the map resolution and the mapping environment. In addition, each grid cell also stores a bool flag to record its correction state. A cell is 'Shifted' means the cell is corrected by the ITM algorithm already on the map. The bool flag here is to prevent a cell from being involved in more than one correction.

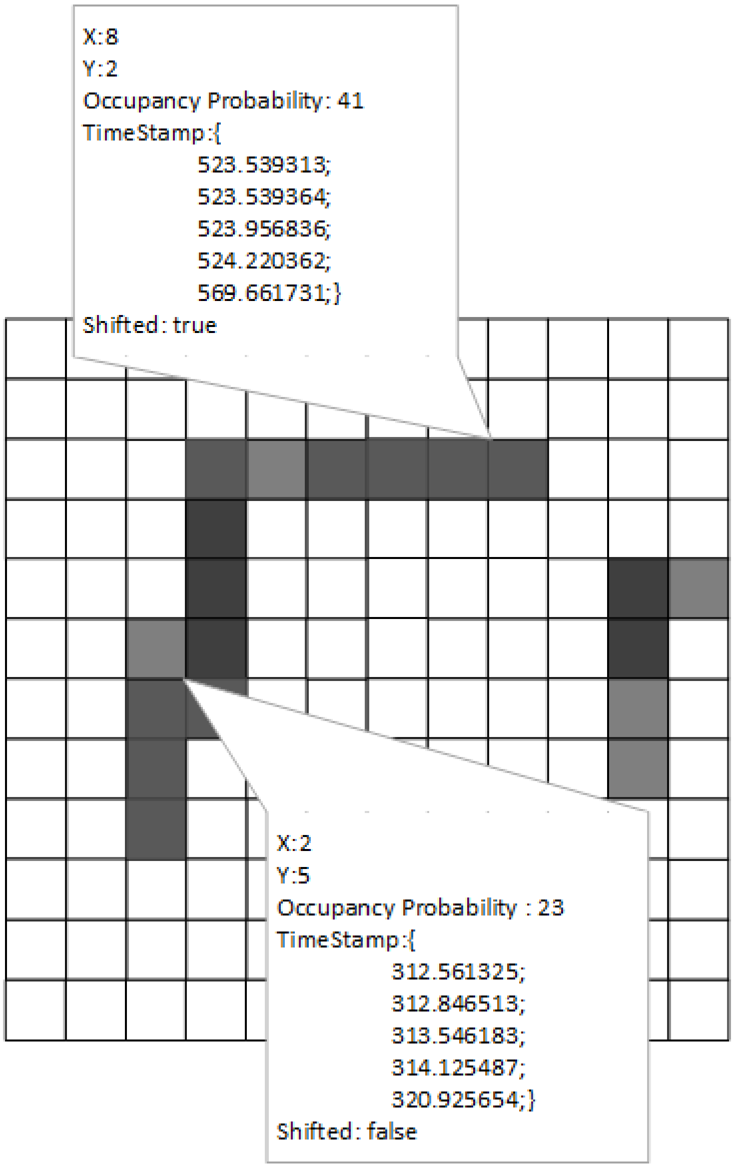


Figure 4.4 Using timestamp to help selecting grid cells on the map.

Using Figure 4.4 as an example, with the top-left corner as the origin of the map coordinate system (0,0), the grid cell (8,2) indicating a cell on the grid map that 8 cells to the right and 2 cells to the bottom from the origin of the map. The occupancy probability indicates the likelihood of having obstacles in this position, which in this case is 41. The cell was last updated by the LiDAR at system time 569.661731. Since the 'Shifted' state is true, the cell has been corrected by the ITM algorithm once already, thus it needs to be ignored by future corrections.

4.4 Correction window and computational complexity

The selection of grid cells with timestamp is not based on the entire grid map. Instead, to speed up the process, the correction is based on a subset of the grids. A sliding window is made according to the centre of the matching trajectory. Let (Mx, My) be the mean value of a given section of the trajectory. Defining the map resolution as *res*, the LiDAR range is d. The selection window of grid cells (x, y) is :

$$\begin{aligned} &(\frac{Mx}{res} - \frac{d}{res}p) < x < (\frac{Mx}{res} + \frac{d}{res}p) \\ &(\frac{My}{res} - \frac{d}{res}q) < y < (\frac{My}{res} + \frac{d}{res}q) \end{aligned} \tag{4.5}$$

p and q is a pair of window size parameters which can be changed to better fit the selection window size for a particular LiDAR scanning range.

Equation (4.6) shows the translation from Equation (4.5) scaled into the grid map coordinate system.

$$\hat{tr} = \frac{tr}{res} \tag{4.6}$$

With the selected grids and the scaled translation, for a collection of the grids $G = \{g_{t_u}, \ldots, g_{t_v}\}$ where $\{t_u, \ldots, t_v\} \in T$, a transform of G can be rewritten as:

$$I(G) = R \cdot G + \hat{tr} \tag{4.7}$$

Searching window is defined by the LiDAR scanning range. In most of the indoor scenarios, the LiDAR scanning range is limited to 10 meters, regardless of the working range of the sensor.

Table 4.1 Number of the cells with different map resolution and LiDAR range.

Map Resolution $Grids/m$	0.1	0.05	0.025	0.01
RpLiDAR A1 10m	100^2	200^2	400^2	1000^2
Hokuyo UST-20LX 100m	1000^2	2000^2	4000^2	10000^2

Table 4.2 Computational complexity comparison with wall-time recorded.

Map Resolution ($cell\ length$)	0.1	0.05	0.025	0.01
RpLiDAR A1 10m	$1.328ms$	$9.551ms$	$1059.412ms$	$8711.688ms$
Hokuyo UST-20LX 50m	$1.990ms$	$190.512ms$	$15702.359ms$	N/Ams

The range limitation is much higher in outdoor scenarios. However, since the accuracy of the LiDAR sensor reduces as the distance increases, the scan reading range is often limited to half of the sensor's physical limitation. Table 4.1 listed the number of cells included in the search window with the relationship between LiDAR reading range and map resolution. These tests use RpLiDAR for indoor mapping and Hokuyo UST-20LX for outdoor mapping. It is obvious that the total number of the cells in the window grows exponentially with the increase of LiDAR range and map resolution.

4.4.0.1 Rolling shutter problem in grid map correction

The extensive search range introduces the problem of high computational complexity. The ITM needs to traverse all cells in the searching window to find the transform candidates. The computational complexity of the ITM process is $O(m(d*r)*n)$, where m and n are the length and width of the searching window, respectively, and d and r are the scanning range and map resolution, respectively. Hector SLAM updates the grid map 'on the fly'. New grid cells are still updated according to the existing system pose until the system receives a new pose from the ITM algorithm. The time gap between map correction and position correction will create a rolling shutter problem on the map, as the newly updated cells are not included in the correction.

Table 4.2 follows the same structure as Table 4.1. Instead of comparing the number of cells, the wall-time of the actual map correction process with different configuration was compared. From the table, the time consumed during each ITM match grows exponentially. During the test,

Table 4.3 Computation wall-time recorded with different scanning range and map resolution.

Map Resolution (*cell length*)	0.1	0.05	0.025	0.01
Hokuyo UST-20LX 6m	$13.55ms$	$204.81ms$	$982.38ms$	$2405.13ms$
Hokuyo UST-20LX 15m	$637.92ms$	$4171.33ms$	N/A	N/A
Hokuyo UST-20LX 30m	$9200.12ms$	$48112.77ms$	N/A	N/A

the ideal processing time depends on the travel speed, the data rate of the two trajectories and also the map resolution. In the testing environment, it was found that a good match requires both trajectories containing more than 30 pose points. Assuming Hector SLAM generate 3 DOF pose information in 8Hz, collecting 30 pose point will take about 4 seconds. With the SLAM unit travelling at 0.75 meters per second, 4 seconds will cause a 3 meters of displacement. Define the search window as a 23*23 meters square box is sufficient to cover all the gird cells that likely to be updated by the LiDAR in the past 4 seconds. This configuration was tested with Hokuyo UST-20LX with 4 different sensing range limitations. The goal of this test was to find a suitable configuration for the ITM algorithm to operate smoothly.

A different set of tests were conducted to evaluate the loop time for one iteration of the ITM. Since the map is updated 'on the fly', a long processing time will affect the map quality. Listed in Table 4.3, only certain combinations of LiDAR range and map resolution is able to complete each iteration in a relatively reasonable amount of time. Iterations taking longer than $1000ms$ will cause a significant rolling-shutter problem in the mapping results.

4.4.0.2 Pose updates with ITM in Hector SLAM

Hector SLAM uses multi-layer maps to overcome a local minima problem in the scan-matching. As described previously, the deeper the map level, the lower the map resolution. Depending on the user scenario, map levels can be set between 1 to 3 for the best mapping results. The map level is related to the proposed ITM algorithm because it is needed to be applied to all levels of the map to correct the mapping results. In the implementation, both the selection window and system pose are scaled to the corresponding resolution to correct mapping results in all exist mapping levels.

In theory, this would further burden the calculation. However, in practice, the number of cells decreases exponentially as the map resolution reduced, which makes the processing time of map correction mainly related to only the first level of the map.

4.4.0.3 Map updates with ITM in Hector SLAM

Each cell on the grid map uses a log-likelihood to represent its probability of having an obstacle in its location. The map correction process involves translating and rotating grid cells to new locations. In the implementation, this is achieved by copy the log-likelihood of the original cell the destination cell and reset the original cell.

4.4.1 Indoor experiment with Hector SLAM

To validate the correction effect of the proposed ITM method, the first set of experiments was took in the first floor lab in RMRL. The device constructed for this experiment is shown in Figure 4.5, which mainly features a RpLiDAR A2 and a Leica laser tracker LT-500. The handheld device, with the LiDAR on top and a CatEye Reflector on the side, is able to scan the lab space in 2D. A laser tracker was placed in the middle of the lab to track the CatEye Reflector. The room environment is full of laboratory equipment and two glass walls (Figure 4.6), which challenges the Hector SLAM from three aspects: the low resolution of RpLiDAR, the arbitrary obstacles in the space, and the two glass walls that generate noise in the LiDAR (Figure 4.6(c)(d)). The room is about 11 meters long, 7 meters wide. During the experiments, the handheld device remained about 1.3 meters above the floor and travelled through the room. The device visited some corners in the room and experienced a few sharp turns during the experiment. The device was deliberately pointed to the transparent wall during the test to simulate LiDAR degradation situation. In the end of the experiment, the device eventually returned to its starting position.

Figure 4.7(a) shows the mapping results from native Hector SLAM algorithm. The green dash line indicates the trajectory estimated by Hector SLAM. A noticeable mapping error can be seen on the left side of the map as the handheld unit travelling close to the glass wall. The interference results in a misaligned wall on the map shown on the left side of the figure.

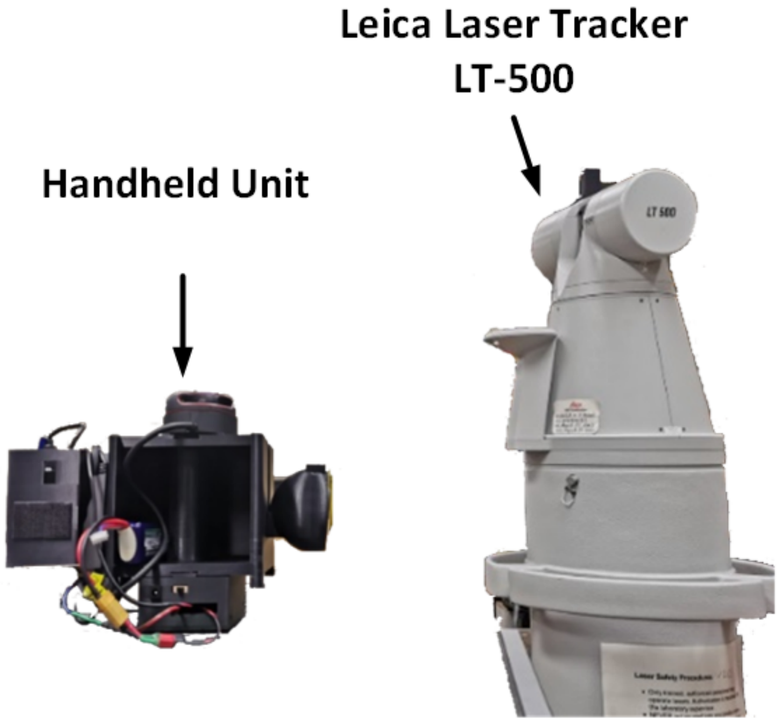


Figure 4.5 The handheld device with laser tracker.

Figure 4.7(b) illustrates another mapping attempt with the ITM involved. Comparing with the results in Figure 4.7(a), the ITM algorithm successfully detected the misalignment of the Hector trajectory and the reference trajectory. A match was found between trajectories in the circled area, which indicates the ITM process successfully restored the misaligned wall on the left side of the map. While correcting the posture of the system, it also transforms the grids inside the sliding window. Some minor mapping error can still be seen around the left wall on the map as the effect of the rolling shutter problem discussed in Section 4.4.0.1. The effect can be minimised by tuning the ITM to fit the application scenario. Specifically, in this test, setting the trajectory matching iterates to every 10 seconds with 100 trajectory points from both Hector and laser tracker provides the best correction results. The searching window is defined as 6X6m^2 with first-level map resolution set to 0.025. The RpLiDAR module was updating measurements at 8 hz with 4000 points per update. During the test, the average processing time of each ITM iteration is about 176ms.

In the experiment, the ITM algorithm only triggered once as the threshold is set relatively high. Figure 4.9 shows a detailed illustration of the section of trajectories that involved in the

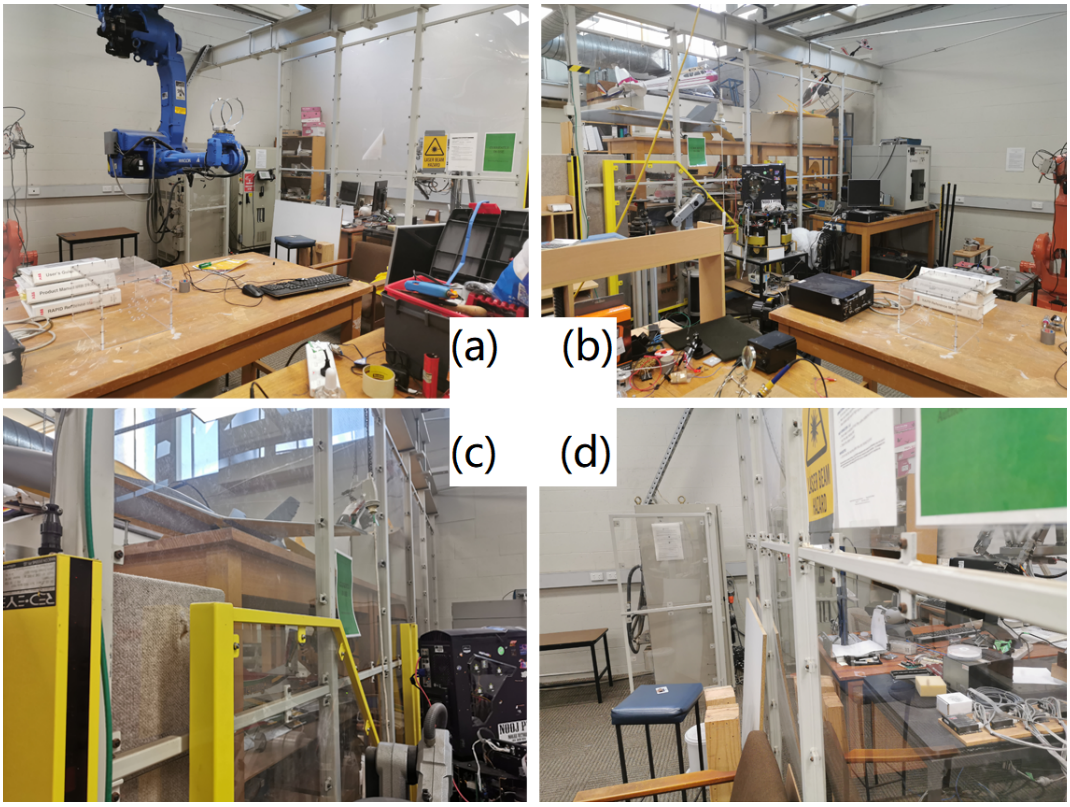
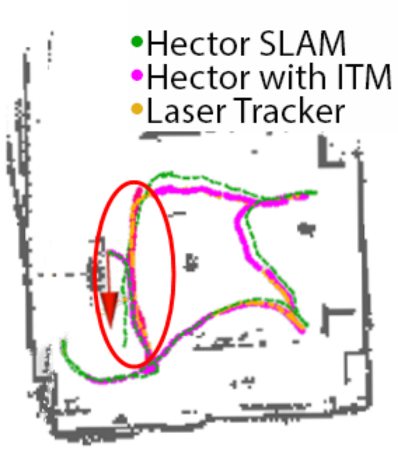


Figure 4.6 Experiment environment in first floor of RMRL (a) obstacles in the room. (b) obstacles with transparent wall. (c) a transparent wall in the middle of the room (d) another transparent wall in the lab.

(a) Native Hector SLAM Process.



(b) ITM interpolated Hector SLAM Process.

Figure 4.7 2D Experiments in RMRL lab. (a) Original Hector SLAM mapping result. (b) ITM with Hector SLAM mapping result.

correction process circled in Figure 4.7(b). It is indicated in the figure that Hector trajectory is aligned to the laser tracker trajectory.

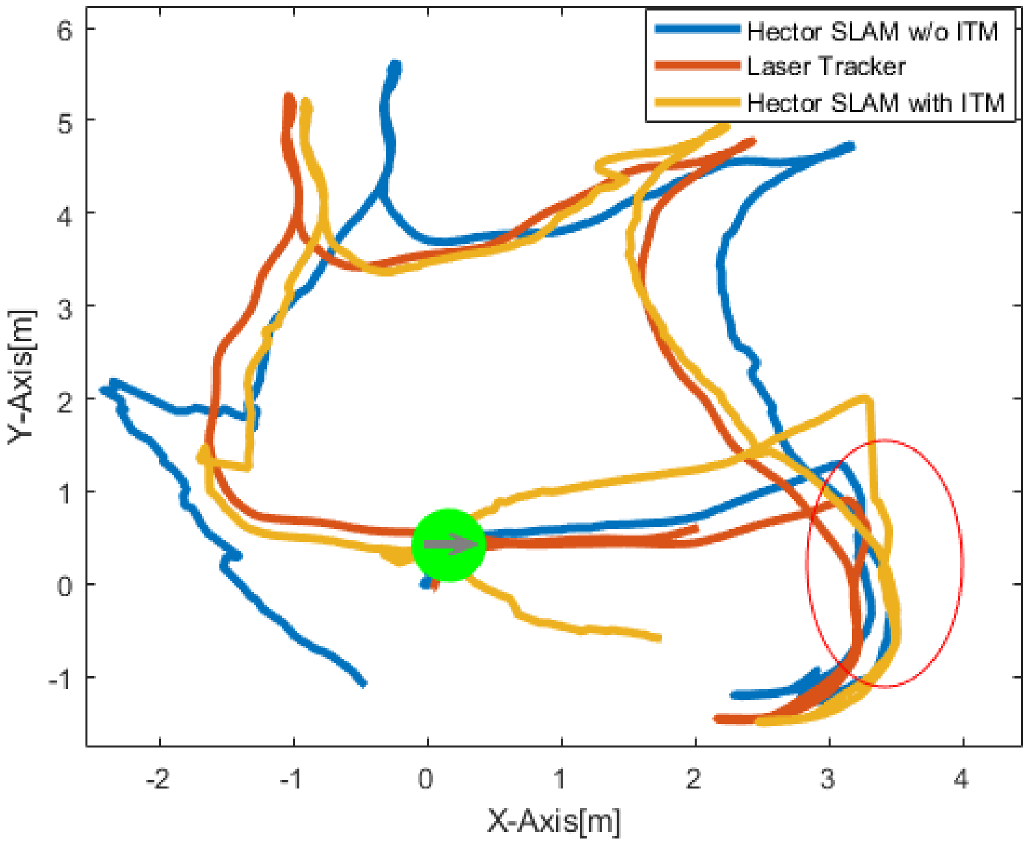


Figure 4.8 Trajectory comparison for ITM and original Hector SLAM.

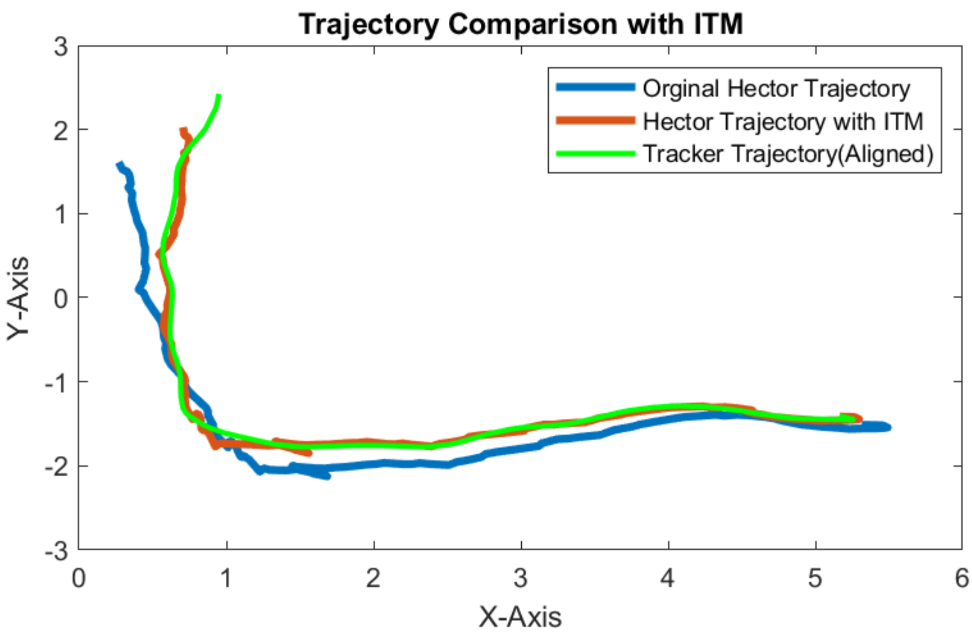


Figure 4.9 A zoom-in on the section of trajectories in the active ITM process.

The detailed performance of ITM is shown in Figure 4.10 where X and Y axes are compared separately. The effects of an ITM process is highlighted in green boxes on this diagram. The trigger threshold for an ITM in this experiment was set to 0.1 meters. This means when comparing the Hector SLAM trajectory with the laser trajectory, an ITM is activated if the translation between the two is greater than 0.1 meters.

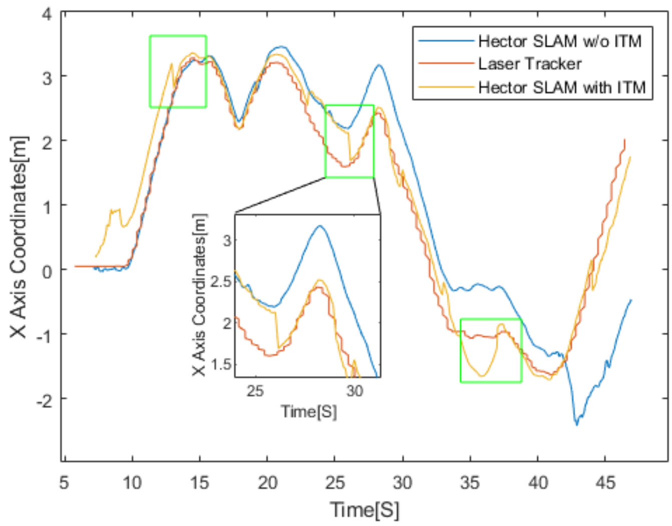
As a result, green boxes in both Figure 4.10(a) and Figure 4.10(b) record three triggered ITM processes. It helps the SLAM process to relocate its system pose estimation to a more reliable estimation. The deviations highlighted in boxes on the yellow line here indicates relocation processes of the system state. It can be observed that each ITM process significantly reduces the gap between the Hector SLAM trajectory and the reference trajectory.

The laser tracker provides position estimations far more accurate than the Hector SLAM algorithm. This study uses Position Difference (PD) to examine the errors of Hector SLAM trajectory with and without ITM compared with the laser tracker trajectory. In Equation (4.8), x and y are the position estimations in two Hector trajectories. $\hat{x}$ and $\hat{y}$ are the position estimations from the laser tracker. $k(t)$ here is the time vector of the mapping process. The result of the PD analysis is shown in Figure 4.11. It is worth noting that the PD of position estimation from Hector SLAM with ITM is significantly more stable and accurate than the one without ITM.

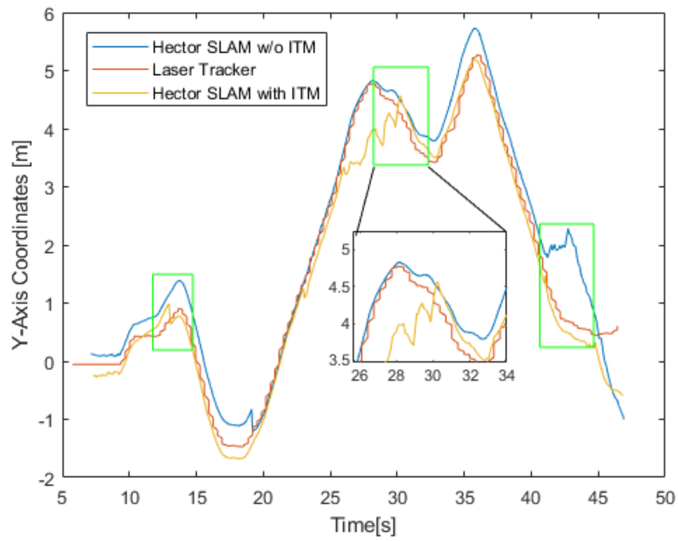
$$\mathrm{Pd}(t) = \sum_{i=1}^{k} \sqrt{(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2} \tag{4.8}$$

4.4.2 RoboCup 2011 rescue arena dataset

The aim of this experiment is to validate the consistency of the ITM method for the middle to large scale mapping implementation. The ITM method should be able to identify the accumulated drift from a robotic mapping algorithm during the mapping process. In a SLAM process, without a loop-closing method, the errors accumulated during mapping will result in a displacement between its finishing system pose and the system pose in the ground truth. In RoboCup2011 dataset, the mapping device travelled back to the starting point with a total running time of 260 seconds, which brings out any drift during the mapping process. The basic information of



(a) x-axis comparison



(b) y-axis comparison

Figure 4.10 X and Y axes movement compared for Hector SLAM with and without ITM.

RoboCup2011 is described in Section 3.3.3.1. Only the 2D LiDAR and GPS reading were used from the dataset. The total running time is around 260 seconds with a handheld LiDAR device travelled through a simulated rescue environment.

The LiDAR used in the dataset is a Hokuyo UTM-30LX. We have downsampled the LiDAR reading range to 6 meters to make the sensor reading comparable with the sensors we installed on the robot. Figure 4.12(a) illustrates the mapping result of the original Hector SLAM. Two most apparent mapping errors are highlighted in the red boxes, which are both caused by rapid rotation

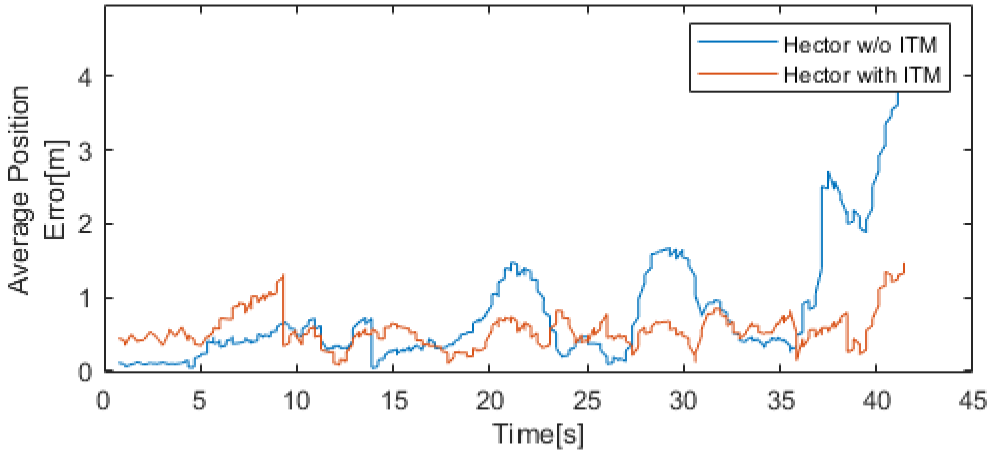


Figure 4.11 Position Difference for Hector SLAM with and w/o ITM.

on the z-axis. The error directly results in two misaligned walls on the map. Additionally, since Hector SLAM uses established grid map to estimate its current system pose, the misaligned cells misled the mapping process and eventually caused large map deformation over the entire constructed map. The mapping error is significant on the left bottom of the figure (in the blue cycle) as the maps are doubled in the result.

Figure 4.12(b) is the simulation based on the same dataset, but with the ITM activated. This time both mapping errors and system pose are significantly improved. The algorithm was able to correctly identify the drift on the map, and relocate the system pose to the corrected location. Compared with the ground truth shown in Figure 4.13, most of the walls are correctly aligned. Since the SLAM process circulated the space with its starting and finishing positions both on the left bottom of the map, this area is heavily affected by accumulated mapping drifts. Comparing Figure 4.12 (a) and (b) with the ground truth map, the proposed ITM method appreciably improved the mapping result.

Figure 4.14 is shows the progress of the ITM method with RoboCup2011 dataset. On the figure, the mapping results are aligned according to the ITM results. The reference trajectory is coloured in orange and the Hector trajectory is in green. Red dot trajectory illustrate the results of suitable ITM matches. Figure 4.15 shows a detailed sample of an ITM process. Once found a match, the system is assigned with a new pose after restoring mapping results. Noting that some parts of the trajectory is not accompanied with the red dots as the proposed algorithm did not

(a) Original Hector SLAM mapping result with errors.

(b) ITM Hector SLAM mapping result

Figure 4.12 Mapping results from Hector SLAM with and without ITM.

find a high quality match in that section.

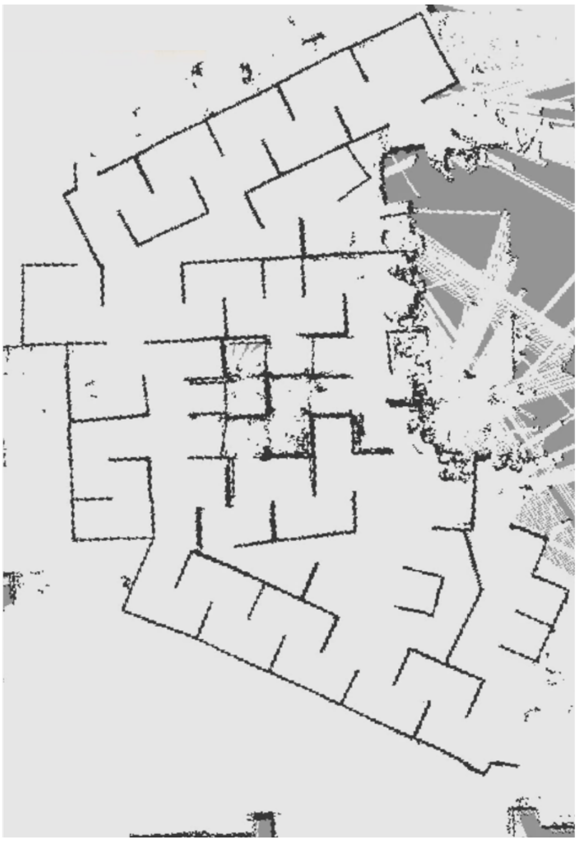


Figure 4.13 Ground truth mapping result of RoboCup2011 dataset. © [2011] IEEE. Reprinted, with permission, from [Stefan Kohlbrecher; Oskar von Stryk; Johannes Meyer; Uwe Klingauf, A flexible and scalable SLAM system with full 3D motion estimation, 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics, 1-5 Nov.2011]

4.4.3 Indoor-outdoor experiment with Hector SLAM

This set of experiments was conducted in an indoor-outdoor mixed environment. The concept for this set of experiments was to evaluate the performance of the proposed algorithm in a mixed environment with reference trajectory partially available. The device shown in Figure 4.16 was designed and constructed for the experiments. The proposed handheld SLAM device uses a Hokuyo UTx-20L as the LiDAR sensor and a GPS modular to provide reference trajectory. The experiment was conducted around Monash University campus, with part of the trip through an indoor corridor. Readings from GPS is filtered by an Extended Kalman Filter (EKF). As the GPS sensor stops providing measurements when entering indoor areas, the reference trajectory is only available partially in the experiment. The ITM algorithm iteration cycle is set to 30 seconds. In this experiment, the handheld device was held about 1.3 meters above the ground.

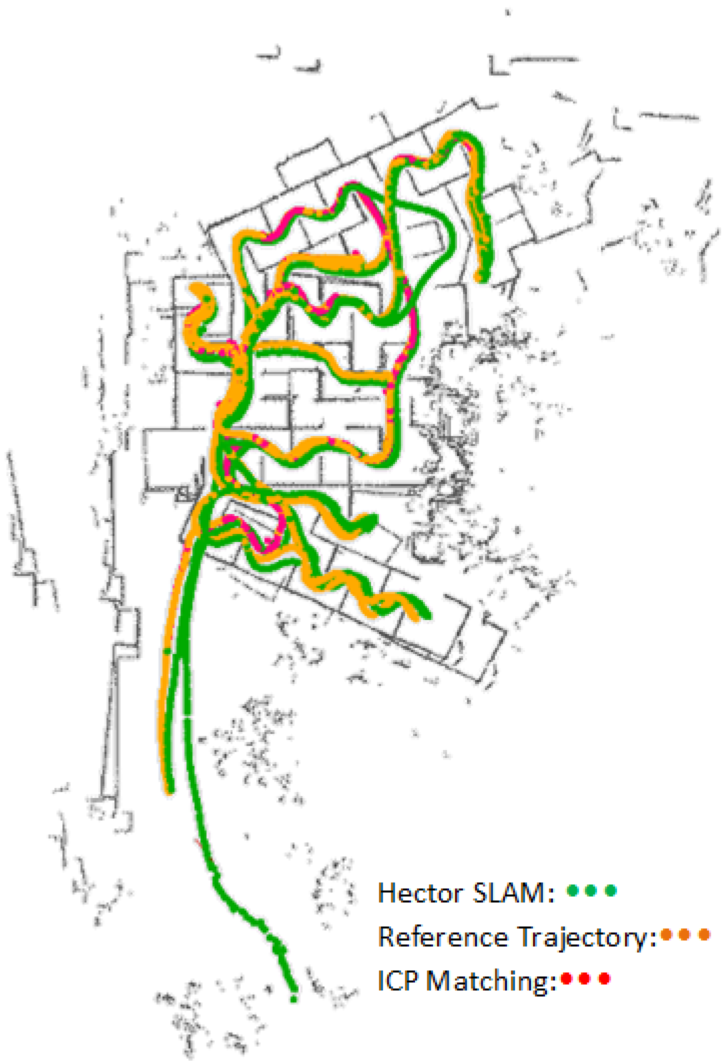


Figure 4.14 The ITM supported Hector SLAM process with RoboCup2011 dataset.

It started from the South entrance of a building (Figure 4.17(a)). After that, the device went through a long indoor corridor before exiting the building from an automated glass door. There is a featureless open space outside the North exit of the build. Then the handheld device entered an alley surrounded by metal meshes and plants. ITM algorithm is triggered twice during the experiments, as shown in Figure 4.17(b) with the solid trajectory. The dashed trajectory is the outcome from original Hector SLAM. The diamond trajectory is formed from the readings from GPS module. As illustrated on Figure 4.17(b), the effects of ITM were evident. With the handheld device started at $(x = -50, y = -20)$ on the chart and travelled clockwise, it finished the experiment by returning to the starting point. During the experiment, the handheld device

Figure 4.15 Detail of a match and correction with ITM approach from Figure 4.14.

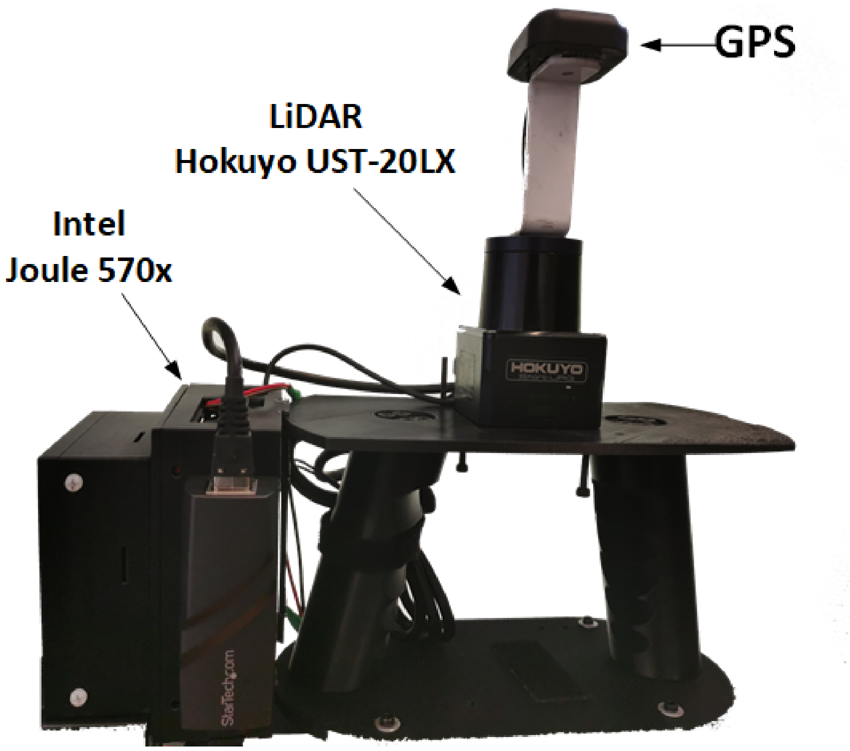
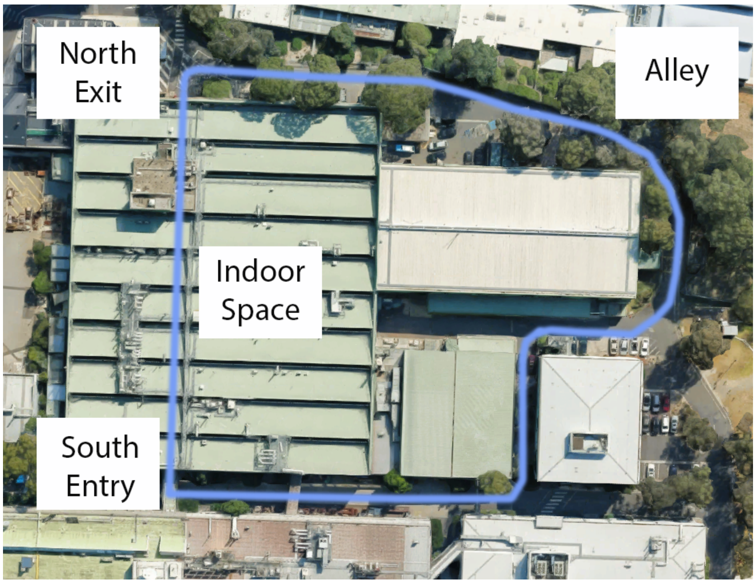
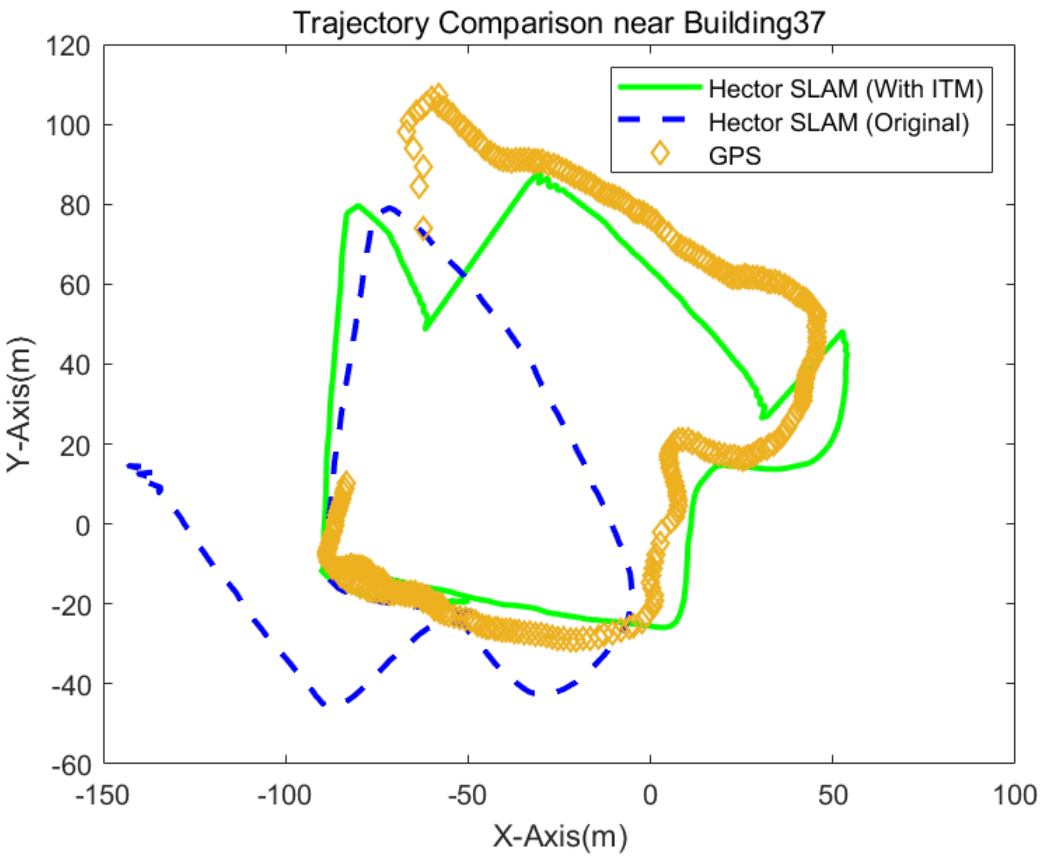


Figure 4.16 The proposed handheld device with Hokuyo UTx-20L and GPS module.

travelled around the building a bit more than one lap. Noting that the trajectory from the GPS is not available while travelling through the corridor.



(a) Satellite photo of Monash campus with trail highlighted.



(b) Trajectories compared for the Monash campus experiments.

Figure 4.17 Experiments around Monash campus with trajectories.

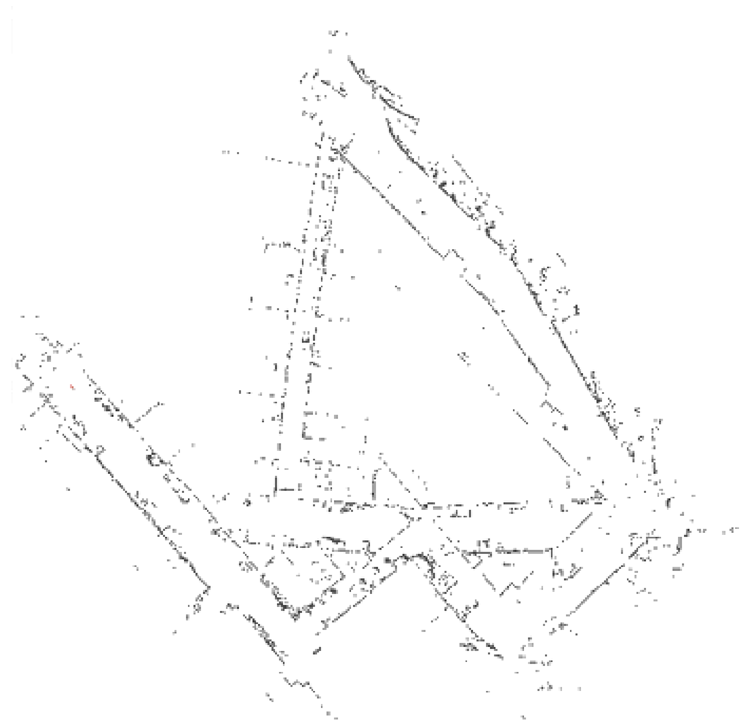
Figure 4.18 shows a comparison between mapping results of Hector SLAM with and without the proposed algorithm. In Figure 4.18(a), without the ITM algorithm, the Hector SLAM is vulnerable to the complex environment. Two significant mapping errors were recorded during travelling through the corridor and entering the featureless space via the automated glass door. The heavily twisted mapping results accumulated through the mapping process and eventually caused overlapping of map sections. These problems are improved using ITM in Figure 4.18(b) with most of the twisted map sections restored. The two successfully triggered ITM processes helped the Hector algorithm to relocate the system pose on the map after major failures.

To quantify the effects of the proposed ITM algorithm, assume position difference (PD) is the difference of positions in the Hector trajectory with and without ITM compared with the position estimation from the GPS with EKF. Using Equation (4.8), x and y are the coordinates of the position estimation in GPS frame. $\hat{x}$ and $\hat{y}$ are the coordinates of the position estimation from a corresponding algorithm.

In Figure 4.19, the difference between the original Hector trajectory and the GPS reading is shown in the solid line. The dashed line indicates the difference between Hector with ITM and GPS reading. The gap on the chart represents the time period without GPS reception. Two sharp decreases can be seen on the dash line. These decreases indicate the correction of system posture caused by the ITM algorithm. Each ITM correction reduces the position difference between the Hector position estimation and the GPS reading.

4.5 System pose correction in 3D

The work documented in previous sections evaluated the proposed ITM method on 2D mapping system. However, the proposed ITM method should also work in 3D mapping algorithms. This section discusses the work which extends the ITM method to 3D SLAM scenarios. Specifically, LiDAR Odometry and Mapping (LOAM) was selected as the targeted 3D SLAM algorithm to evaluate the performance of the proposed methodology.



(a) Mapping result with original Hector SLAM.

(b) Mapping result with the proposed ITM algorithm.

Figure 4.18 Comparison of Mapping Results with and without ITM near Building 37 on Monash campus.

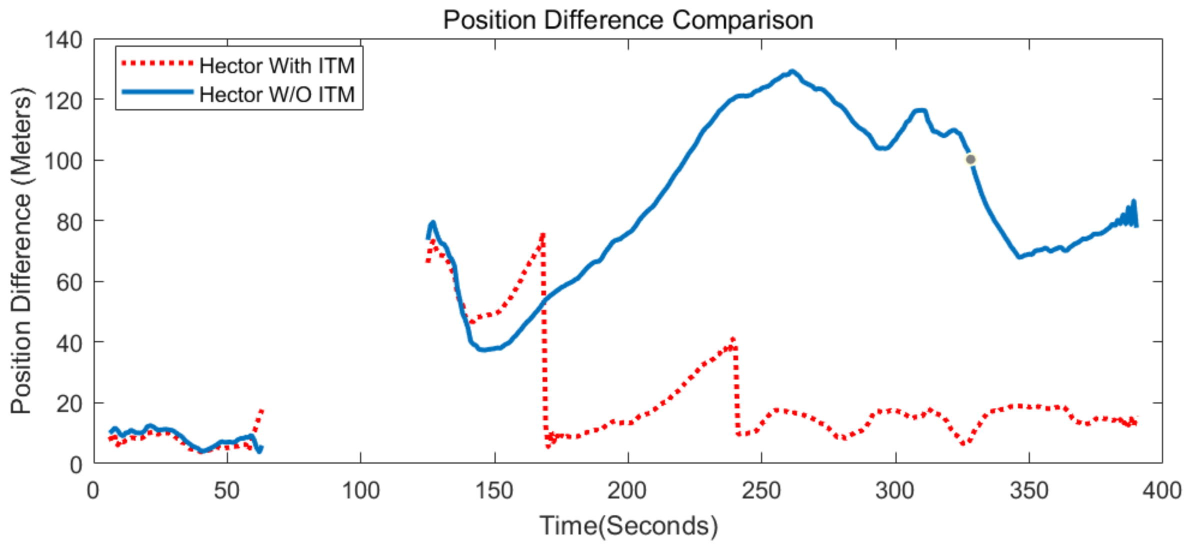


Figure 4.19 Position difference comparison of Hector SLAM with and without ITM.

4.5.1 Posture interpolation and correction in 3D point clouds

LOAM is one of the state-of-art 3D LiDAR odometry algorithms. LOAM uses the current LiDAR frame to match with the previous frame and submaps, respectively. Its two-phase matching helps the algorithm to reduce drift errors, especially when travelling in a built map. However, the drift may still exist if mapping on an uncharted path. As shown in Figure 4.20, the LiDAR Odometry is the high frequency odometry of LOAM, where the current system posture at t is purely dependent on the previous posture at $t-1$. The LiDAR to map trajectory indicates the low frequency odometry, where the current system posture is matched with the surrounding submap. The orange line shows the trajectory recorded by the GPS module. Starting at the origin of the coordinate system, the gap developed between the GPS reading and the LOAM odometry.

Unlike 2D grid maps, 3D point clouds maps are much more dense. The large number of points and high updating frequency on the map makes mapping result corrections discussed in Section 4.4 unrealistic. Therefore, in this section the focus is only on posture correction in 3D implementation. Once a correction is triggered, relocating the LiDAR frame to a new posture will result in a dislocation on the map. Spherical linear interpolation (Slerp) can significantly smooth the process as fractions of the transform is evenly applied to a series of future system posture estimations. Figure 4.21 shows a comparison of mapping results with and without Slerp.

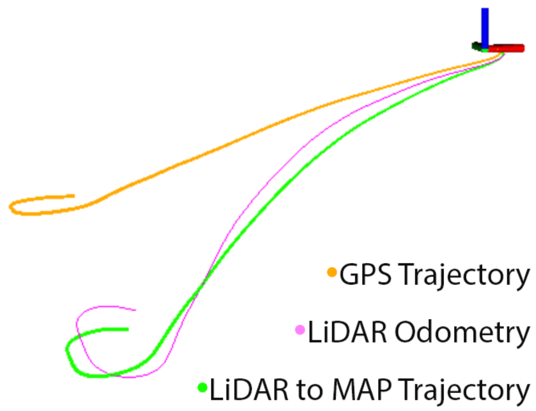


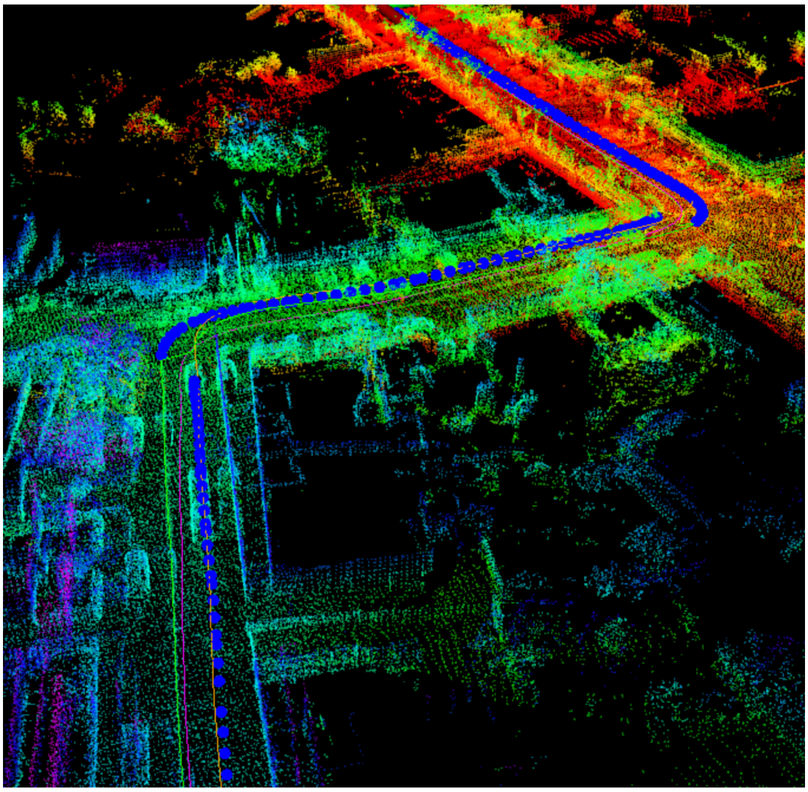
Figure 4.20 Trajectory from LOAM (Dataset01 from KITTI VO/SLAM [137]).

Mapping result in Figure 4.21(a) illustrates a sharp posture correction without Slerp smoothing. The relocation of the LiDAR frame results in a point clouds ghosting. In Figure 4.21(b), the ghosting problem is notably improved by the Slerp process. In this particular experiment, a transform is divided into 20 steps and interpolated into the LiDAR-Submap matching process in LOAM. The process runs in $4hz$, thus takes about 5 seconds to complete the Slerp.

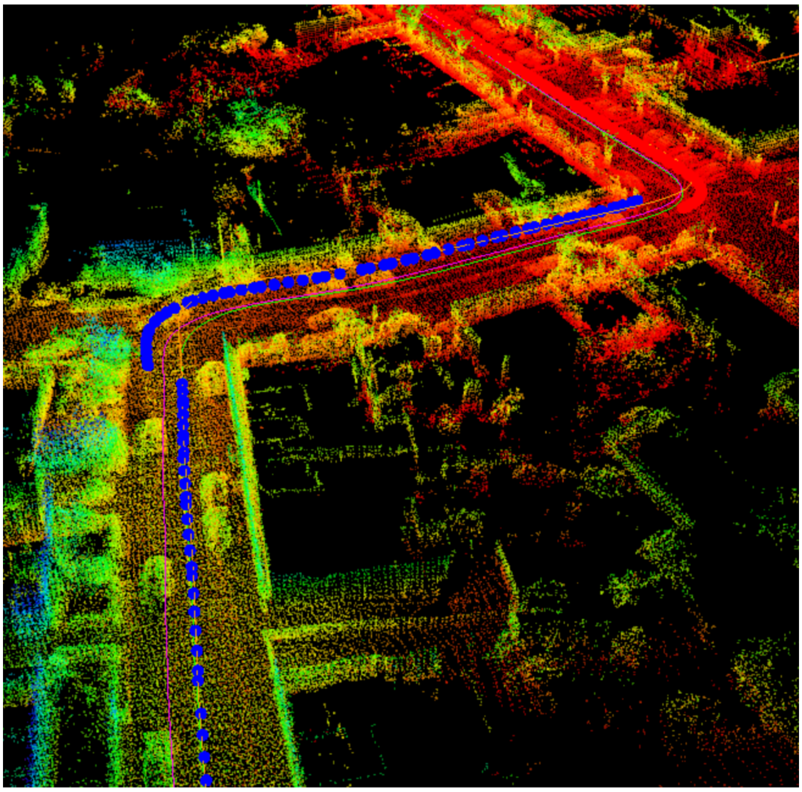
4.5.2 Matching threshold

This section extends the discussion from Section 4.3.1. Previously, the map correction process is triggered whenever the matching residual exceeds the threshold. System state correction was immediately performed with old system state directly replaced by the new system state. Immediately allocate the system to a new state will cause misaligned mapping updates which shows as mapping blurs. However, since the previous methodology uses grid map in 2D SLAM, the resolution of the map helped reducing the mapping blur to an unnoticeable level.

On the other hand, in 3D mapping, with LOAM, each LiDAR point is directly recorded on the map without translating into grid cells. This makes the system correction, especially mapping blur, more visible on the map. As explained in Section 4.5.1, using Slerp can improve the motion blur caused by ITM. However, it is also important to stop the algorithm from unnecessary corrections. In order to improve the clarity of the map, this experiment uses a low pass filter to



(a) Point could map after correction with direct system state relocation.



(b) Point could map after correction with Slerp

Figure 4.21 3D Points cloud map comparison with and with Slerp smoothing the mapping result.

Figure 4.22 Sensor Setup of KITTI Dataset. ©[2012] IEEE. Reprinted, with permission, from [Andreas Geiger, Are we ready for autonomous driving? The KITTI vision benchmark suite, 2012 IEEE Conference on Computer Vision and Pattern Recognition, June 2012]

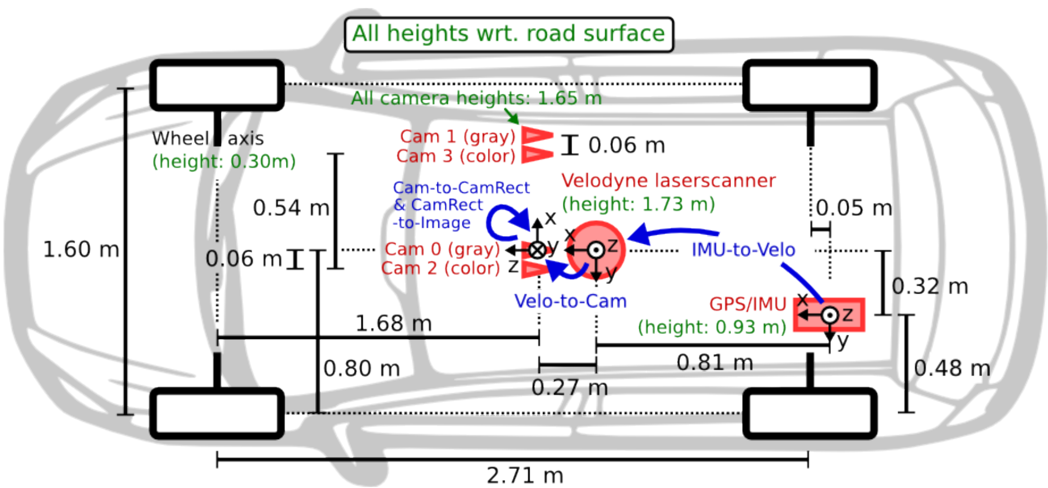
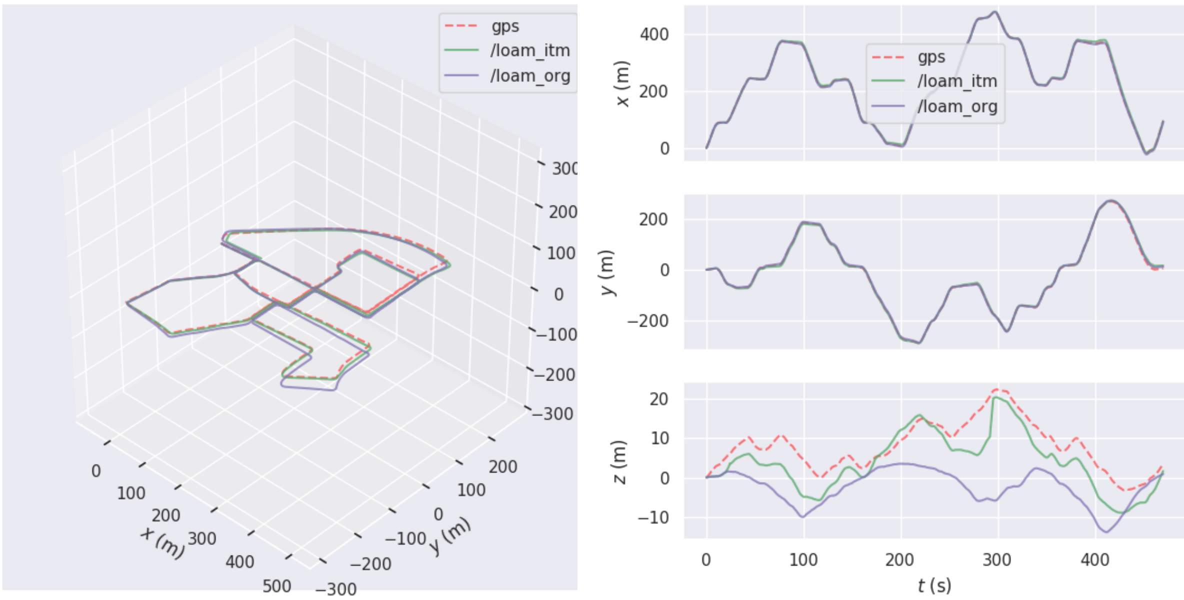


Figure 4.23 Sensor Extrinsic Information of KITTI Dataset. ©[2012] IEEE. Reprinted, with permission, from [Andreas Geiger, Are we ready for autonomous driving? The KITTI vision benchmark suite, 2012 IEEE Conference on Computer Vision and Pattern Recognition, June 2012]

remove the unnecessary corrections. In the proposed implementation, a correction is cancelled if the result of translation and rotation is too small.

4.5.3 Testing on KITTI dataset with LOAM

Under 3D category, the proposed method was evaluated with LOAM using KITTI dataset [137]. KITTI Visual Odometry and SLAM dataset is a benchmarking dataset featuring LiDAR, stereo camera, GPS and other sensors. Figure 4.22 shows the layout of the sensors utilised for the dataset. The subset of scenes selected to test the proposed algorithm features streets in the suburban area of Karlsruhe and highways. LOAM with LiDAR readings only was used as the targeting 3D mapping algorithm. GPS readings were selected as the reference trajectory. The extrinsic information of the selected sensors are reflected in Figure 4.23.

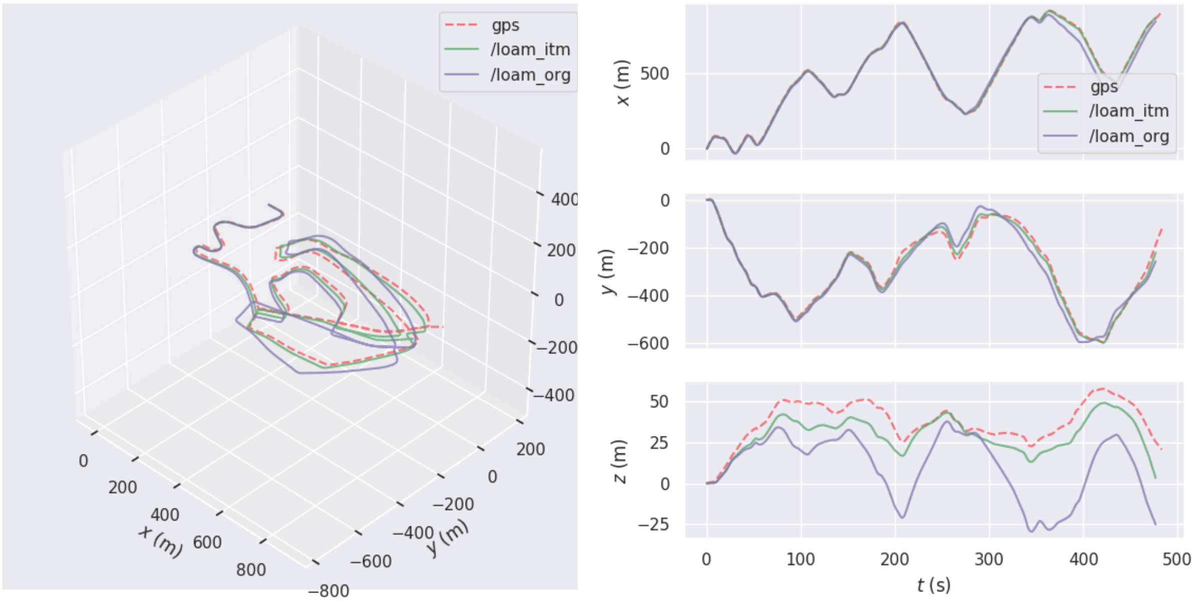


(a) Trajectories comparison.

(b) Axes comparison.

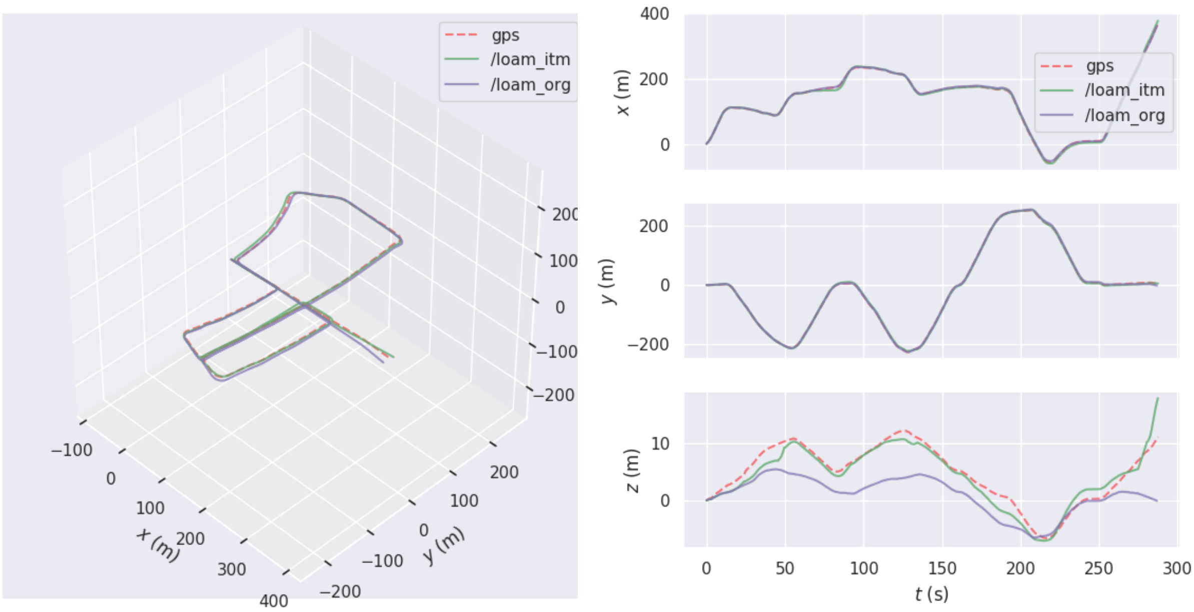
Figure 4.24 Comparison of mapping results with and without ITM using KITTI 00 dataset.

Similar to the previous experiments with Hector SLAM, after each qualified ITM process, a translation is applied to the current system posture. As explained previously, 3D map restoration is computationally expensive. Slerp was used to smooth the translation and rotation in these tests. The detailed comparisons between original LOAM, LOAM with ITM interpolation, and GPS frame are shown from Figure 4.24 to Figure 4.27. The proposed ITM algorithm successfully identified the gap between the GPS trajectory and the LOAM trajectory. The results of the posture correction can be seen in the comparison between each axes.



(a) Trajectories comparison. (b) Axes comparison.

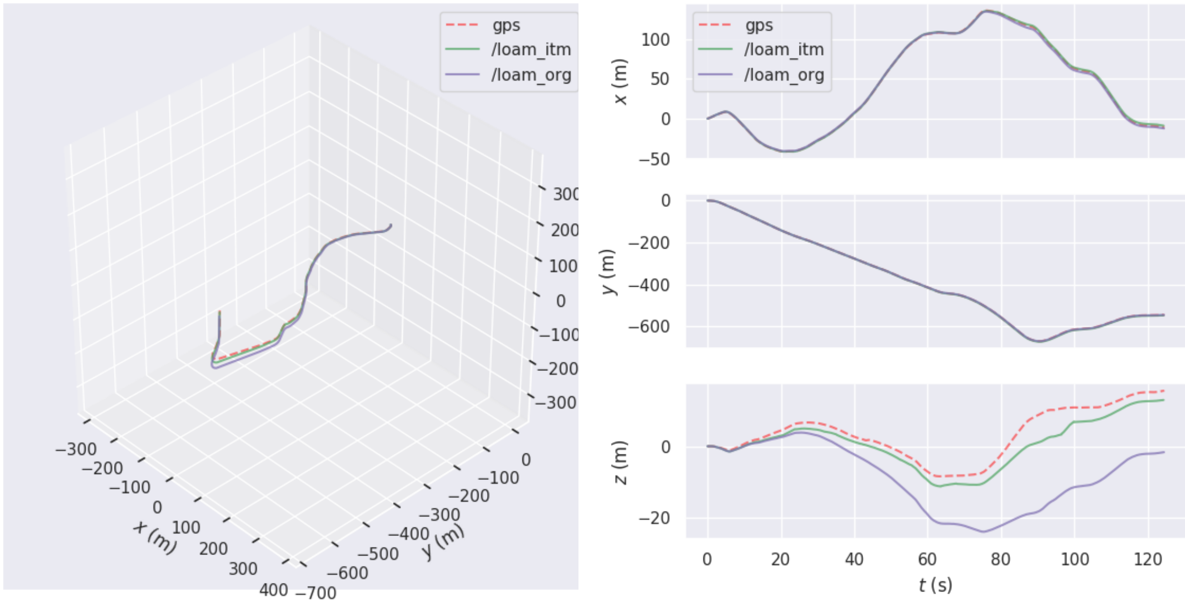
Figure 4.25 Comparison of mapping results with and without ITM using KITTI 02 dataset.



(a) Trajectories comparison. (b) Axes comparison.

Figure 4.26 Comparison of mapping results with and without ITM using KITTI 05 dataset.

In the experiments, LOAM presents the state-of-art LiDAR SLAM in terms of its accuracy. Its posture estimation for x and y axes are almost identical to the GPS reading in most tests. However, as illustrated on the charts, its z axis estimation suffers from drift errors. This problem



(a) Trajectories comparison.

(b) Axes comparison.

Figure 4.27 Comparison of mapping results with and without ITM using KITTI 10 dataset.

is more obvious in Figure 4.25 and Figure 4.27 where the car was travelling in an open loop. The drifts on the z axis are significantly reduced on the LOAM with ITM methodology, where postures were aligned with the GPS reading.

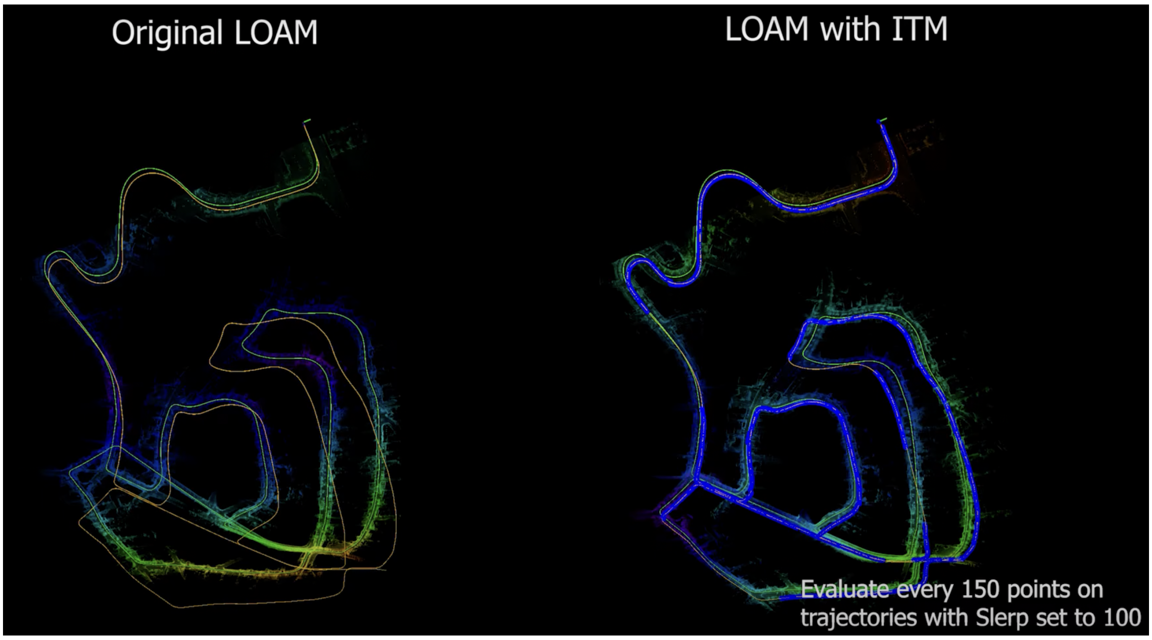
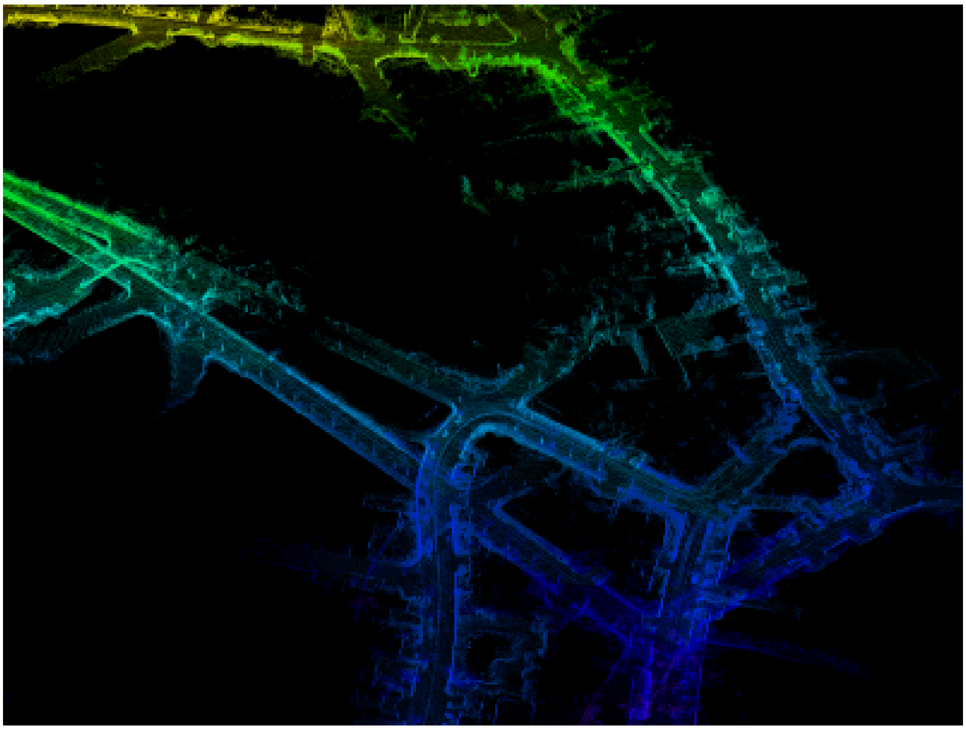
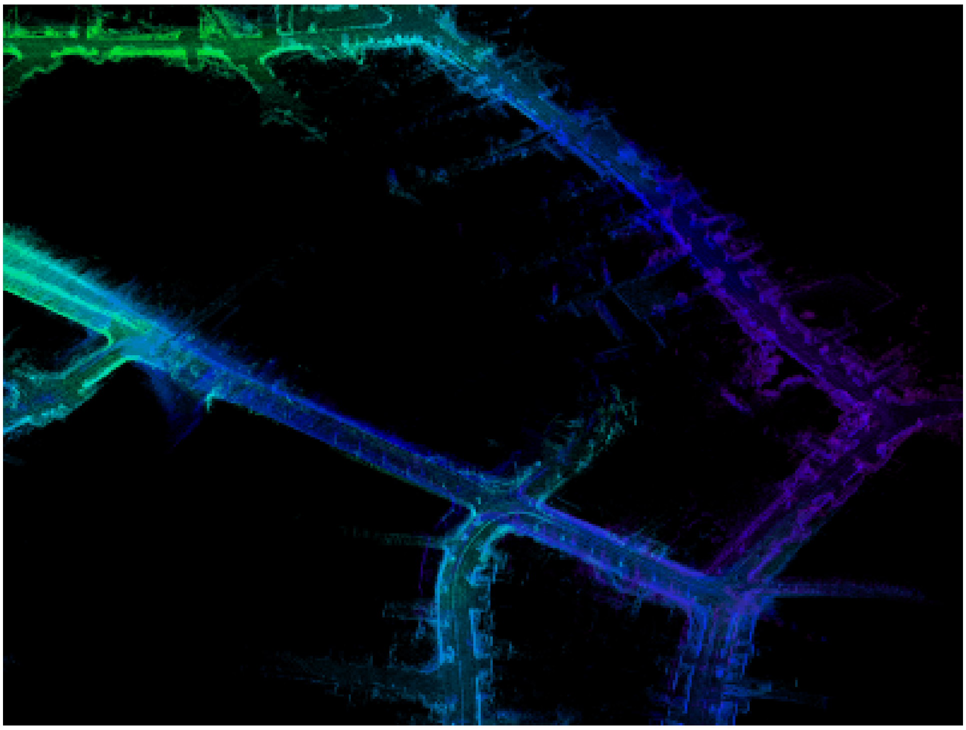


Figure 4.28 Mapping results of LOAM with and without ITM on KITTI dataset 02.

Considering KITTI 02 dataset as an example, the original LOAM results in severe mapping



(a) LOAM without ITM.



(b) LOAM With ITM.

Figure 4.29 Detailed mapping results of LOAM with and without ITM on KITTI dataset 02.

errors in multiple locations on the map, as shown in Figure 4.28. The sections of the trajectory highlighted in blue on the right map indicating the instance where ITM successfully found a match and corrected the system pose. On both maps, orange trajectories represent the ground truth of the vehicle. It is notable that LOAM with ITM is more robust against accumulated drift, compared with the original LOAM.

The LOAM does not include loop-closing. In Figure 4.29(a), when the drifts accumulated on the map, LOAM failed to correctly aligning the streets. Figure 4.29(b) illustrates a mapping results of same region with ITM. the situation was significantly improved as most of the streets were correctly aligned during the mapping process.

Drift accumulation is harder to identify when the robotic system is travelling on a single trip, for example on a highway. This is due to the fact that the robot cannot compare its scanning results with the established map. KITTI dataset 05 is simulating such a scenario. This dataset recorded sensor readings while the car travelling on the highway near Karlsruhe. The experiment result on this dataset is illustrated in Figure 4.30. In this experiment, it was found that original LOAM is less accurate with the movement on the z-axis. The error is correctly identified using ITM. On the bottom map in Figure 4.30, the proposed algorithm correctly aligned the map using GPS information and restored the system state.

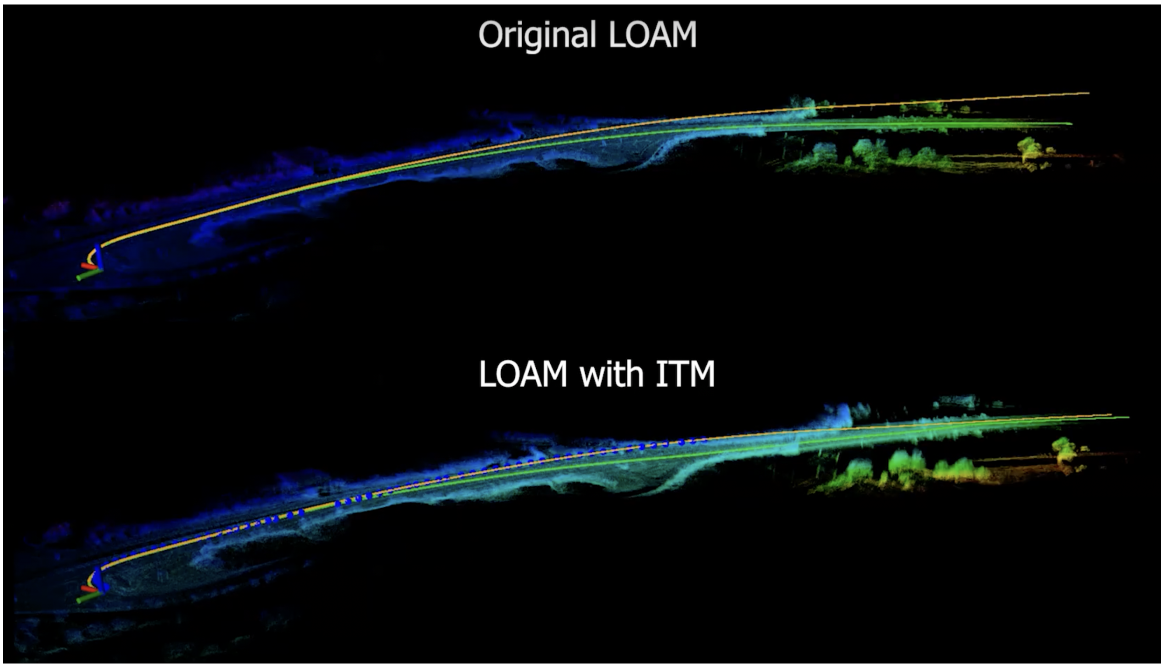


Figure 4.30 Mapping results of LOAM with and without ITM on KITTI Dataset 05.

4.6 Discussion

This chapter demonstrated an approach to improve the robustness and accuracy of a SLAM system during a mapping task. Extended from the approach proposed in the previous chapter, the ITM method iteratively corrected both the system pose and the mapping result of a SLAM algorithm. The method have been adapted to both 2D and 3D SLAM algorithms to improve their mapping quality. The grid map has been extended to include timestamp for map correction. The relationship between correction window and computation complexity has been discussed along with its limitation in 3D scenarios. A correction strategy using Slerp is introduced to the 3D mapping systems to balance between the mapping quality and effectiveness. The experimental results demonstrated the adaptability of the proposed ITM method on both 2D and 3D SLAM implementations using grid maps and point clouds.

Chapter 5

2D Enhanced 3D Multi-LiDAR SLAM for Solid State LiDAR on UGV in Urban Environment[1]

5.1 Introduction

Numerous studies have been conducted on 3D SLAM. In these studies, different approaches based on various combinations of sensors were proposed. Among all different sensing methods, algorithms using LiDAR provide some of the state-of-the-art results. Unlike cameras, a LiDAR scanner scans the space using individual laser beams. Beam-wise scanning results in limited coverage of the field-of-view (FOV). Different 3D LiDARs generate different scan patterns, which also affects the processing of the LiDAR readings.

Using parallel laser beams, multi-line spinning LiDARs are the most adopted LiDARs in both research and practical application. One of the most significant advantages of a spinning 3D LiDAR is that the laser beams repetitively scan over the same area during rotations. The spinning feature benefits the object detection algorithms as the same objects will appear on two consecutive scans. However, since the laser beams are parallel to each other, there are no

[1]The works contained within this chapter have previously been published in: [J1]

scanning coverages between beams. This means the density of the scanning results is directly related with the number of beams. In addition, the spinning mechanism requires rotational joints in the sensor, which greatly limits the life-span of the LiDAR. Abrasive wear of the rotational joints also requires the LiDAR to be manually calibrated periodically. Last but not least, one of the biggest drawbacks of multi-line spinning LiDARs is the price. A high accuracy multi-line spinning LiDAR on a unmanned ground vehicle (UGV) will likely cost as much as the rest of the robotic system. The limitations of multi-line spinning LiDAR has created a research gap in the study of 3D LiDAR mapping algorithms.

While spinning LiDARs are dominating the market, the Semi-Solid-State and Solid-State LiDAR (SSL) have quickly attracted a growing amount of users. Compared with a spinning LiDAR, SSLs are very competitive due to their high cost-efficiency. A mapping system equipped with a couple of 2D spinning LiDAR and SSLs will likely still cost lower than a high-end spinning multi-line LiDAR. Furthermore, spinning LiDAR requires manual tuning before and after shipping to the customer, which could be avoided with SSL manufactured by micro-electromechanical system (MEMS).

However, most of the SSLs have a narrow FOV. The irregular scan pattern is also challenging the scan-matching process of existing algorithms. Considering Livox Mid-40 as an example, This LiDAR has a 38.4-degree FOV with scans in a petal shape. The non-repetitive scan trajectory makes the laser beam impossible to cover the same spot twice within a reasonable amount of time. All these features make traditional SLAM algorithms running on SSLs showing poor performance. Table 5.1 shown the performance comparison between a 2D spinning LiDAR Hokuyo UST-20LX, a SSL Livox Mid-40, and a multi-line 3D spinning LiDAR Velodyne HDL-64E.

Figure 5.1 shows a frame comparison of point cloud received from Livox Mid-40 within 100ms, 120ms, 500ms and 1000ms, respectively. According to the specifications from Livox, the LiDAR only covers 20% of the FOV in 100ms. Livox Mid-40 will need 1000ms to cover 95% of the FOV. The irregular scan pattern makes the same spot takes about 1 second to be scanned twice. The relatively long integration time increases the difficulty for novel LiDAR odometry

Table 5.1 Performance comparison between Hokuyo UST-20LX, Livox Mid-40 and Velodyne HDL-64E.

	Channels	Range (Up to)	Rotation Rate	Horizontal FOV	Vertical FOV	Angular resolution	Accuracy
Hokuyo UST-20LX	1	60m	43,240 pts/s	270°	N/A	0.25°	±40mm
Livox Mid-40	1	260m	100,000 pts/s	38.4°	38.4°	0.05°	±2cm
Velodyne HDL-64E	64	120m	1,300,000 pts/s	360°	26.9°	0.08°	±2cm

algorithms to find a match between two consecutive scans. It is also worth noting the petal shape scan pattern created an uneven coverage of the scan FOV, with the center of the FOV having a higher scan density than the edges.

5.2 2D-3D mixed LiDAR SLAM in urban scenario

It is believed the weakness of an SSL mapping system can be improved by adding a low-cost 2D LiDAR alongside the SSL. Without significantly increasing the overall cost of the system, a 2D spinning LiDAR provides a large scanning field to compensate for the system FOV. Moreover, most of the 2D spinning LiDARs provide much faster scanning rates, which could potentially help the SSLs to reduce the impact of long integration time.

The approach proposed for this study addresses these problems using a combination of a narrow FOV 3D SSL and a large FOV 2D LiDAR. The proposed approach features high robustness and low computational power requirements. The main contributions in this chapter are:

- **A two-phased point cloud segmentation.** The irregular scan pattern of a SSL makes it easy to lose feature points between two consecutive scans. With the Manhattan-World assumption [127], the 2D LiDAR scan results were used to vertically segment the 3D

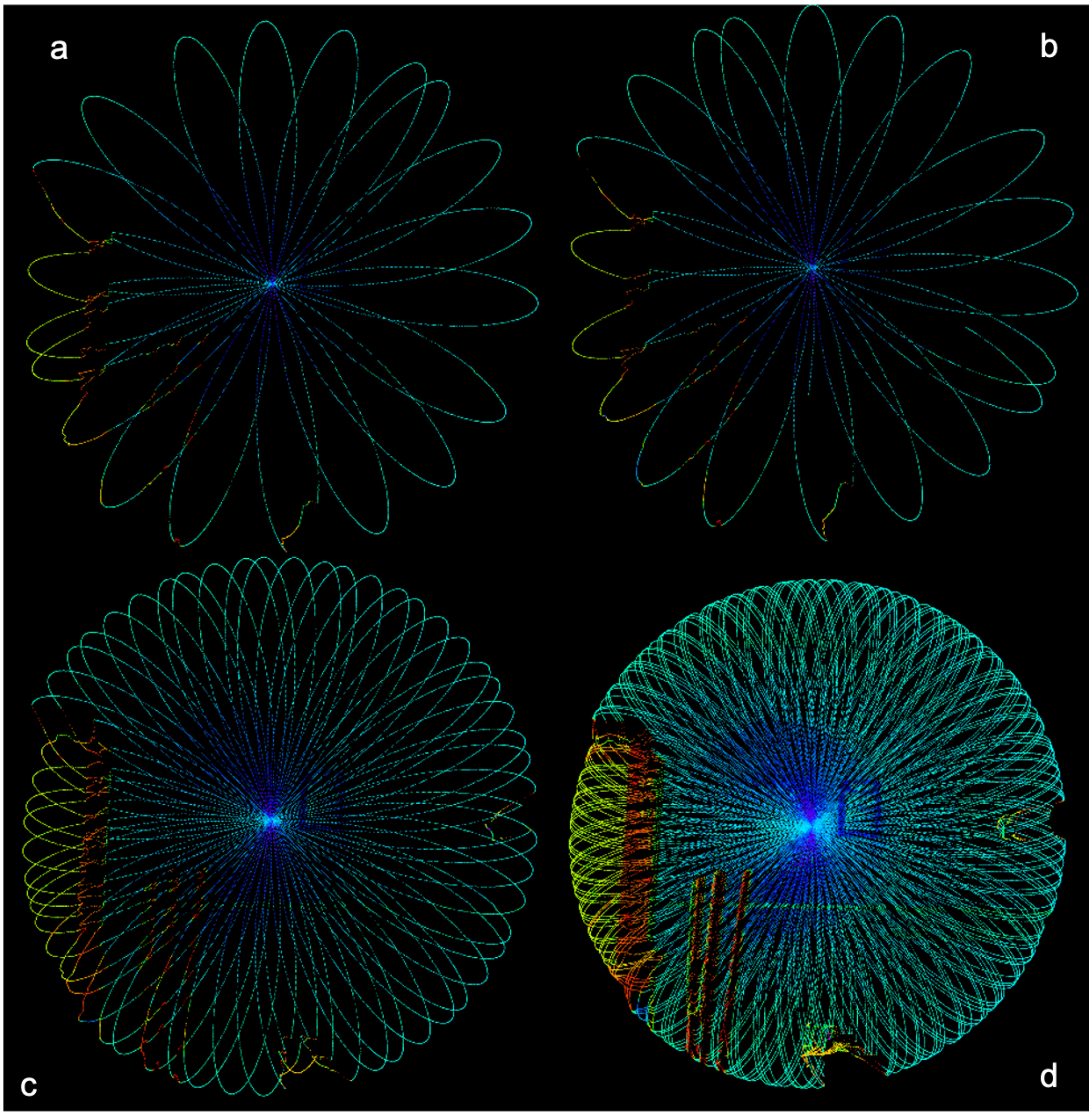


Figure 5.1 Point clouds from Livox Mid-40 reading: (a)100ms, (b)120ms, (c)500ms, (d)1000ms.

LiDAR FOV into several corners and plane sections. The segmentation is based on the fact that in urban synthetic scenes, most of the features are vertically aligned. A plane feature seen near the ground will likely be repeated vertically. Only corner and plane feature points from corresponding sections were selected for system state estimation. In this way, the system was able to identify major features in the scenario, and thus avoid small features that easy to lose track of.

- **A supplementary odometry frame.** Compared with 2D LiDARs, SSLs offer a much slower scanning frequency. The large time gap between two scans results in a large displacement between the previous system pose and the current system pose, which could introduced some difficulties to the scan-matching functions in a SLAM process. This study uses the high frequency 3 degree-of-freedom (DOF) pose estimation from 2D LiDAR to smooth out the transformation between each 6 DOF odometry update from a 3D LiDAR.

- **Enhanced Field-Of-View.** Narrow FOV of an SSL limits the mapping performance of a robotic system. A large proportion of the FOV will be occupied with featureless surfaces if the obstacle is too close. Since a 2D LiDAR has a larger FOV, the proposed mapping algorithm could provide an additional source of obstacle detection, which helps the robotic system to overcome sensor degradation.

These contributions make the proposed approach significantly more robust than approaches with a single narrow FOV SSL, and also more cost-effective than most of the high-end 3D spinning LiDARs. The proposed system features a 2D Hokuyo UST-20LX and a 3D Livox Mid-40, which aimed at improving the mapping quality of a single SSL mapping system.

5.3 System design

Most of the current mapping approaches have focused on improving the mapping performance of a specific LiDAR. However, as a system, the stability and accuracy of a SLAM process depend on a broad range of factors. This is especially the case for an UGV, as such a system often has multiple LiDARs mounted. In most of the studies, multiple LiDARs only function as a source of extra FOV. The backbone of these approaches is still a single LiDAR mapping system, with the FOV extended.

In this work, a different approach is presented to systemically improve the reading of a SSL for mapping and odometry purposes. The proposed approach utilises the advantage of a 2D large FOV LiDAR and a 3D narrow FOV SSL to provide a more adaptive and robust mapping system. The proposed approach is designed to fit on top of a UGV. The following sections present the structure of the proposed multi-LiDAR system, the feature extraction methods and the experimental results.

5.3.1 Multi-LiDAR Field-of-View integration

A multi-LiDAR scanning unit was designed and developed for this study. In the scanning unit, the main mapping component is the Livox Mid-40, which provides scanning beams in 38.4° petal scan pattern. The supplementary 2D Hokuyo UST-20LX provide 270° FOV of a horizontal fraction of the mapping space. Instead of mapping, the aim of adding a supplementary 2D LiDAR is to enhance the robustness and accuracy of the SLAM algorithm. Figure 5.2 shows an overview of the FOV layout of the LiDARs used in this work. As illustrated, the x-axis of both Hokuyo UST-20LX and Livox Mid-40 are facing forward. The overlapping FOV between the two LiDARs is a 2D 38.4° scan fraction.

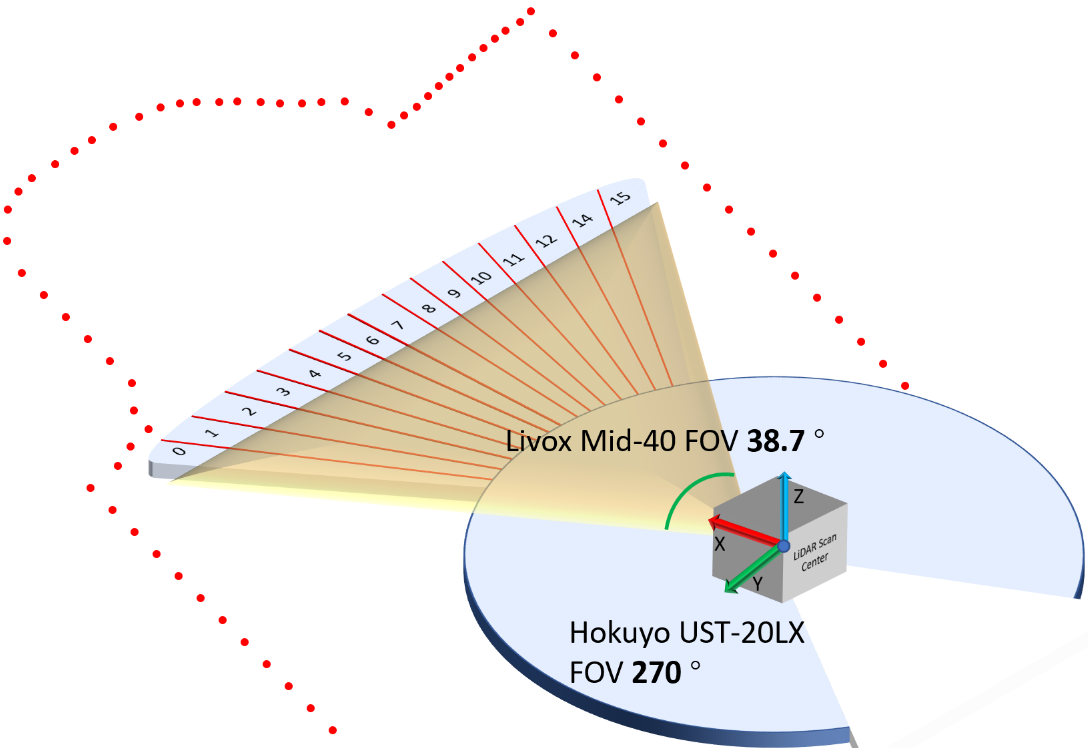


Figure 5.2 A comparison of FOV between Livox Mid-40 and Hokuyo UST-20LX.

The two LiDARs are vertically aligned with their z-axis both pointing upward. Figure 5.3 reflects the mapping unit structure and the layout of the LiDARs. From the UGV coordinates system, the x-axis and y-axis origin of both LiDARs coordinates system are the same. There is a 125*mm* difference between the z-axis origin of the two LiDARs.

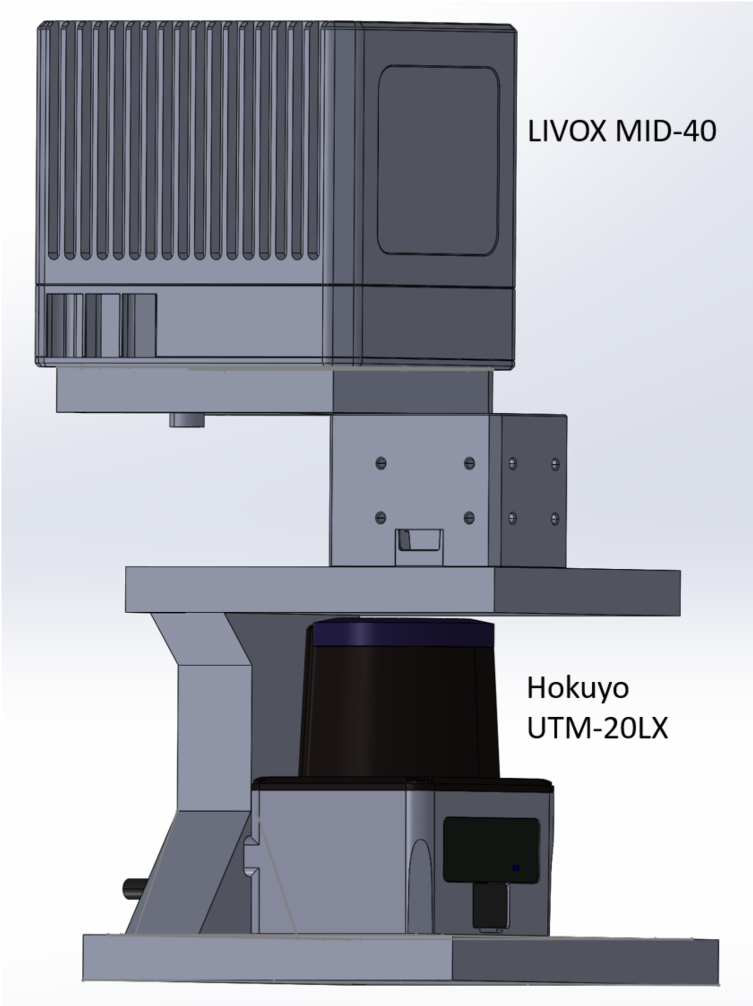


Figure 5.3 Hardware setup of the mapping unit.

5.3.2 System integration

Modern UGV systems generally contain multiple LiDARs. As explained previously, previous studies have used the supplementary LiDARs only to extend the total FOV. However, a novel mapping system architecture was proposed in this work, in which the LiDARs in the system are operating individually, but also in a combination. Specifically, the LiDARs are operating individually, and the laser readings from different LiDAR are processed separately. The information extracted from these readings is then used for different purposes. By stating that the LiDARs also work in combination, it means that the approach proposed in this chapter essentially requires both LiDARs to participate in providing mapping results.

Figure 5.4 shows the structure of the mapping unit and the data flow between different system component. Instead of meshing the LiDAR reading together, the proposed algorithm uses a 2-step point cloud processor. Overall, the 2D LiDAR readings was used to segment the FOV of the 3D LiDAR, which allows the system to adapt to the scanning scene.

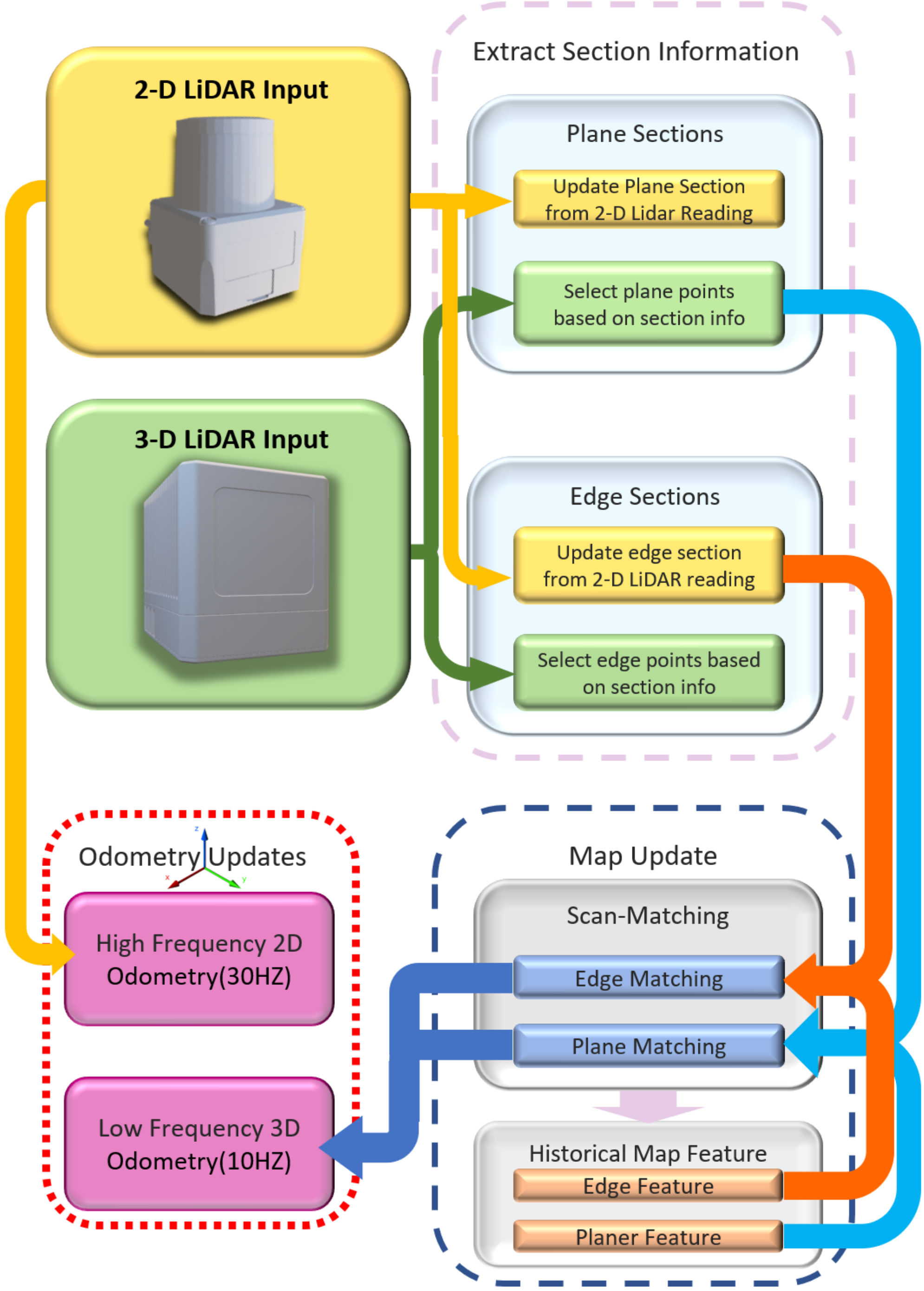


Figure 5.4 A software data flow chart of the proposed approach.

As shown in Figure 5.4, the approach has two major sensing modules: a 2D spinning LiDAR and a 3D SSL. The two LiDARs provide the system with the ability to sample the surroundings in different frequency and dimensions. The curvature information gathered from the high-frequency 2D LiDAR was used to classify the surrounding environment into a number of sections. The 3D point cloud is then constructed based on the segmentation information. It was found that targeting the feature searching functions to a specific interest area helps the algorithm to identify more feature points.

The feature extraction part of the algorithm uses the preprocessed scene information to process 3D LiDAR readings into plane points and edge points. Using these feature points to compare with the existing features recorded on the map yields the 6 DOF system pose estimation.

It is worth noting that the proposed approach also incorporates a 3 DOF pose estimation module besides the 6 DOF pose estimation. The two system components work together to ensure the accuracy and robustness of the mapping process.

5.4 2D point segmentation and feature extraction

2D LiDAR is responsible for the preprocessing of the scanning scene. The proposed system uses a Hokuyo UST-20LX which samples the 2D horizontal fraction of the space in 40hz. Overall, the purpose of preprocessing is to segment the 3D LiDAR FOV into vertical sections as illustrated in Figure 5.5.

5.4.1 The Manhattan-World assumption

The FOV of the 2D LiDAR in the proposed system has an overlap with the 3D LiDAR. However, the overlapping area is only a 2D fraction of the 3D LiDAR scanning space. This study utilises the Manhattan-World assumption [126] to estimate the relationship between the 2D and 3D scan result.

Urban scenes mostly follow the assumption of a Manhattan-World, where the structures in the environment are exhibiting strong geometrical patterns. Features such as walls and corners often follow an axis-aligned convention.

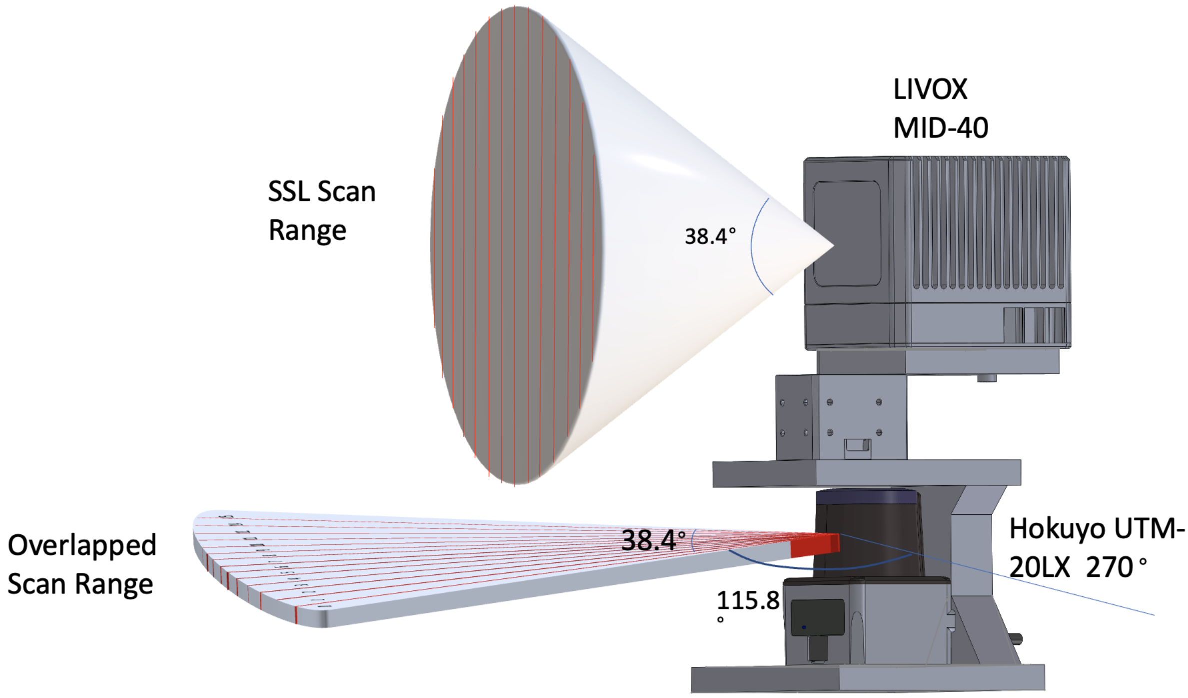


Figure 5.5 FOV overlapping in the Dual-LiDAR unit and segmentation in 3D SSL FOV.

If the 2D LiDAR is pointing to a horizontal direction, then the probability of a detected edge or surface feature perpendicular to the scan slice is substantial. This is especially the case in the constructed area where walls and corners are often in an axis-aligned convention. Figure 5.6 indicating the layout of the corridor in Engineering Building on Monash University campus. In the picture, the corner and surface features are axes-aligned. Vertically split the scanning field helps to isolate corner or planes from the scan results.

The avoid concept was adopted and the system designed in this study assumes the features captured by the 2D LiDAR would be repeated in both directions of the z-axis, thus captured by the 3D LiDAR.

5.4.2 2D feature selection

With the Manhattan-World Assumption, it is believed that the plane and corner features observed with the 2D LiDAR are likely to be repeated vertically in the 3D LiDAR reading. The consistency of features on z-axis allows the algorithm to use the 2D scan to pre-sample the 3D space.

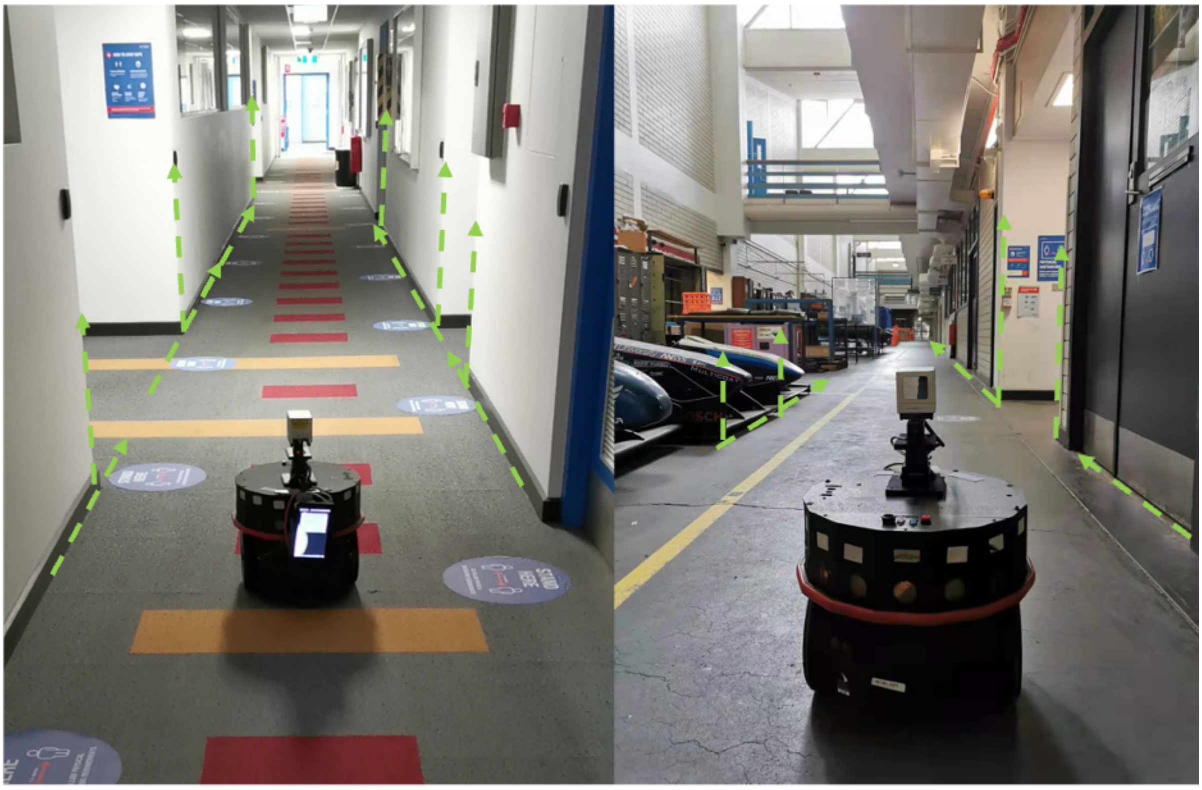


Figure 5.6 Corridors in the Engineering Department Monash University with the wall and corner features vertically aligned.

The preprocessing is achieved by calculating the curvature of the 2D reading in each sweep. Since this study is only interested in the 2D points overlapping in the 3D scan, unrelated points need to removed. Shown in Figure 5.7, the width of the FOV of Livox Mid-40 is 38.4°, whereas the FOV of the Hokuyo UST-20LX is 270°. Data from Hokuyo contains a distance reading d and a sequence number. Let ϕ_a be the incremental angle between each scan point and s_p be the point sequence number.

$$\phi_a = \frac{270^\circ}{s_p} \tag{5.1}$$

With the incremental angle of each point ϕ_a and distance reading d, the x and y coordinate of each LiDAR scan are:

$$x = d_p * \cos(\phi_a * s_p) \tag{5.2}$$

$$y = d_p * \sin(\phi_a * s_p) \tag{5.3}$$

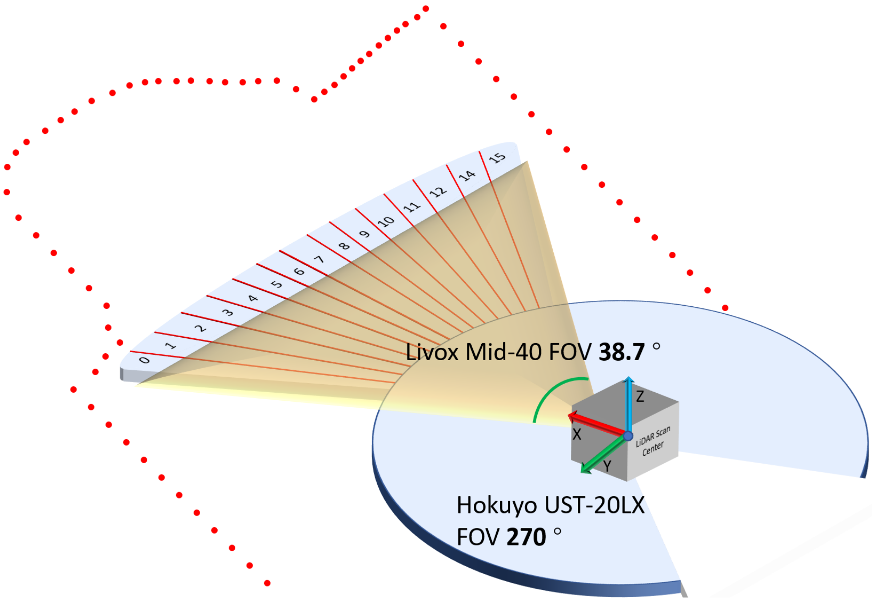


Figure 5.7 Overlapping FOV of the two LiDARs.

Let f be the scale between the FOV of two LiDARs where

$$f = \frac{38.4^{\circ}}{270^{\circ}} \tag{5.4}$$

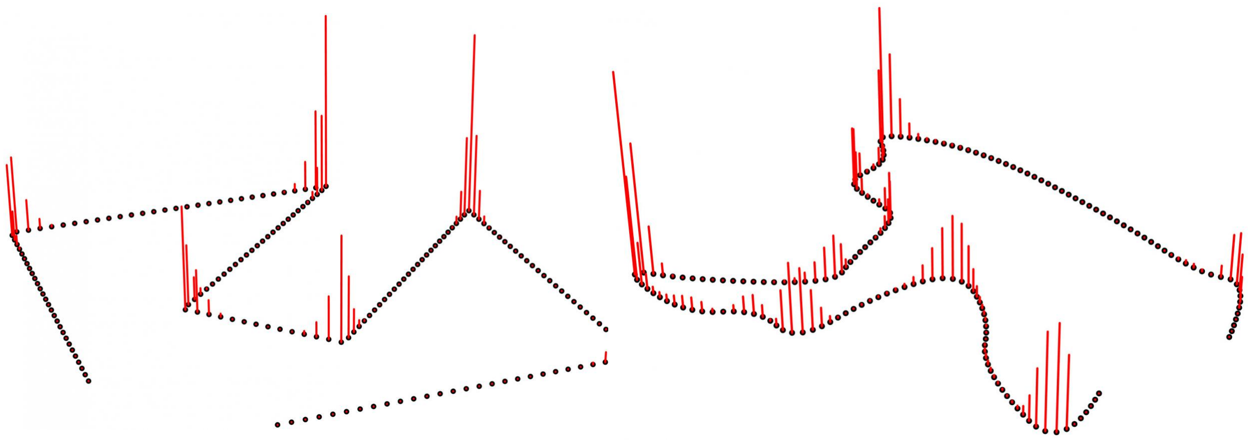
Only 2D LiDAR points overlapped with the 3D LiDAR FOV are used in the section feature extraction. Calibration is required to align the FOV of the two sensors. This work uses the method described in [105] and [107] to obtain the transformation matrix between the Hokuyo UST-20LX and Livox Mid-40. Assuming each 2D LiDAR scan S has m points. After calibration, from 3D LiDAR's coordinate frame, the point p from 2D LiDAR is selected for preprocessing if:

$$\frac{m}{2} - \frac{f * m}{2} <= \phi_a * s_p <= \frac{m}{2} + \frac{f * m}{2} \tag{5.5}$$

Assuming there are n points between two points p_a and p_c in the scan S. Let p_b be the middle point of this sweep section. The curvature of the scan section between p_a and p_c, κ_{p_b}, can be described using Equation (5.6). Sorting all LiDAR measurements between p_a and p_c according to their curvatures provides the list of points with their curvatures in descending order.

$$\kappa_{p_b} = \frac{1}{n \cdot ||p_b||} \left\| \sum_{i \in n, i \neq b} (p_b - p_i) \right\| \tag{5.6}$$

Sorting the scan reading *S* according to the curvature ranks the points in terms of their the likelihood of located on plane surface. Similarly, corner points can be extracted from the bottom of the list. Figure 5.8 illustrates an example of points ranked by their curvature values. As illustrated in Figure 5.2, the viewing window of the 2D LiDAR is evenly divided into 15 sections, including 10 corner sections and 5 plane sections. Algorithm 2 explains the approach of tagging sections to planes and corners.



(a) (b)

Figure 5.8 Edge points ranked in a 2D LiDAR scan (height of red bars indicating the curvature values).

5.5 3D LiDAR point cloud processing

3D feature selection in LOAM is based on the curvature of the scan line. While using a spinning LiDAR, the laser beams are travelling in a circle. The repeated scan path lays the foundation for the feature selection method in most of the state-of-the-art SLAM algorithms. However, directly applying this method to the Livox Mid-40 in this project will face difficulties, including:

1. The petal shape scan path gives the scan trajectory a non-even curvature. This makes differentiating corners from planes more challenging.

2. The slow scanning rate results in a larger displacement for the same feature point appear in two consecutive scans, thus harder to be paired by the matching function.

algorithm 2 Mark Sections with Feature Tag

H

Input: $Points_{sorted}$; $Section[\,]$

Output: $Section[\,]$ with feature tag

```
plane_number = 5
for (i = 0, i < plane_number) do
    S_id = φ_{Points_sorted[i]} / 2.56
    if (Section[S_id] NOT marked) then
        Section[S_id] is plane section
    else
        plane_number+ = 1
    end if
end for
corner_number = 10
for (i = 0, i < corner_number) do
    S_id = φ_{Points_sorted[Points.Size−i]} / 2.56
    if (Section[S_id] NOT marked) then
        Section[S_id] is corner section
    else
        corner_number + = 1
    end if
end for
```

To compensate for the above short comings, researchers strengthen the point selection standard when adapting existing SLAM approaches to work with SSLs. As a result, SLAM mythologies working with SSLs have to rely on less feature points and are, thus, less stable.

This problem is improved with the section flag described in the previous sections. Instead of traverse the entire point cloud for the plane and corner points, in this work, the feature points are only selected from their corresponding sections. As shown in Figure 5.9, the 3D point cloud from Livox LiDAR is divided into 15 sections according to the 15 feature zone detected by 2D Hokuyo LiDAR scan. Feature points selection in targeted sections can be less restrictive as only a specific kind of point will be extracted from the section. A larger cluster of connected feature points reduces the possibility of a feature not being acquired by future scans.

Each Livox Mid-40 measurement received by the algorithm is divided according to section information from the latest Hokuyo scan result. $\phi(p_{xy})$ is the angle between a laser beam p and x-axis projected onto the x- and y-axes planes

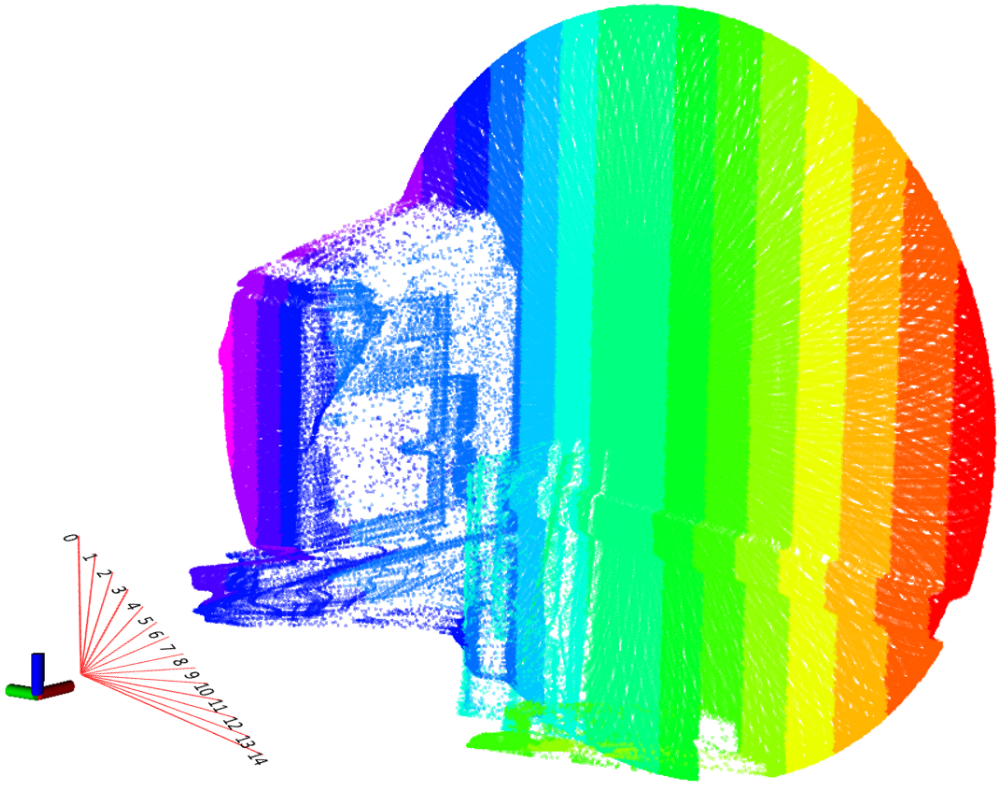


Figure 5.9 Division of the 3D LiDAR scan measurements into sections according to 2D scan information

$$\phi(p_{xy}) = \tan^{-1}\frac{\mathrm{p_y}}{\mathrm{p_x}} \tag{5.7}$$

Considering the FOV of the Livox Mid-40 is 38.4°, $\phi(p_{xy})$ is between -19.2° and +19.2°. A beam can be assigned to a corresponding section segment according to:

$$Section\_ID = \left\lfloor \phi(p_{xy}) \frac{38.4^o}{Section\_Num} \right\rceil \tag{5.8}$$

With the Livox reading being assigned to sections, the first step is to filter the points based on their qualities. Figure 5.10 illustrates an example of unwanted point removal within one section of the point cloud. The choice of unwanted points is based on the shape of the point cloud section, the scanning surface, and the FOV of the LiDAR. This work considers four kinds of point as low-quality measurements:

- Scan pattern belongs to different sections are processed independently. Since it is difficult to estimate the curvature of a point on the end of a line, points close to the edge of each section are ignored. In Figure 5.10, these points include: s,t,r,u,g,h,m,n.

- The fringe points on the edge of the LiDAR FOV are not considered for feature points due to the curved fringe beam path of the Livox Mid-40. The proposed methodology limited the FOV of the SSL to 37° to remove fringe points. In Figure 5.10, these points include: k, j, i, h.
- When a corner is covered in a scan, the point on the far side of the LiDAR scan will be not considered as a feature point. It is considered that the far side of a corner point may not be visible in future scans. In Figure 5.10, these points include: e, p.
- Since the 3D LiDAR scan is divided into sections, some scan lines only have a tiny intersection with a section. In the proposed method, a scan line with less than 6 points in a section will not be considered as candidature points in that section. In Figure 5.10, these points include: a,b,c.

After removing all the unwanted points from the LiDAR reading, the proposed methodology select the plane feature in each section according to the curvature in Equation (5.8). Two different approaches was applied to select corner features in different scenarios. Indoor corner points can be defined as a point between two planes. Let $C_{p_{ik}}$ be the curvature of point p_k with its i-nearest neighbours to its left and $C_{p_{kj}}$ be the curvature of point p_k with its j-nearest neighbours to its right. In a set of LiDAR points S, the point p_k is considered as a corner point if:

$$C_{p_{ij}} > C_{p_{ik}} + C_{p_{kj}} \text{ and } C_{p_{ik}} < 0.1 \text{ and } C_{p_{kj}} < 0.1 \tag{5.9}$$

In outdoor scenarios, the plane feature selection in 3D is similar to the 2D process, which is based on the curvature calculation in Equation (5.6). However, the number of neighbour points involved in the calculation is reduced in 3D operations. The point is considered a plane feature if the average curvature with its six nearest neighbours is less than 0.1.

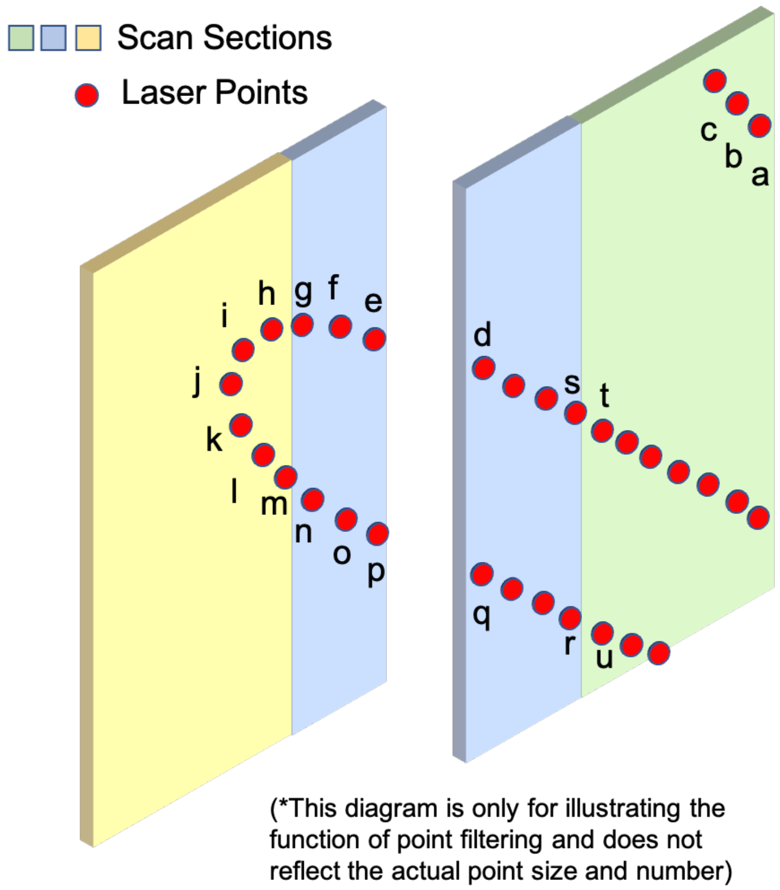


Figure 5.10 Example of point selection in a section of point cloud in LiDAR frame.

On the contrary, the corner features are calculated differently. Let L_a and L_b be the two lines formed by the five nearest neighbours on each side of the target point p_c, respectively. Assume κ_{L_a} and κ_{L_b} are the curvatures of the two lines, where

$$\kappa_{L_a} = \frac{1}{5 \cdot ||p_c||} \left\| \sum_{i=c-5}^{c} (p_c - p_i) \right\| \tag{5.10}$$

$$\kappa_{L_b} = \frac{1}{5 \cdot ||p_c||} \left\| \sum_{i=c}^{c+5} (p_c - p_i) \right\| \tag{5.11}$$

Let θ_c be the angle between the two lines L_a and L_b normalised to the unit vector. The point p_c is considered as a corner point where:

$$\kappa_a < 0.1 \textit{ and } \kappa_b < 0.1 \textit{ and } 70^\circ < \theta_c < 120^\circ$$

With all the 3D feature points selected, the next step of the proposed SLAM approach is to estimate system states based on the selected points.

5.6 Mixed frequency odometry

The low update frequency of Livox Mid-40 limited the odometry update from LOAM to around 10HZ. On the contrary, the 2D Hokuyo UST-20LX can provide 3 DOF pose estimation on x-y-axes and yaw in about 40HZ. This study uses the high frequency 3 DOF pose estimation from the 2D LiDAR to interpolate the 3D LiDAR 6 DOF odometry to improve the mapping results.

5.6.1 Pose estimation with multi-LiDAR sensing unit

Single SSL SLAM approaches suffer from their limited FOV. Fewer feature pairs in two consecutive scans cause the algorithm to be sensitive to rapid movements, especially in sharp turns. To further strengthen the stability of the mapping system, this work also introduces a pose stabilisation mechanism that uses the pose estimation from the 2D LiDAR to stabilise the estimation from 3D LiDAR.

The calculation of the 6 DOF system pose is based on the plane and corner feature distance as in [135]. Besides the 6 DOF pose estimation, the proposed system also generates a 3 DOF incremental pose estimation on x-, y- and yaw-axis from the 2D LiDAR scan using Point-to-Line Iterative Closest Point (PL-ICP). While the mapping algorithm mainly relies on the 6 DOF estimation, the 3 DOF estimation provides a supplemental pose update.

The proposed design of dual-odometry targets the instability of the mapping algorithm in extreme scenarios. As explained in Figure 5.2, the FOV of the 6 DOF odometry is restricted to 38.4°, thus in the risk of insufficient feature points. Also, the rapid change of the scan scenes will increase the difficulty of calculating the displacement between feature points. On the other hand, with Point-to-Line Iterative Closest Point (PL-ICP), calculating 3 DOF pose estimation based on 2D LiDAR readings provides a more stable and higher frequency odometry.

In the proposed work, the quality of the 6 DOF pose estimation is evaluated via two cost functions, which is in the same fashion of LOAM and Livox-LOAM. Let p_l be a point in the LiDAR frame. After applying rotation and translation using the current LiDAR pose, the coordinates of p_l in map frame is p_m . For a corner point, the Principal component analysis (PCA) is used to assure the nearest 5 neighbour points of p_m on the map belongs to a corner feature where the biggest eigenvalue is three times lager than the second biggest eigenvalue. If the PCA process indicates the neighbours surrounding p_m is forming a line, then Equation (5.12) is the residual function of the pose estimation.

$$\mathbf{r}_{corner} = \frac{|(\mathbf{P}_m - \mathbf{P}_5) \times (\mathbf{P}_m - \mathbf{P}_1)|}{|\mathbf{P}_5 - \mathbf{P}_1|} \tag{5.12}$$

Similarly, if p_m is a plane point, and the smallest eigenvalue of PCA of its 5 nearest neighbours is three times smaller than the second smallest eigenvalue, then p_m is considered as a valid plane feature. Equation (5.13) is used for the pose estimation of plane features.

$$\mathbf{r}_{plane} = \frac{(\mathbf{P}_w - \mathbf{P}_1)^T\left((\mathbf{P}_3 - \mathbf{P}_5) \times (\mathbf{P}_3 - \mathbf{P}_1)\right)}{|(\mathbf{P}_3 - \mathbf{P}_5) \times (\mathbf{P}_3 - \mathbf{P}_1)|} \tag{5.13}$$

On the other hand, the 3 DOF pose estimation from the 2D LiDAR uses PL-ICP, where the optimisation target is the minima squire error between current point and the normal vector of its two closest neighbours in the previous scan. Since the two LiDARs evaluated in this study have different publish rates, the 2D LiDAR scans used in 3 DOF pose estimation are recorded based on the frequency of 3D LiDAR measurements. The 2D LiDAR scans received between the 3D LiDAR frames are excluded from the pose estimation process.

The proposed approach keeps examining the two residuals from the 6 DOF pose estimation. If either of the preset thresholds are exceeded, the current 6 DOF pose update is replaced by the 3 DOF pose estimation transformed into the map frame. In this study, it was found that setting the threshold of plane feature residual to 0.01, and the corner feature residual threshold to 0.02 provide the most suitable outcome.

Upon updating the system state with 3 DOF (x-axis, y-axis and yaw) pose information, the z-axis, pitch and roll states are inherited from last system state. Noting that the proposed approach only modifies the 3 out of the total 6 DOF, which are the x-, y- and yaw axes of the

SLAM system. The UGV developed in this project can travel on a slope to generate motions in z-, roll and pitch axes, but the majority of motions, especially the sharp turns are more related to the x-, y- and yaw axes. Stabilising the motions in the modified 3 DOF provides the algorithm with a more accurate 6 DOF system state for the next round of 6 DOF pose estimation, thus improves the 6 DOF pose estimation.

5.6.2 Trajectory matching system integration

Iterative Trajectory Matching (ITM) method proposed in Chapter 4 is integrated into the system to further enhance the robustness of the proposed robotic system. The fundamental idea behind it is to use a supplementary trajectory to estimate the drifts occurring on the SLAM algorithm.

In this work, a 3D trajectory produced by the LOAM and a 2D trajectory generated by the Hector SLAM was used as the target trajectory and supplementary trajectory, respectively. Projecting the 3D trajectory to 2D makes the two trajectories comparable. As described in Chapter 4, the output of ITM algorithm are the translation and rotation between the two trajectories, which are used to spherical linear interpolate (Slerp) the 6 DOF system pose. This improves the overall system stability, especially on rough road conditions where the movement on the z-axis is active.

5.6.3 Motion blur compensation

Many of the existing algorithms use linear interpolation to improve the motion blur problem for LiDAR scanning. Consider Q_{t-1} and T_{t-1} are the quaternion rotation and translation of the system at time t and $t-1$ under map frame. Since the system receives LiDAR points following its timing sequence, the rotation and translation of an individual LiDAR point k in a scan at time t can be interpolated using spherical linear interpolation at time $t-1$.

$$\mathbf{Q}_k = \mathrm{Slerp}(\mathbf{Q}_{t-1}\mathbf{Q}_{t-1}^{k}), \quad \mathbf{T}_k = \mathbf{Q}_{t-1}\mathbf{Q}_{t-1}^{k} + \mathbf{T}_{t-1} \tag{5.14}$$

An alternative to linear interpolation is to use 2D pose to interpolate the 3D LiDAR readings. With the 2D pose estimation output with a higher frequency, the algorithm can generate three

2D pose estimations between two consecutive 3D pose estimation. This provides the system with the ability to interpolate the 3D point cloud with 2D movements. To achieve this, each 3D LiDAR scan is divided into three parts. Each part of the points is then interpolated with the corresponding 3 DOF pose estimation based on the timestamp. This alternative method particularly suits ground vehicle scenarios where the movements on z-axis is minimized.

5.7 System workflow

With all the designed feature described, Figure 5.11 illustrates the data flow of point cloud processing with the readings from the Livox Mid-40 used in this work. Only corresponding feature points in the section are selected based on the tag type of the section. When only one feature is selected in each section, the proposed algorithm loosens the restriction of the number of points. Instead of 4 points on each scan line, maximally 1000 points are selected in each section. A VoxelGrid filter is applied to enhance the evenness of the sample feature points. The length of each edge of the voxel cube is set to 0.3 metres. The choice of voxel size is made based on the environmental feature, the point cloud density, and the system performance.

5.8 Hardware design and experiments

The UGV illustrated in Figure 5.12(b) was built for the propose of validating the algorithm proposed in this work. The mapping unit is manufactured based on the design shown in Figure 5.12(a). Hokuyo and Livox LiDAR are vertically mounted on top of the robot with their axes manually aligned and calibrated using reflective tape. The robot uses a two-wheeled differential drive actuator and an onboard computer with quad-cores running at 1.7 GHz and 4 GB of RAM. Experiments were performed to validate the accuracy, robustness and efficiency of the methodologies.

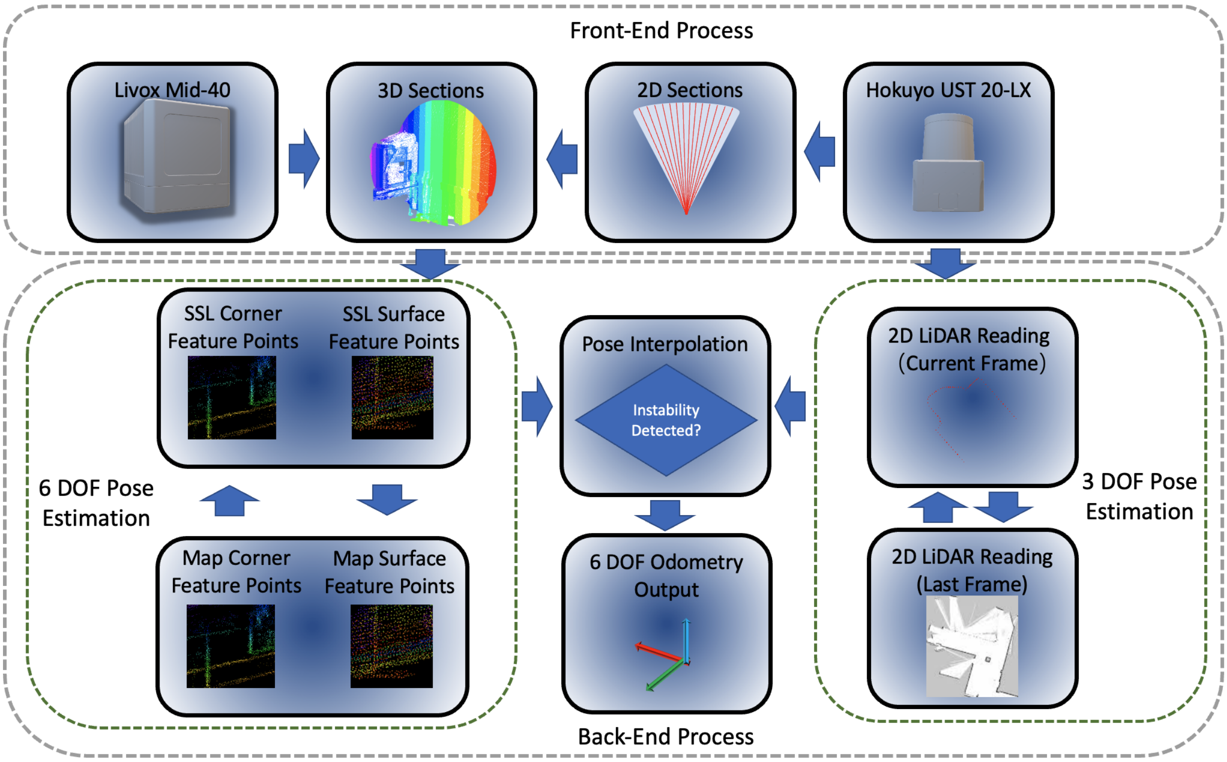


Figure 5.11 Data flow of the proposed dual-LiDAR odometry system.

(a) Multi-LiDAR mapping unit.

(b) The testing platform

Figure 5.12 Developed mapping unit and testing platform.

The experiments were performed in the first level of the Engineering building in Monash University shown in Figure 5.13). The two sharp turns in the corridor are the main challenges of

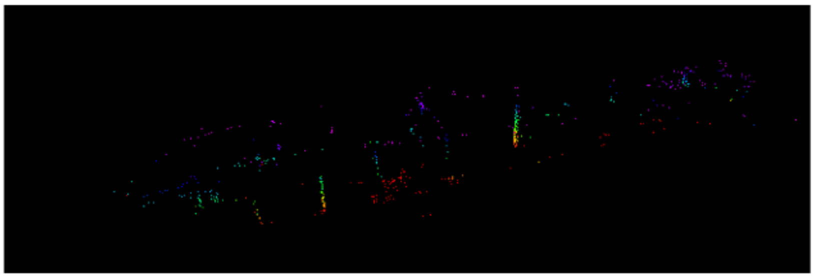
Figure 5.13 Experiment environment in the Engineering building Monash University.

Figure 5.14 Designed challenges in the experiment: sharp turns with closing obstacles.

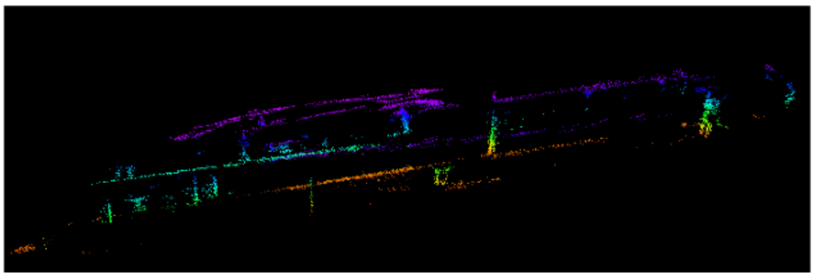
the proposed SLAM approach (Figure 5.14)). The number of 2D section segments is limited to 15, with 10 corner sections and 5 plane sections. The 3 DOF pose estimations only interpolate the system state when the residual of the 6 DOF estimation is exceeding 0.35. During the test, the system is able to deliver 6 DOF odometry in 10 Hz and 3DOF odometry in 30 Hz. During the experiments, the range of both LiDARs were limited to 30 metres. The ground vehicle was travelling at around $0.95m/s$ with an angular velocity of $1.17rad/s$ while turning. A laser interferometer-based tracker was used in these experiments to record the ground truth of the system state. The laser tracker tracks the retroreflector mounted on the vehicle to record its 6 DOF motions.

5.8.1 Evaluation of the proposed feature selection method

The presented feature selection algorithm was evaluated with the ground platform travelling through a long corridor inside the Monash University Engineering Building. With the proposed feature selection method, the SLAM algorithm is able to identify feature points in the environment more efficiently. Shown in Figure 5.15(a), Livox-LOAM is less sensitive to corner features in the testing environment. The algorithm extracts a very limited amount of corner points from the building structure. Compare with Livox-LOAM, in Figure 5.15(b), the proposed algorithm successfully covered a larger number of corner points. It is worth noting that the proposed algorithm correctly identifies the corners between the floor, ceiling and the wall, which significantly improves the coverage of corner features.



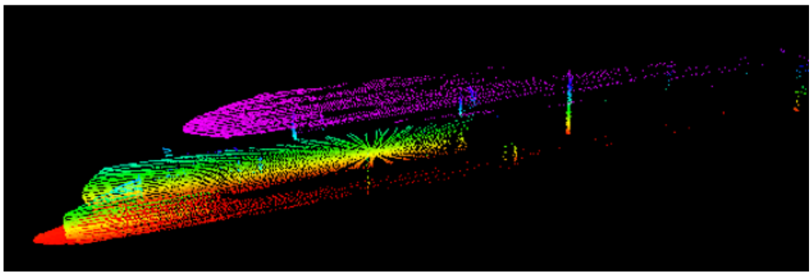
(a) Livox-LOAM



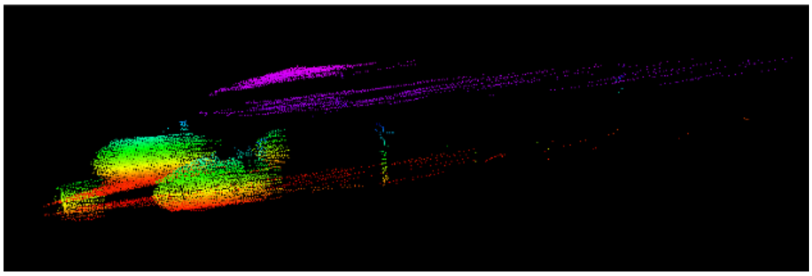
(b) The proposed algorithm

Figure 5.15 Compare corner feature collection between the proposed algorithm and Livox-LOAM in 10 scan frames.

On the other hand, Livox-LOAM classifies a vast amount of points as plane features. From Figure 5.16(a), it could be seen that plane feature selection process include some non-plane points in the results. However, plane feature selection is more restricted with the proposed algorithm, where only five plane sections are considered in each scan. As a result, the proposed algorithm only picks the five smoothest surfaces in the current scan frame as the plane feature.



(a) Livox-LOAM



(b) The proposed algorithm

Figure 5.16 Comparison of plane feature collection between the proposed algorithm and Livox-LOAM in 10 scan frames.

To further investigate the feature selection difference between the proposed algorithm and the Livox-LOAM, the numbers of feature points collected by both algorithms were compared in six different attempts. Each attempt is base on a single scan reading from a Livox Mid-40 running at $10Hz$. To average the result, this study took the tests in different environments, with indoor and outdoor scenarios. The results of the tests are shown in Table 5.2. Using Livox-LOAM, the mapping algorithm only identified a limited amount of corner points in the environment. The unbalanced feature numbers lead the algorithm to rely more on plane points than corner points.

With the proposed point preprocessing method, the algorithm built a pre-knowledge about the scanning surface, which helps the system identify more corner features from the environment. Additionally, since feature selections are limited by sections, only the high-quality surfaces are considered as the plane feature in the proposed approach. Overall, the proposed feature selection algorithm can create a more accurate and balanced feature selection results for the following scan-matching process.

Table 5.2 Number of different feature points selected by Livox-LOAM and the proposed algorithm in 1 frame of Livox Mid-40 scan reading with the sensor running at 10 Hz.

	Livox-LOAM		Proposed Algorithm	
	Corner	Plane	Corner	Plane
test_1_Indoor	15	3544	209	1544
test_2_Outdoor	11	1945	150	1325
test_3_Indoor	9	3064	174	2004
test_4_Outdoor	32	2931	95	1754
test_5_Indoor	17	2815	357	1388
test_6_Indoor	4	2032	235	2084

5.8.2 Evaluation by the odometry comparison

With three corridors connected by two sharp turns shown in Figure 5.13, Figure 5.17 describes the trajectory comparison between the 3 DOF pose estimation from the 2D LiDAR, the 6 DOF pose estimation from 3D SSL with Livox-LOAM, the presented 2D-3D mixed SLAM approach and the ground truth collected by the laser tracker.

In Figure 5.17, the outcome of 2D LiDAR incremental pose estimation illustrates its outstanding performance in corners. However, with PL-ICP algorithm, the 3 DOF odometry is vulnerable to feature less long corridors. On the other hand, 6 DOF pose estimation using Livox-LOAM successfully positioned the system in the long corridor scenario during the first one-third of the test. Nevertheless, as shown in Figure 5.17, with limited FOV, the system has poor performance in sharp turns, especially when obstacles are close to the LiDAR. The error accumulated on the map which affected future mapping results and caused large drifts in the trajectory. The trajectory of the proposed algorithm is the closest to the ground truth in the experiments.

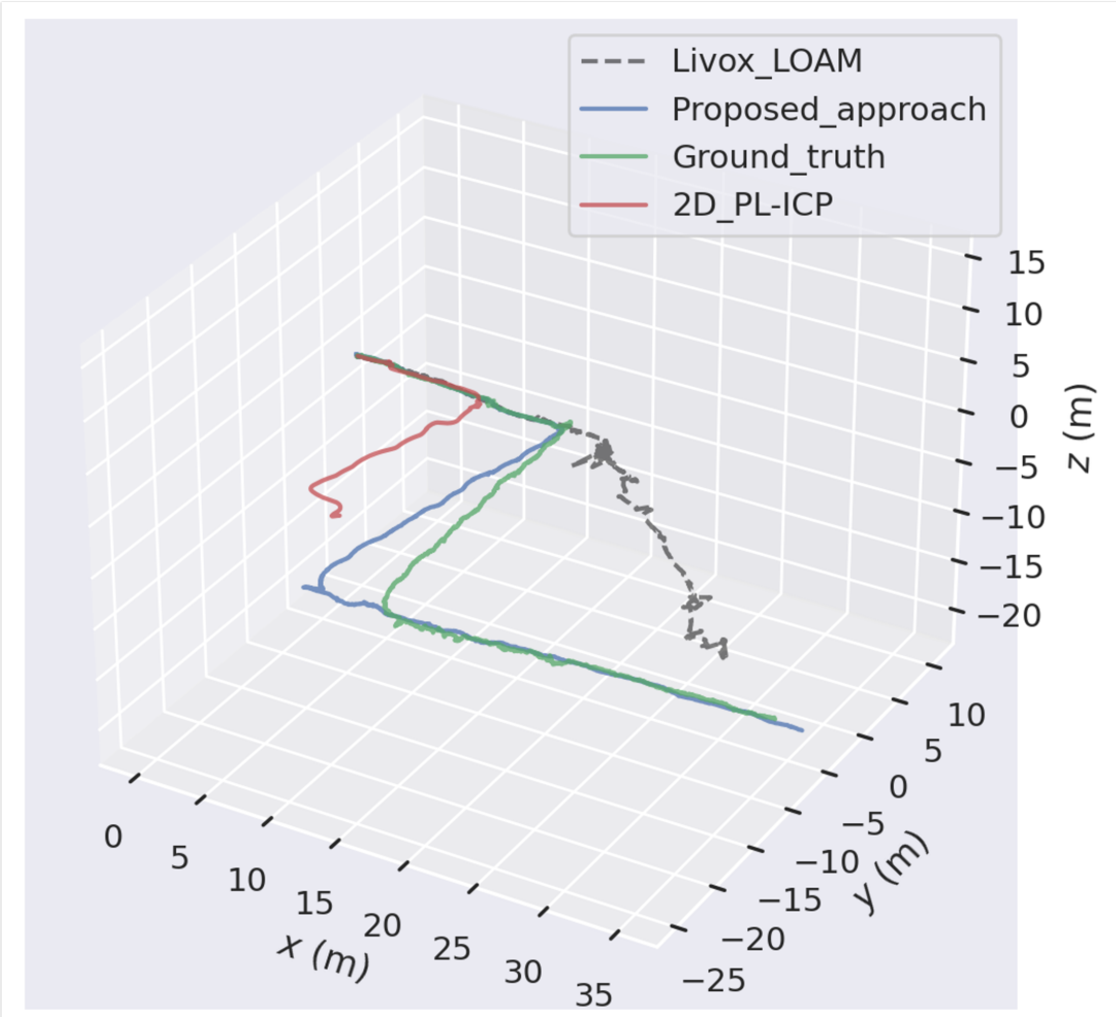


Figure 5.17 Trajectory Comparison between proposed approach, Livox-LOAM and the ground truth.

The LiDAR odometry performance significantly improved with the proposed system, where the displacement between system trajectory and the ground truth is minimised. An axis-wise comparison between the proposed system, the Livox-LOAM and the ground truth is illustrated in Figure 5.18. The proposed odometry interpolation method successfully enhanced the robustness in x- and y- axes. Similar results can be seen in Figure 5.19, where the proposed system significantly outperforms the compared approach in motions on yaw axis. It is worth to note that compared with the ground truth the proposed system has less accuracy on z- roll and pitch axes as they are not enhanced by the 3 DOF odometry interpolation method described in Section 5.6.1. However, the presented approach still outperforms the Livox-LOAM algorithm with the dual-LiDAR feature extraction method described in Section 5.4.

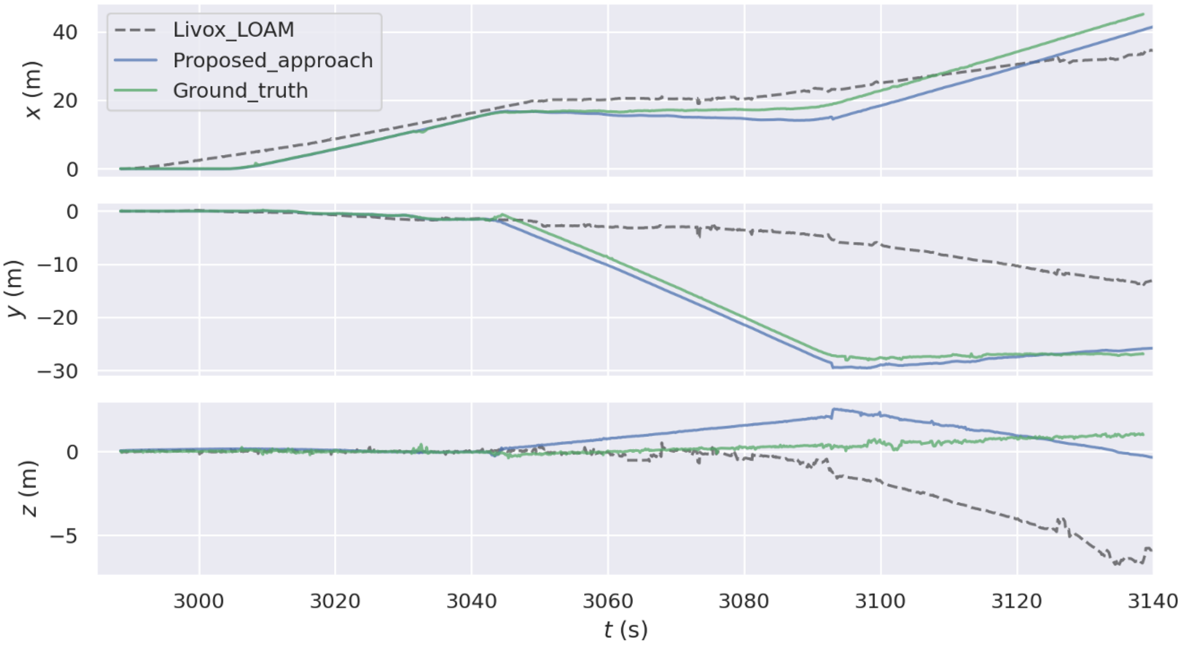


Figure 5.18 X-, y- and z-axes comparison between proposed approach, Livox-LOAM and the ground truth.

Compared with the ground truth, the proposed approach has absolute position error (APE) of 5.64. Since the accumulated mapping error of the Livox-LOAM approach is significantly larger, its APE in the same test was 41.32. Additionally, on the average, the proposed algorithm is able to utilize 12 times more corner feature points than the Livox-LOAM.

Figure 5.20 illustrates the mapping results from the Livox-LOAM and the proposed method compared with the ground truth. From the experiments, it is observed that the mapping results from Livox-LOAM are vulnerable to the shape turn. The mapping algorithm fails to locate the mapping unit after the first sharp turn in the experiment. As a result, fatal errors have been recorded on the map with major part of the area left blank. On the other hand, with the help of the 2D LiDAR, the system is able to handle the corners correctly and minimise the drifts. From the map, it can still be observed that the proposed system recorded some error on the z-axis with the upper part of the map not aligned. However, compare with the mapping result from Livox-LOAM, the improvements of the proposed algorithm can be considered as significant.

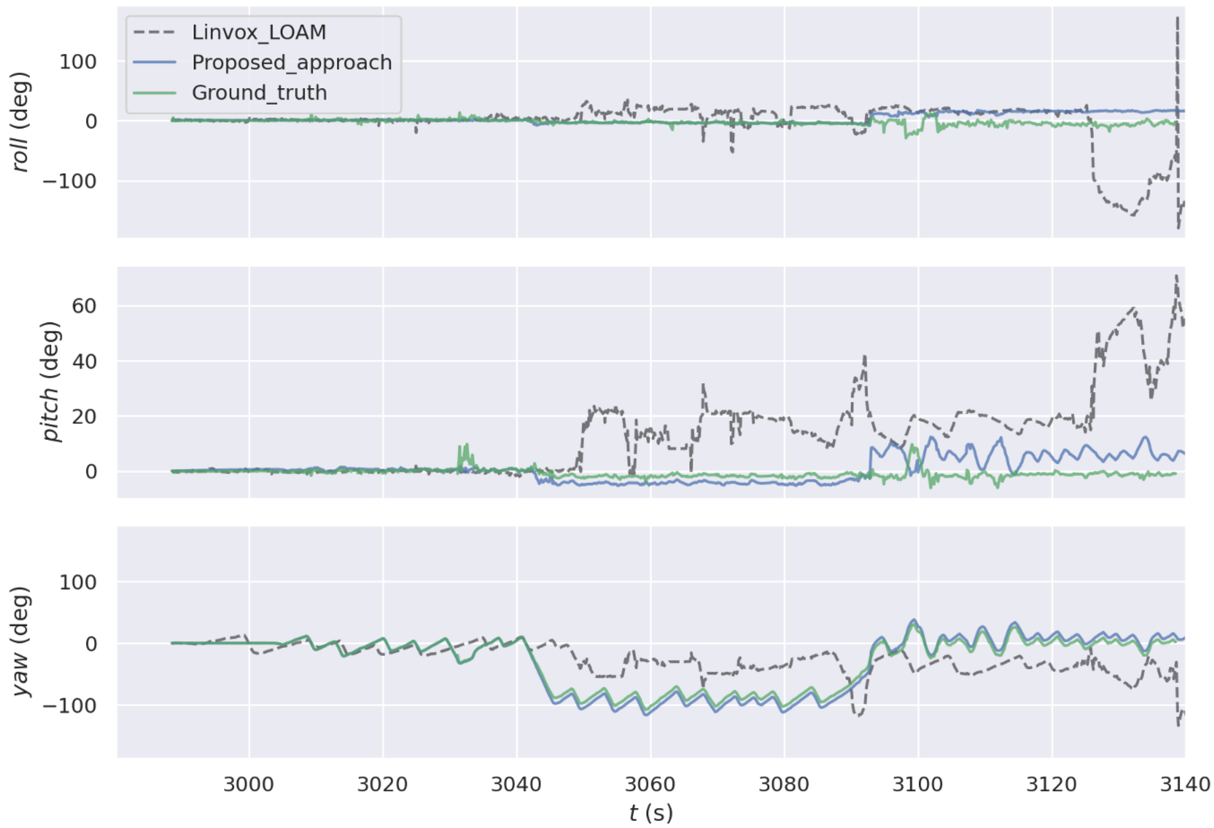
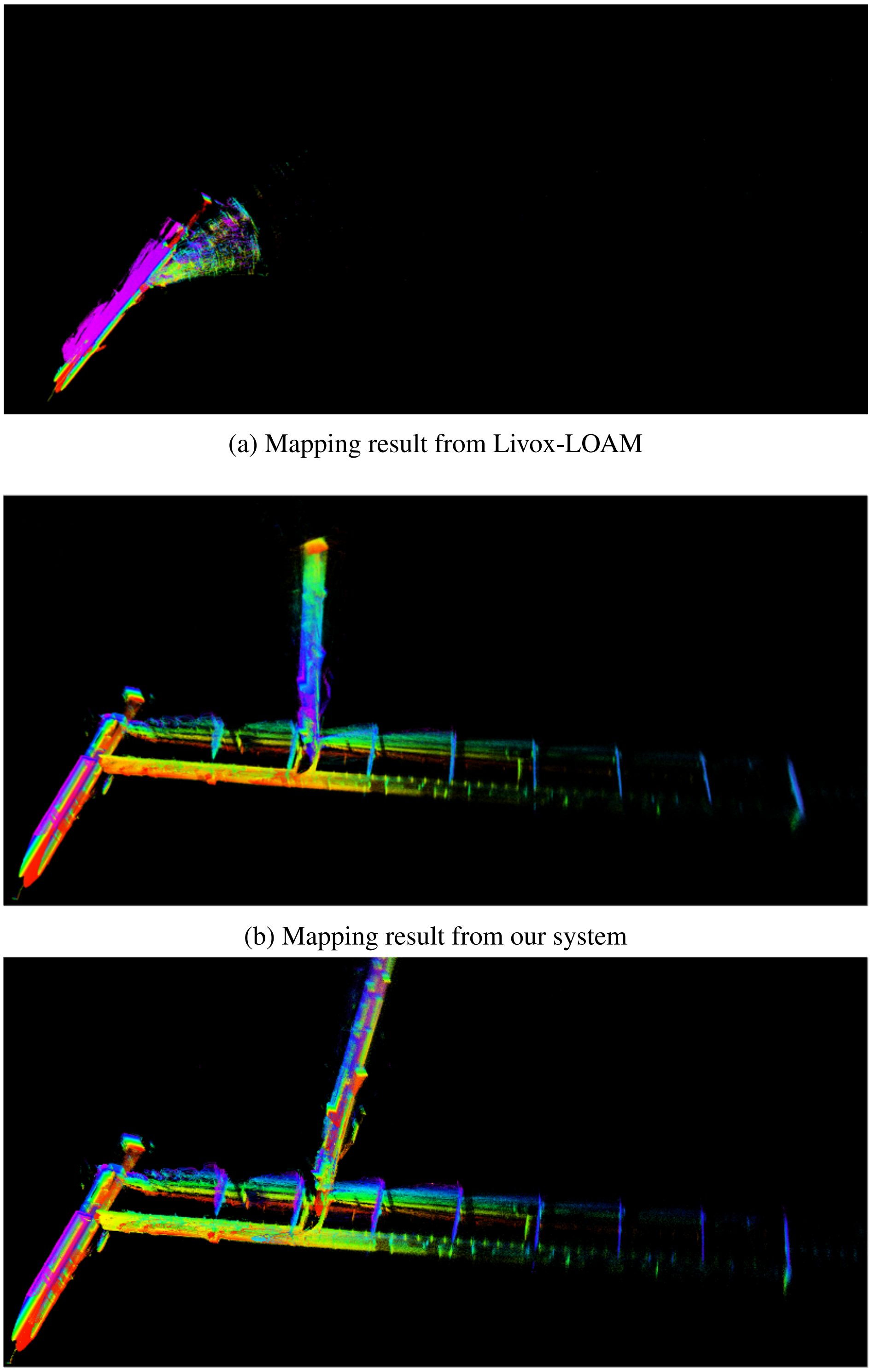


Figure 5.19 Roll, pitch and yaw comparison between proposed approach, Livox-LOAM and the ground truth.

5.8.3 Evaluation by mapping result

More sets of experiments were conducted around the Monash University campus to further investigate our proposed system's performance compared with the Livox-LOAM. These experiments were designed in scenarios that could potentially receive different results from the two algorithms.

Figure 5.21 demonstrates the experiment results of the testing platform travelling through an automated glass door. While the robot was approaching the door, both side of the door opens towards the mapping system. Using Livox-LOAM, even the algorithm is able to receive the majority of the readings through the glass, the moving door still caused significant mismatches, which results in the mapping error on Figure 5.21(a). In the same test, the mapping result from our system (Figure 5.21 (b)) shows a noticeable improvement as no significant errors are recorded on the map.



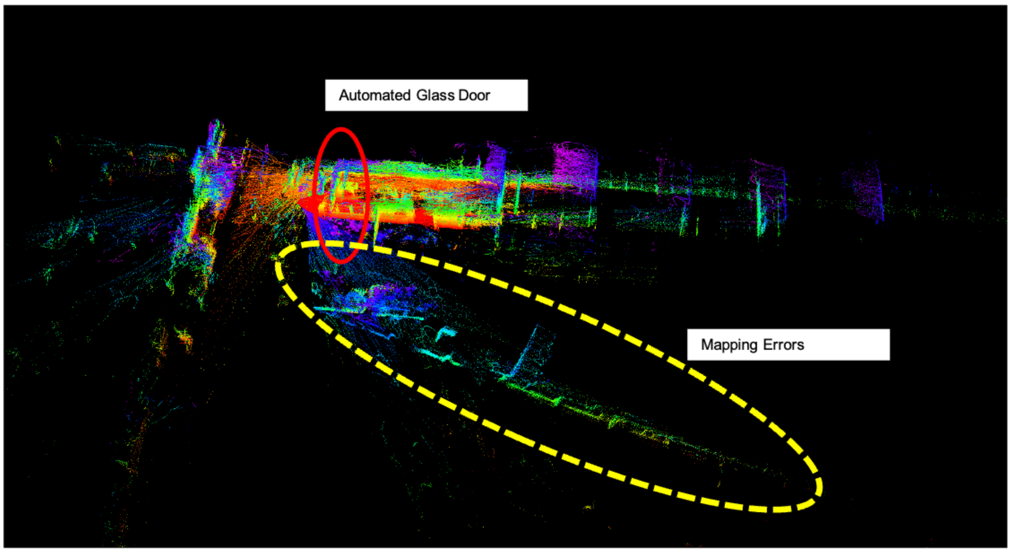
(a) Mapping result from Livox-LOAM

(b) Mapping result from our system

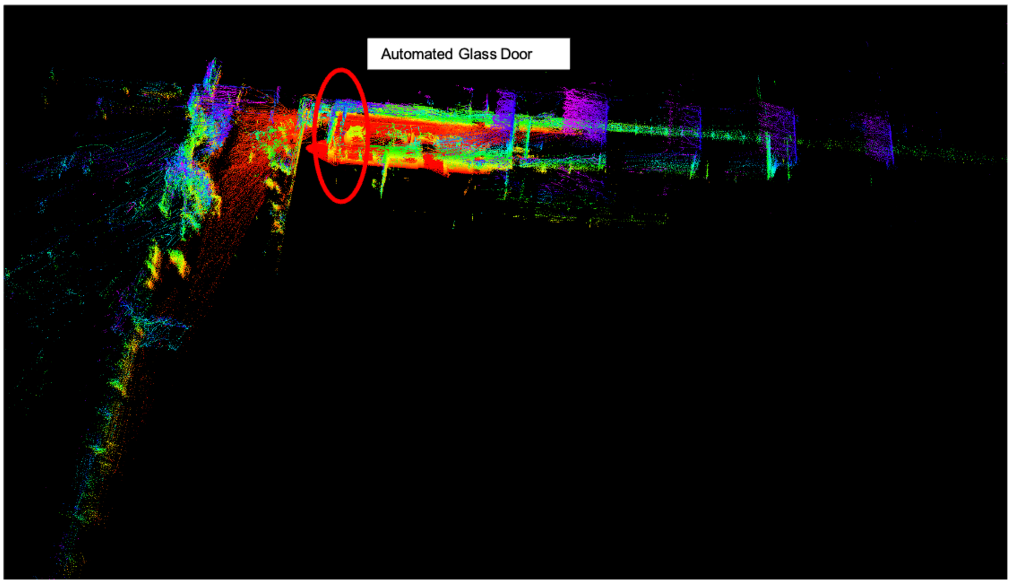
(c) Ground truth map

Figure 5.20 Mapping results of the corridor from Livox-LOAM, the proposed method and the ground truth.

A pair of sharp hook turn tests were conducted to investigate the performance of the proposed system, especially its odometry stability. From the results illustrated in Figure 5.22, where the



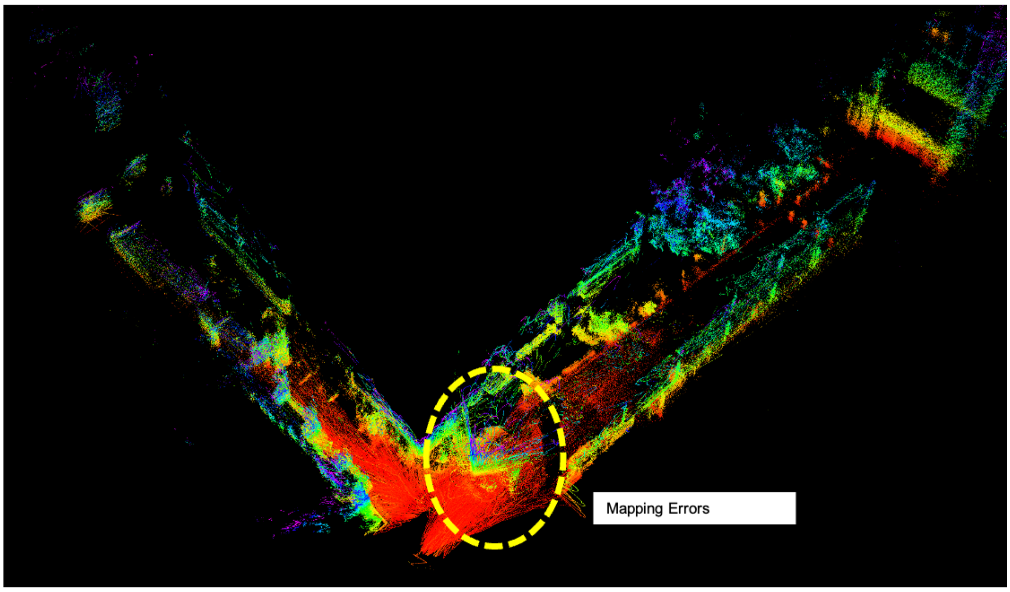
(a) Mapping result from Livox-LOAM.



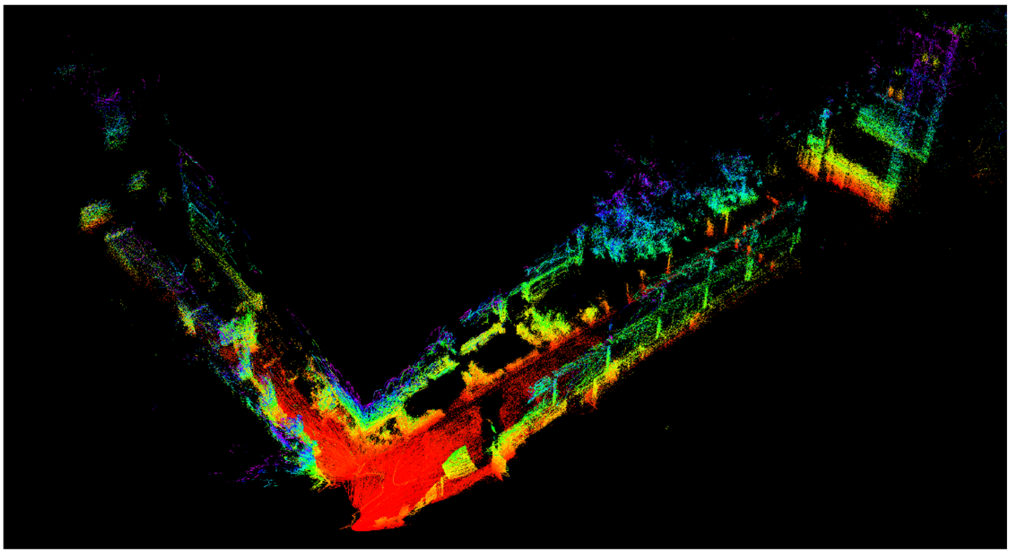
(b) Mapping result from our system.

Figure 5.21 Mapping through an automated glass door near the Engineering Building at theMonash University.

improvement of the proposed method over the Livox-LOAM is significant. Through observation, single SSL mapping is vulnerable to sizeable obstacles which occupying a large proportion of its FOV. In the test, while the robot is turning, it moves towards a large and featureless wall, which introduces error to the matching function.



(a) Mapping result from Livox-LOAM.



(b) Mapping result from our system.

Figure 5.22 Mapping results of a sharp hook turn action in the Monash University.

5.9 Discussion

This work presented a systemic approach to improve the mapping quality of a SLAM system which utilises a narrow FOV 3D SSL and a large FOV 2D LiDAR. The proposed system combines the advantage of both LiDARs to improve its performance, especially in feature points selection. Compared with the single LiDAR LOAM approaches, the work proposed in this chapter particularly contributes to the optimisation of 2D and 3D fusion on a multi-

LiDAR structure. The developed algorithm divided scanning window into sections based on the Manhattan-World assumption, which enables the algorithm to find more feature points in the scan range. Furthermore, it was demonstrated that 2D LiDAR could also contribute directly to the pose estimation under particular conditions. The proposed mixed odometry further stabilised the algorithm in challenging scenarios. With the conducted experiments, this work proves its capability to improve the stability of a SSL mapping algorithm by utilising 12 times more corner feature points. The proposed algorithm only recorded an APE of 5.64 in the experiment, which is significantly improved compare with the Livox-LOAM.

The system described is highly robust in tested scenarios. However, this comparison is limited as the proposed approach uses an extra 2D LiDAR than the Livox-LOAM system. In addition, enhancing the corner feature selection through FOV segmentation restricted the system's capability of selecting plane features. Moreover, the approach developed in this work takes advantage of urban terrain features. The effectiveness of applying this system to other terrains is still unstudied. Furthermore, the implemented dual-LiDAR pose estimation methodology does not take into account the motions in the z-axis, pitch or roll. Improving the system performance in these three axes requires further study.

Chapter 6

Conclusion and Future Directions

Existing LiDAR SLAM approaches generally rely on continuous feature extraction with LiDAR measurements. Researchers have made efforts to correctly identify features in scans and discover the geographical relationship between the consecutive scans. The continuity of the scan-matching process requires a SLAM system to operate in a controlled environment with a smooth trajectory. Such a SLAM system is sensitive to environmental change as well as rapid motions, thus limiting its application to industrial and consumer level scenarios. Losing feature matches would cause the optimisation method to fail to converge and results in less accurate system state estimation. Additionally, the errors recorded between the scans are accumulative during the SLAM process and eventually will lead to system failure. A more robust and general approach to improve the stability of the SLAM process would accelerate the development of SLAM technology and facilitate general-purpose mapping and localisation approaches.

Moreover, the development of LiDAR SLAM methodologies is increasing related to the innovation of LiDAR sensors. The severely limited Field-Of-View (FOV) and scan rate available on MEMS-based LiDAR sensors raise new research challenges for the system developers. SLAM algorithms are required to facilitate various scan patterns with narrow sensing window.

This thesis has presented the methodologies that permit high robustness mapping and localisation of LiDAR-based SLAM systems with a focus on map distortion, feature points extraction, and odometry stabilisation. Subsequent analysis of the proposed methods showed significant improvement in mapping stability compared with the existing studies reviewed in this thesis. Accompanying this study was the development of three sets of LiDAR-based SLAM systems, which demonstrated excellent performance of the methods presented in this thesis.

Two handheld LiDAR mapping devices were constructed for evaluating the proposed trajectory correction method. With a horizontally mounted 2D rotatory RpLiDAR, the first device adopted a supplementary trajectory from a laser interferometer-based tracker. Using Hector SLAM and laser tracker, the algorithm was able to generate two sets of independent trajectories, which then pass to the proposed trajectory correction strategy. Experimentation to determine the effectiveness of the trajectory correction approach was performed in an indoor room-size environment. Experimental results demonstrated that the proposed system is able to identify and correct localisation errors recorded by the Hector SLAM on the handheld device. The study was then extended to an indoor-outdoor mixed environment which resulted in the design of the second handheld device. Instead of laser tracker, the second handheld device uses a GPS module as the supplementary trajectory source. In addition, a mapping correction method was presented along with an asynchronous correction approach that schedules the system state correction according to the signal reception of the GPS. Through a comparison of the mapping outcome with the existing approaches, the experiments illustrated significant improvements of the proposed strategy in mapping stability and error corrections.

A study of feature extraction in 3D LiDAR SLAM algorithms, including feature points identification, pose estimation and map generation, was also presented, along with their identified performances. A novel 2D-3D mixed LiDAR SLAM system was designed and implemented, with a multi-LiDAR mapping unit developed for evaluation purposes. It was found that the proposed 2D-3D mixed LiDAR mapping approach could capture a considerably larger amount of feature points in the mapping process, and thus generate a more stable pose estimation during movement. As a result, a completed ground vehicle-based 3D SLAM solution was also presented in this work, which adopted the proposed 2D-3D mixed LiDAR SLAM approach. In addition, the ground vehicle platform uses 2D LiDAR readings to enhance the x-,y-axis and yaw motions in a 6 DOF system state model. The experimental results indicated that the interpolated 6 DOF pose estimation in the proposed approach outperforms the original 6 DOF odometry in urban terrain scenarios.

Finally, throughout the study, investigations were undertaken in various aspects of 2D and 3D SLAM approaches to improve system performances. The contributions of these studies include:

- The characterisation and modelling of SLAM localisation drift using trajectory comparison method, with a design of a system state correction method which uses the Iterative Closest Point (ICP) algorithm to estimate the accumulated drift in system states for both 2D and 3D SLAM systems.
- The development of a grid map correction mechanism which allows adjustment of 2D mapping results.
- Design and implementation of a Spherical-linear-interpolation-based (Slerp) based system state correction method and thus improving the smoothness of pose correction in a 3D SLAM system.
- A study of narrow FOV LiDAR and its effects on the LiDAR SLAM algorithms, with the design and implementation of a FOV enhanced multi-LiDAR mapping unit.
- The development of an enhanced 3D feature extraction algorithm that uses Manhattan-World-Assumption to improve 3D SLAM feature extraction.
- An odometry interpolation mechanism for multi-LiDAR odometry, which uses the residual from the Levenberg–Marquardt algorithm to evaluate the quality of an odometry update and dynamically interpolate its value.

6.1 Application discussion

In summary, this research study provided two kinds of completed LiDAR SLAM solutions. The first being a handheld LiDAR mapping solution suitable for indoor-outdoor large scale mapping applications. The system features high portability and robustness, which directly contribute to improving the quality of the resulting map. It is anticipated that the future application for such a 2D SLAM system would relate to complex environment autonomous ground vehicle systems, for example, a package delivery system. The proposed approaches are tested with GPS and laser interferometer-based tracker. The excellent robustness demonstrated by the developed system permit its usage in challenging mapping conditions including recovering from kidnapping and rollover.

Second LiDAR SLAM solution delivered by this research study is an urban scenario 3D LiDAR mapping system. The design of the system emphasised the required mapping robustness in scenarios where features are unstable. It is particularly designed for overcoming feature degradation in the mapping process. Combining the trajectory correction method proposed in the Section 4.3, the system demonstrated its outstanding performance in the mapping environment such as a university campus. Consequently, the mapping system is expected to be suitable for urban scene mapping tasks such as plant-scale 3D reconstruction.

6.2 Future works

There is a scope to improve the work documented in this thesis. This section discusses some of the possible modifications that could be made to the developed approach to further improve its effectiveness.

The trajectory matching method proposed in Chapter 2 uses Point-to-Line Iterative-Closest-Points (PL-ICP) to discover the geographical relationship between the two provided trajectories. This method demonstrated its improved accuracy in selected environments where the rotations caused by corners in the robot movement provide strong pattern features to the PL-ICP algorithm to identify the transformation matrix. However, by its own, the PL-ICP, and other ICP family of algorithms, do not tolerate rotational symmetry. In cases where a trajectory is symmetrical, the calculation will provide an untrusted result with a small residual. Thus, rotational invariance is required in the future to improve the accuracy of the algorithm. A possible approach to achieve rotational invariance in the trajectory matching is to attach orientation feature descriptor to the trajectory, such as a directional vector. Investigations are required to develop a suitable mechanism to generate orientation feature descriptor in a stable and real-time manner.

As demonstrated in Chapter 3, the map correction in 3D point clouds requires a significant amount of computing power. This operation was replaced by the Slerp system state relocation method in the proposed approach to reduce computational complexity. However, map correction in 3D point clouds is still a possible extension to this work. Other than oct-trees and nearest-neighbour-based algorithms, a more efficient and flexible 3D point clouds storage structure is therefore required to correct point clouds. A dynamic sub-mapping technique is desired to be developed, with a focus on low-cost point cluster translation and rotation.

The investigation of enhanced 3D LiDAR feature extraction in Chapter 4 demonstrated an improvement over the feature selection in mapping algorithms. However, based on the Manhattan-World-Assumption, the proposed algorithm only outperform the existing methodologies on ground vehicles in urban scenarios. Further study is necessary to extend the feature selection model to additional platforms and different environments, especially aerial-platforms with substantial movement on z-axis, pitch and roll.

References

[1] W. Wei, B. Shirinzadeh, S. Esakkiappan, M. Ghafarian and A. Al-Jodah, "Orientation Correction for Hector SLAM at Starting Stage," *2019 7th International Conference on Robot Intelligence Technology and Applications, RiTA 2019*, pp. 125–129, 2019, ISSN: 2340-9711. DOI: 10.1109/RITAPP.2019.8932722.

[2] W. Wei, B. Shirinzadeh, M. Ghafarian, S. Esakkiappan and T. Shen, "Hector SLAM with ICP trajectory matching," *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, vol. 2020-July, no. 1, pp. 1971–1976, 2020. DOI: 10.1109/AIM43001.2020.9158946.

[3] K. S. Arun, T. S. Huang and S. D. Blostein, "Least-Squares Fitting of Two 3-D Point Sets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-9, no. 5, pp. 698–700, 1987, ISSN: 01628828. DOI: 10.1109/TPAMI.1987.4767965.

[4] A. W. Fitzgibbon, "Robust registration of 2D and 3D point sets," DOI: 10.1016/j.imavis.2003.09.004. [Online]. Available: https://ac-els-cdn-com.ezproxy.lib.monash.edu.au/S0262885603001835/1-s2.0-S0262885603001835-main.pdf?%7B%5C_%7Dtid=36bf9e58-ff16-11e7-8289-00000aab0f6c%7B%5C&%7Dacdnat=1516585833%7B%5C_%7D30af114d070de69a01960126ffab4b20.

[5] Z. Zhang, "Iterative point matching for registration of free-form curves and surfaces," *International Journal of Computer Vision*, vol. 13, no. 2, pp. 119–152, 1994, ISSN: 09205691. DOI: 10.1007/BF01427149.

[6] A. Censi, "An ICP variant using a point-to-line metric," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2008, pp. 19–25, ISBN: 9781424416479. DOI: 10.1109/ROBOT.2008.4543181. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.329.6781%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[7] S. Rusinkiewicz and M. Levoy, "Efficient variants of the icp algorithm," in *Proceedings third international conference on 3-D digital imaging and modeling*, IEEE, 2001, pp. 145–152.

[8] J. Serafin and G. Grisetti, "Nicp: Dense normal based point cloud registration," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2015, pp. 742–749.

[9] A. Segal, D. Haehnel and S. Thrun, "Generalized-icp.," in *Robotics: science and systems*, Seattle, WA, vol. 2, 2009, p. 435.

[10] F. Moosmann and C. Stiller, "Velodyne slam," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, 2011, pp. 393–398.

[11] J. L. Martınez, J. González, J. Morales, A. Mandow and A. J. Garcıa-Cerezo, "Genetic and icp laser point matching for 2d mobile robot motion estimation," *chapter book in "Studies in computational intelligence" edited by stefano Cagnoni, Springer-Verlag Berlin Heidelberg*, 2009.

[12] R. Tiar, N. Ouadah, O. Azouaoui, M. Djehaich, H. Ziane and N. Achour, "ICP-SLAM methods implementation on a bi-steerable mobile robot," *Informal Proceedings of the 11th International Workshop of Electronics, Control, Measurement, Signals and Their Application to Mechatronics, ECMSM 2013*, no. March 2017, 2013. DOI: 10.1109/ECMSM.2013.6648973.

[13] R. Tiar, M. Lakrouf and O. Azouaoui, "Fast ICP-SLAM for a bi-steerable mobile robot in large environments," in *Proceedings of the 17th International Conference on Advanced Robotics, ICAR 2015*, 2015, pp. 611–616, ISBN: 9781467375092. DOI: 10.1109/ICAR.2015.7251519.

[14] Y.-J. Lee, J.-B. Song and J.-H. Choi, "Performance improvement of iterative closest point-based outdoor slam by rotation invariant descriptors of salient regions," *Journal of intelligent & robotic systems*, vol. 71, no. 3-4, pp. 349–360, 2013.

[15] S. Kohlbrecher, O. Von Stryk, J. Meyer and U. Klingauf, "A flexible and scalable SLAM system with full 3D motion estimation," *9th IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2011*, pp. 155–160, 2011, ISSN: 2374-3247. DOI: 10.1109/SSRR.2011.6106777.

[16] W. Hess, D. Kohler, H. Rapp and D. Andor, "Real-time loop closure in 2D LIDAR SLAM," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2016-June, pp. 1271–1278, 2016, ISSN: 10504729. DOI: 10.1109/ICRA.2016.7487258. arXiv: 1704.05959.

[17] P. Cheeseman, R. Smith and M. Self, "A stochastic map for uncertain spatial relationships," in *4th International Symposium on Robotic Research*, MIT Press Cambridge, 1987, pp. 467–474.

[18] T. Bailey, *Mobile robot localisation and mapping in extensive outdoor environments*. Citeseer, 2002.

[19] J. Guivant and E. Nebot, "Improving computational and memory requirements of simultaneous localization and map building algorithms," in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, IEEE, vol. 3, 2002, pp. 2731–2736.

[20] J. Leonard and P. Newman, "Consistent, convergent, and constant-time slam," in *IJCAI*, 2003, pp. 1143–1150.

[21] S. J. Julier and J. K. Uhlmann, "Using multiple slam algorithms," in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, IEEE, vol. 1, 2003, pp. 200–205.

[22] T. Bailey, J. Nieto, J. Guivant, M. Stevens and E. Nebot, "Consistency of the ekf-slam algorithm," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2006, pp. 3562–3568.

[23] S. Bonnable, P. Martin and E. Salaün, "Invariant extended kalman filter: Theory and application to a velocity-aided attitude estimation problem," in *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, IEEE, 2009, pp. 1297–1304.

[24] E. A. Wan and R. Van Der Merwe, "The unscented kalman filter for nonlinear estimation," in *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No. 00EX373)*, Ieee, 2000, pp. 153–158.

[25] S. Thrun, Y. Liu, D. Koller, A. Y. Ng, Z. Ghahramani and H. Durrant-Whyte, "Simultaneous localization and mapping with sparse extended information filters," *The international journal of robotics research*, vol. 23, no. 7-8, pp. 693–716, 2004.

[26] J. Wang and W. Chen, “An improved extended information filter slam algorithm based on omnidirectional vision,” *Journal of Applied Mathematics*, vol. 2014, 2014.

[27] M. Montemerlo, S. Thrun, D. Koller, B. Wegbreit *et al.*, “Fastslam: A factored solution to the simultaneous localization and mapping problem,” *Aaai/iaai*, vol. 593598, 2002.

[28] ——, “Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges,” in *IJCAI*, 2003, pp. 1151–1156.

[29] G. Grisetti, C. Stachniss and W. Burgard, “Improved techniques for grid mapping with Rao-Blackwellized particle filters,” *IEEE Transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007, ISSN: 15523098. DOI: 10.1109/TRO.2006.889486. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.1897%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[30] N. Andreas, “Improving Google ’ S Cartographer 3D Mapping By Continuous-Time Slam,” no. 2014, 2015.

[31] A. Nüchter, M. Bleier, J. Schauer and P. Janotta, “Continuous-time slam—improving google’s cartographer 3d mapping,” *Latest Developments in Reality-Based 3D Surveying and Modelling; MDPI: Basel, Switzerland*, pp. 53–73, 2018.

[32] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai and R. Vincent, “Efficient sparse pose adjustment for 2d mapping,” in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2010, pp. 22–29.

[33] J. Fossel, D. Hennes, D. Claes, S. Alers and K. Tuyls, “Octoslam: A 3d mapping approach to situational awareness of unmanned aerial vehicles,” in *2013 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, 2013, pp. 179–188.

[34] N. Kwak, O. Stasse, T. Foissotte and K. Yokoi, “3d grid and particle based slam for a humanoid robot,” in *2009 9th IEEE-RAS International Conference on Humanoid Robots*, IEEE, 2009, pp. 62–67.

[35] E. Einhorn and H.-M. Gross, “Generic 2d/3d slam with ndt maps for lifelong application,” in *2013 European Conference on Mobile Robots*, IEEE, 2013, pp. 240–247.

[36] J. Yang, H. Li, D. Campbell and Y. Jia, “Go-icp: A globally optimal solution to 3d icp point-set registration,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 11, pp. 2241–2254, 2015.

[37] J. Zhang and S. Singh, “Loam: Lidar odometry and mapping in real-time.,” in *Robotics: Science and Systems*, vol. 2, 2014.

[38] ——, “LOAM: Lidar Odometry and Mapping in Real- time,” *IEEE Transactions on Robotics*, vol. 32, no. July, pp. 141–148, 2015, ISSN: 15523098. DOI: 10.15607/RSS.2014.X.007. arXiv: 9605103 [cs].

[39] R. M. Murray, Z. Li, S. S. Sastry and S. S. Sastry, *A mathematical introduction to robotic manipulation*. CRC press, 1994.

[40] X. Liu, L. Zhang, S. Qin, D. Tian, S. Ouyang and C. Chen, “Optimized loam using ground plane constraints and segmatch-based loop detection,” *Sensors*, vol. 19, no. 24, p. 5419, 2019.

[41] M. Yan, J. Wang, J. Li and C. Zhang, “Loose coupling visual-lidar odometry by combining viso2 and loam,” in *2017 36th Chinese Control Conference (CCC)*, IEEE, 2017, pp. 6841–6846.

[42] C. Gonzalez and M. Adams, “An improved feature extractor for the lidar odometry and mapping (loam) algorithm,” in *2019 International Conference on Control, Automation and Information Sciences (ICCAIS)*, IEEE, 2019, pp. 1–7.

[43] T. Shan and B. Englot, "LeGO-LOAM: Lightweight and Ground-Optimized Lidar Odometry and Mapping on Variable Terrain," *IEEE International Conference on Intelligent Robots and Systems*, no. September 2019, pp. 4758–4765, 2018, ISSN: 21530866. DOI: 10.1109/IROS.2018.8594299.

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

[45] B. Schoen-Phelan, A. S. M. Mosa, D. Laefer and M. Bertolotto, "Octree-based indexing for 3d point clouds within an oracle spatial dbms," 2013.

[46] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, no. 3, pp. 189–206, 2013, ISSN: 09295593. DOI: 10.1007/s10514-012-9321-0.

[47] K. Wenzel, M. Rothermel, D. Fritsch and N. Haala, "An out-of-core octree for massive point cloud processing," in *PROCEEDINGS, IQMULUS 1ST WORKSHOP ON PROCESSING LARGE GEOSPATIAL DATA*, 2014, p. 53.

[48] S. Han, S. Kim, J. H. Jung, C. Kim, K. Yu and J. Heo, "Development of a hashing-based data structure for the fast retrieval of 3d terrestrial laser scanned data," *Computers & Geosciences*, vol. 39, pp. 1–10, 2012.

[49] H. Xue, H. Fu, R. Ren, T. Wu and B. Dai, "Real-time 3D Grid Map Building for Autonomous Driving in Dynamic Environment," pp. 40–45, 2020. DOI: 10.1109/icus48101.2019.8996066.

[50] E. A. L. Narváez and N. E. L. Narváez, "Point cloud denoising using robust principal component analysis.," in *GRAPP*, 2006, pp. 51–58.

[51] F. Zaman, Y. P. Wong and B. Y. Ng, "Density-based denoising of point cloud," in *9th International Conference on Robotic, Vision, Signal Processing and Power Applications*, Springer, 2017, pp. 287–295.

[52] O. Schall, A. Belyaev and H.-P. Seidel, "Adaptive feature-preserving non-local denoising of static and time-varying range data," *Computer-Aided Design*, vol. 40, no. 6, pp. 701–707, 2008.

[53] O. Schall, A. Belyaev and H.-P. Seidel, "Robust filtering of noisy scattered point data," in *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, IEEE, 2005, pp. 71–144.

[54] G. Guennebaud, M. Germann and M. Gross, "Dynamic sampling and rendering of algebraic point set surfaces," in *Computer Graphics Forum*, Wiley Online Library, vol. 27, 2008, pp. 653–662.

[55] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin and C. T. Silva, "Computing and rendering point set surfaces," *IEEE Transactions on visualization and computer graphics*, vol. 9, no. 1, pp. 3–15, 2003.

[56] P. Jenke, M. Wand, M. Bokeloh, A. Schilling and W. Straßer, "Bayesian point cloud reconstruction," in *Computer Graphics Forum*, Wiley Online Library, vol. 25, 2006, pp. 379–388.

[57] G. Taubin, "Estimating the tensor of curvature of a surface from a polyhedral approximation," in *Proceedings of IEEE International Conference on Computer Vision*, IEEE, 1995, pp. 902–907.

[58] C. Lange and K. Polthier, "Anisotropic smoothing of point sets," *Computer Aided Geometric Design*, vol. 22, no. 7, pp. 680–692, 2005.

[59] U. Clarenz, M. Rumpf and A. Telea, *Fairing of point based surfaces*. IEEE, 2004.

[60] J. Fuentes-Pacheco, J. Ruiz-Ascencio and J. M. Rendón-Mancha, "Visual simultaneous localization and mapping: A survey," *Artificial intelligence review*, vol. 43, no. 1, pp. 55–81, 2015.

[61] A. J. Davison, I. D. Reid, N. D. Molton and O. Stasse, "Monoslam: Real-time single camera slam," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1052–1067, 2007.

[62] G. Klein and D. Murray, "Parallel tracking and mapping for small ar workspaces," in *2007 6th IEEE and ACM international symposium on mixed and augmented reality*, IEEE, 2007, pp. 225–234.

[63] J. Shi and C. Tomasi, *Good features to track. 9th ieee conference on computer vision and pattern recognition*, 1994.

[64] R. Mur-Artal, J. M. M. Montiel and J. D. Tardos, "Orb-slam: A versatile and accurate monocular slam system," *IEEE transactions on robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[65] R. Mur-Artal and J. D. Tardós, "Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[66] C. Campos, R. Elvira, J. J. G. Rodriguez, J. M. Montiel and J. D. Tardós, "Orb-slam3: An accurate open-source library for visual, visual-inertial and multi-map slam," *arXiv preprint arXiv:2007.11898*, 2020.

[67] R. A. Newcombe, S. J. Lovegrove and A. J. Davison, "Dtam: Dense tracking and mapping in real-time," in *2011 international conference on computer vision*, IEEE, 2011, pp. 2320–2327.

[68] J. Engel, T. Schöps and D. Cremers, "Lsd-slam: Large-scale direct monocular slam," in *European conference on computer vision*, Springer, 2014, pp. 834–849.

[69] C. Forster, Z. Zhang, M. Gassner, M. Werlberger and D. Scaramuzza, "Svo: Semi-direct visual odometry for monocular and multicamera systems," *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 249–265, 2016.

[70] Y. Yang, G. Yang, Y. Tian, T. Zheng, L. Li and Z. Wang, "A robust and accurate SLAM algorithm for omni-directional mobile robots based on a novel 2.5D lidar device," *Proceedings of the 13th IEEE Conference on Industrial Electronics and Applications, ICIEA 2018*, pp. 2123–2127, 2018. DOI: 10.1109/ICIEA.2018.8398060.

[71] X. Kang, S. Yin and Y. Fen, "3D reconstruction assessment framework based on affordable 2D lidar," *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, vol. 2018-July, pp. 292–297, 2018. DOI: 10.1109/AIM.2018.8452242. arXiv: 1803.09167.

[72] L. Jian, I. Qiang, Z. Yang, L. Huican and W. Heng, "A kinectV2-based 2D Indoor SLAM Method," in *Proceedings of the 2017 International Conference on Artificial Intelligence, Automation and Control Technologies - AIACT '17*, New York, New York, USA: ACM Press, 2017, pp. 1–5, ISBN: 9781450352314. DOI: 10.1145/3080845.3080877. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3080845.3080877.

[73] W.-C. Jiang, "Implementation of Odometry with EKF in Hector SLAM Methods," *International Journal of Automation and Smart Technology*, vol. 8, no. 1, pp. 9–18, 2018. DOI: 10.5875/ausmt.v8i1.1558.

[74] N. Yu and B. Zhang, "An Improved Hector SLAM Algorithm based on Information Fusion for Mobile Robot," *Proceedings of 2018 5th IEEE International Conference on Cloud Computing and Intelligence Systems, CCIS 2018*, pp. 279–284, 2019. DOI: 10.1109/CCIS.2018.8691198.

[75] A. Bassiri, M. Asghari Oskoei, A. Basiri and L. L. Li, "Particle filter and finite impulse response filter fusion and hector SLAM to improve the performance of robot positioning," *Journal of Robotics*, vol. 2018, pp. 278–283, Oct. 2018, ISSN: 16879619. DOI: 10.1155/2018/7806854. [Online]. Available: http://ieeexplore.ieee.org/document/7451625/.

[76] D. Caruso, A. Eudes, M. Sanfourche, D. Vissière and G. Le Besnerais, "An inverse square root filter for robust indoor/outdoor magneto-visual-inertial odometry," *2017 International Conference on Indoor Positioning and Indoor Navigation, IPIN 2017*, vol. 2017-Janua, no. September, pp. 1–8, 2017. DOI: 10.1109/IPIN.2017.8115888.

[77] L.-H. Chen and C.-C. Peng, "A Robust 2D-SLAM Technology with Environmental Variation Adaptability," *IEEE Sensors Journal*, vol. 19, no. 23, pp. 1–1, 2019, ISSN: 1530-437X. DOI: 10.1109/jsen.2019.2931368.

[78] P. Kim, J. Chen and Y. K. Cho, "SLAM-driven robotic mapping and registration of 3D point clouds," *Automation in Construction*, vol. 89, no. February, pp. 38–48, 2018, ISSN: 09265805. DOI: 10.1016/j.autcon.2018.01.009.

[79] V. Kubelka, M. Reinstein and T. Svoboda, "Tracked Robot Odometry for Obstacle Traversal in Sensory Deprived Environment," *IEEE/ASME Transactions on Mechatronics*, vol. 24, no. 6, pp. 1–1, 2019, ISSN: 1083-4435. DOI: 10.1109/tmech.2019.2945031.

[80] A. Azim and O. Aycard, "Detection, classification and tracking of moving objects in a 3d environment," in *2012 IEEE Intelligent Vehicles Symposium*, IEEE, 2012, pp. 802–807.

[81] Y.-J. Lee and J.-B. Song, "Three-dimensional iterative closest point-based outdoor slam using terrain classification," *Intelligent Service Robotics*, vol. 4, no. 2, pp. 147–158, 2011.

[82] D. C. Asmar, J. S. Zelek and S. M. Abdallah, "Smartslam: Localization and mapping across multi-environments," in *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*, IEEE, vol. 6, 2004, pp. 5240–5245.

[83] C. Brand, M. J. Schuster, H. Hirschmüller and M. Suppa, "Stereo-vision based obstacle mapping for indoor/outdoor SLAM," *IEEE International Conference on Intelligent Robots and Systems*, no. Iros, pp. 1846–1853, 2014, ISSN: 21530866. DOI: 10.1109/IROS.2014.6942805.

[84] E. Dill, M. U. De Haag, P. Duan, D. Serrano and S. Vilardaga, "Seamless indoor-outdoor navigation for unmanned multi-sensor aerial platforms," *Record - IEEE PLANS, Position Location and Navigation Symposium*, pp. 1174–1182, 2014. DOI: 10.1109/PLANS.2014.6851489.

[85] J. Collier and A. Ramirez-Serrano, "Environment classification for indoor/outdoor robotic mapping," *Proceedings of the 2009 Canadian Conference on Computer and Robot Vision, CRV 2009*, pp. 276–283, 2009. DOI: 10.1109/CRV.2009.6.

[86] Y. J. Lee and J. B. Song, "Three-dimensional iterative closest point-based outdoor SLAM using terrain classification," *Intelligent Service Robotics*, vol. 4, no. 2, pp. 147–158, 2011, ISSN: 18612776. DOI: 10.1007/s11370-011-0087-6.

[87] J. Qutteineh, "Investigation of Cooperative SLAM for Low Cost Indoor Robots," 2016.

[88] J. Folkesson, P. Jensfelt and H. I. Christensen, "Vision slam in the measurement subspace," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, IEEE, 2005, pp. 30–35.

[89] S. Lee, S. Lee and S. Baek, "Vision-based kidnap recovery with SLAM for home cleaning robots," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 67, no. 1, pp. 7–24, 2012, ISSN: 15730409. DOI: 10.1007/s10846-011-9647-4.

[90] K. Ni, D. Steedly and F. Dellaert, "Tectonic sam: Exact, out-of-core, submap-based slam," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, IEEE, 2007, pp. 1678–1685.

[91] C. Brand, M. J. Schuster, H. Hirschmuller and M. Suppa, "Submap matching for stereo-vision based indoor/outdoor SLAM," *IEEE International Conference on Intelligent Robots and Systems*, vol. 2015-Decem, pp. 5670–5677, 2015, ISSN: 21530866. DOI: 10.1109/IROS.2015.7354182.

[92] P. Newman, D. Cole and K. Ho, "Outdoor slam using visual appearance and laser ranging," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, IEEE, 2006, pp. 1180–1187.

[93] P. Newman and K. Ho, "Slam-loop closing with visually salient features," in *proceedings of the 2005 IEEE International Conference on Robotics and Automation*, IEEE, 2005, pp. 635–642.

[94] Y. Liu and H. Zhang, "Visual loop closure detection with a compact image descriptor," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 1051–1056.

[95] H. Zhang, "Borf: Loop-closure detection with scale invariant visual features," in *2011 IEEE International conference on robotics and automation*, IEEE, 2011, pp. 3125–3130.

[96] F. T. Ramos, D. Fox and H. F. Durrant-Whyte, "Crf-matching: Conditional random fields for feature-based scan matching.," in *Robotics: Science and Systems*, 2007.

[97] F. Campos, L. Correia and J. Calado, "Mobile robot global localization with non-quantized sift features," Jun. 2011, pp. 582–587, ISBN: 978-1-4577-1158-9. DOI: 10.1109/ICAR.2011.6088564.

[98] D. Gálvez-López and J. D. Tardos, "Bags of binary words for fast place recognition in image sequences," *IEEE Transactions on Robotics*, vol. 28, no. 5, pp. 1188–1197, 2012.

[99] F. Caballero, J. Pérez and L. Merino, "Long-term ground robot localization architecture for mixed indoor-outdoor scenarios," *Proceedings for the Joint Conference of ISR 2014 - 45th International Symposium on Robotics and Robotik 2014 - 8th German Conference on Robotics, ISR/ROBOTIK 2014*, pp. 21–28, 2014.

[100] T. Qin, P. Li and S. Shen, "Vins-mono: A robust and versatile monocular visual-inertial state estimator," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 1004–1020, 2018.

[101] A. Rosinol, M. Abate, Y. Chang and L. Carlone, "Kimera: An open-source library for real-time metric-semantic localization and mapping," *arXiv preprint arXiv:1910.02490*, 2019.

[102] R. Ravi, Y. J. Lin, M. Elbahnasawy, T. Shamseldin and A. Habib, "Simultaneous System Calibration of a Multi-LiDAR Multicamera Mobile Mapping Platform," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 11, no. 5, pp. 1694–1714, 2018, ISSN: 21511535. DOI: 10.1109/JSTARS.2018.2812796.

[103] J. Jiao, H. Ye, Y. Zhu and M. Liu, "Robust Odometry and Mapping for Multi-LiDAR Systems with Online Extrinsic Calibration," pp. 1–20, 2020. arXiv: 2010.14294. [Online]. Available: http://arxiv.org/abs/2010.14294.

[104] M. Sualeh and G. W. Kim, "Dynamic Multi-LiDAR based multiple object detection and tracking," *Sensors (Switzerland)*, vol. 19, no. 6, 2019, ISSN: 14248220. DOI: 10.3390/s19061474.

[105] T. Kim and T. Park, “Calibration method between dual 3D lidar sensors for autonomous vehicles,” *2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan, SICE 2017*, vol. 2017-Novem, pp. 1075–1081, 2017. DOI: 10.23919/SICE.2017.8105583.

[106] H. Andreasson and A. J. Lilienthal, “6D scan registration using depth-interpolated local image features,” *Robotics and Autonomous Systems*, vol. 58, no. 2, pp. 157–165, 2010, ISSN: 09218890. DOI: 10.1016/j.robot.2009.09.011.

[107] H. Cai, W. Pang, X. Chen, Y. Wang and H. Liang, “A Novel Calibration Board and Experiments for 3D LiDAR and Camera Calibration,” DOI: 10.3390/s20041130. [Online]. Available: www.mdpi.com/journal/sensors.

[108] A. R. Willis, M. J. Zapata and J. M. Conrad, “A linear method for calibrating LIDAR-and-camera systems,” *Proceedings - IEEE Computer Society’s Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*, no. August 2015, pp. 577–579, 2009, ISSN: 15267539. DOI: 10.1109/MASCOT.2009.5366801.

[109] P. Olivka, M. Krumnikl, P. Moravec and D. Seidl, “Calibration of Short Range 2D Laser Range Finder for 3D SLAM Usage,” *Journal of Sensors*, vol. 2016, 2016, ISSN: 16877268. DOI: 10.1155/2016/3715129.

[110] Q. Zhang and R. Pless, “Extrinsic calibration of a camera and laser range finder (improves camera calibration),” *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, no. January 2004, pp. 2301–2306, 2004. DOI: 10.1109/iros.2004.1389752.

[111] M. Pereira, D. Silva, V. Santos and P. Dias, “Self calibration of multiple LIDARs and cameras on autonomous vehicles,” *Robotics and Autonomous Systems*, vol. 83, pp. 326–337, 2016, ISSN: 09218890. DOI: 10.1016/j.robot.2016.05.010.

[112] N. Heide, T. Emter and J. Petereit, “Calibration of multiple 3D LiDAR sensors to a common vehicle frame,” *50th International Symposium on Robotics, ISR 2018*, pp. 372–379, 2018.

[113] J. Jiao, Y. Yu, Q. Liao, H. Ye and M. Liu, “Automatic calibration of multiple 3D LiDARs in urban environments,” *arXiv*, pp. 15–20, 2019.

[114] M. Sheehan, A. Harrison and P. Newman, “Automatic self-calibration of a full field-of-view 3D n-Laser scanner,” *Springer Tracts in Advanced Robotics*, vol. 79, pp. 165–178, 2014, ISSN: 1610742X. DOI: 10.1007/978-3-642-28572-1_12.

[115] H. Sommer, R. Khanna, I. Gilitschenski, Z. Taylor, R. Siegwart and J. Nieto, “A low-cost system for high-rate, high-accuracy temporal calibration for LIDARs and cameras,” *IEEE International Conference on Intelligent Robots and Systems*, vol. 2017-Septe, pp. 2219–2226, 2017, ISSN: 21530866. DOI: 10.1109/IROS.2017.8206042.

[116] C. Walters, O. Mendez, S. Hadfield and R. Bowden, “A Robust Extrinsic Calibration Framework for Vehicles with Unscaled Sensors,” in *IEEE International Conference on Intelligent Robots and Systems*, 2019, pp. 36–42, ISBN: 9781728140049. DOI: 10.1109/IROS40897.2019.8968244.

[117] M. Bosse and R. Zlot, “Continuous 3D scan-matching with a spinning 2D laser,” no. June 2009, pp. 4312–4319, 2009, ISSN: 1050-4729. DOI: 10.1109/robot.2009.5152851.

[118] ——, “Continuous 3D scan-matching with a spinning 2D laser,” *2009 IEEE International Conference on Robotics and Automation*, pp. 4312–4319, 2009, ISSN: 1050-4729. DOI: 10.1109/ROBOT.2009.5152851. [Online]. Available: http://ieeexplore.ieee.org/document/5152851/.

[119] A. Pfrunder, P. V. K. Borges, A. R. Romero, G. Catt and A. Elfes, “Real-Time Autonomous Ground Vehicle Navigation in Heterogeneous Environments Using a 3D LiDAR,” pp. 2601–2608, 2017. DOI: 10.0/Linux-x86_64.

[120] I. Baldwin and P. Newman, “Laser-only road-vehicle localization with dual 2D push-broom LIDARS and 3D priors,” *IEEE International Conference on Intelligent Robots and Systems*, pp. 2490–2497, 2012, ISSN: 21530858. DOI: 10.1109/IROS.2012.6385677.

[121] S. Hu, D. Wang and S. Xu, “3D indoor modeling using a hand-held embedded system with multiple laser range scanners,” no. October 2016, p. 101552D, 2016, ISSN: 1996756X. DOI: 10.1117/12.2247006. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.2247006.

[122] P. Geneva and K. Eckenhoff, “LIPS: LiDAR-Inertial 3D Plane SLAM [TR],” 2018.

[123] Z. J. Chong, B. Qin, T. Bandyopadhyay, M. H. Ang, E. Frazzoli and D. Rus, “Synthetic 2D LIDAR for precise vehicle localization in 3D urban environment,” *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1554–1559, 2013, ISSN: 10504729. DOI: 10.1109/ICRA.2013.6630777.

[124] Q. Sun, J. Yuan, X. Zhang and F. Sun, “RGB-D SLAM in Indoor Environments with STING-Based Plane Feature Extraction,” *IEEE/ASME Transactions on Mechatronics*, vol. 23, no. 3, pp. 1071–1082, 2018, ISSN: 10834435. DOI: 10.1109/TMECH.2017.2773576.

[125] G. Jiang, L. Yin, S. Jin, C. Tian, X. Ma and Y. Ou, “A simultaneous localization and mapping (SLAM) framework for 2.5D map building based on low-cost LiDAR and vision fusion,” *Applied Sciences (Switzerland)*, vol. 9, no. 10, 2019, ISSN: 20763417. DOI: 10.3390/app9102105. [Online]. Available: www.mdpi.com/journal/applsci.

[126] J. M. Coughlan, “Manhattan World : Compass Direction from a Single Image by Bayesian Inference 2 Previous Work and Three- Dimen- sional Geometry,” *Camera*, vol. 00, no. c, 1999.

[127] J. M. Coughlan and A. L. Yuille, “Manhattan world: Orientation and outlier detection by bayesian inference,” *Neural Computation*, vol. 15, no. 5, pp. 1063–1088, 2003.

[128] K. S. Shankar and T. L. Harmon, *Introduction to Robotics*, 2. 1986, vol. 1, pp. 108–108, ISBN: 0201182408. DOI: 10.1109/MEX.1986.4306961. arXiv: 0712.0689. [Online]. Available: http://ieeexplore.ieee.org/document/4306961/.

[129] C. V. Poulton, A. Yaacobi, D. B. Cole, M. J. Byrd, M. Raval, D. Vermeulen and M. R. Watts, “Coherent solid-state LIDAR with silicon photonic optical phased arrays,” *Optics Letters*, vol. 42, no. 20, p. 4091, 2017, ISSN: 0146-9592. DOI: 10.1364/ol.42.004091.

[130] H. W. Yoo, N. Druml, D. Brunner, C. Schwarzl, T. Thurner, M. Hennecke and G. Schitter, “MEMS-based lidar for autonomous driving,” *Elektrotechnik und Informationstechnik*, vol. 135, no. 6, pp. 408–415, 2018, ISSN: 0932383X. DOI: 10.1007/s00502-018-0635-2. [Online]. Available: http://dx.doi.org/10.1007/s00502-018-0635-2.

[131] A. Asvadi, L. Garrote, C. Premebida, P. Peixoto and U. J. Nunes, “Multimodal vehicle detection: fusing 3D-LIDAR and color camera data,” *Pattern Recognition Letters*, vol. 115, pp. 20–29, 2018, ISSN: 01678655. DOI: 10.1016/j.patrec.2017.09.038. [Online]. Available: https://doi.org/10.1016/j.patrec.2017.09.038.

[132] J. Lin and F. Zhang, “Loam_livox: A fast, robust, high-precision LiDAR odometry and mapping package for LiDARs of small FoV,” in *IEEE International Conference on Robotics and Automation*, Sep. 2019. arXiv: 1909.06700. [Online]. Available: http://arxiv.org/abs/1909.06700.

[133] Y. S. Shin and A. Kim, "Sparse depth enhanced direct thermal-infrared SLAM beyond the visible spectrum," *IEEE ROBOTICS AND AUTOMATION LETTERS*, vol. 4, no. 3, pp. 2918–2925, 2019.

[134] S. Sabatini, M. Carno, S. Fiorenti and S. M. Savaresi, "Improving Occupancy Grid Mapping via Dithering for a Mobile Robot Equipped with Solid-State LiDAR Sensors," *2018 IEEE Conference on Control Technology and Applications, CCTA 2018*, pp. 1145–1150, 2018. DOI: 10.1109/CCTA.2018.8511318.

[135] J. Lin, X. Liu and F. Zhang, "A decentralized framework for simultaneous calibration, localization and mapping with multiple LiDARs," no. 2, 2020. arXiv: 2007.01483. [Online]. Available: http://arxiv.org/abs/2007.01483.

[136] R. B. Rusu and S. Cousins, "3D is here: point cloud library," *IEEE International Conference on Robotics and Automation*, pp. 1–4, 2011, ISSN: 1050-4729. DOI: 10.1109/ICRA.2011.5980567. arXiv: 1409.1556. [Online]. Available: http://pointclouds.org/.

[137] A. Geiger, P. Lenz and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

Appendix A

Codes developed and utilised

This appendix listed some of the code developed and utilised in this study. All the code listed here are following *apache* – 2.0 open source license.

A.1 Hardware drivers

A.1.1 API laser tracker driver

The code documented in this section belongs to the API laser interferometer-based tracker. This program communicate between the laser tracker and the ROS network.

A.1.1.1 ROS to laser tracker bridge

```
// APIClient.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include <iostream>
#include "TrackerAPI.h"
#pragma comment(lib, "C:\\Users\\Lucas\\source\\repos\\APIClient\\Debug\\libs\\TrackerAPI
    .lib")
char c = ' ';
char * ptr = &c;
int main()
{
  CTracker myTracker;
  //void getWinnerApiObject(char*);
  //void getTrackerApiRevision(char*);
  //myTracker.getModelNumber(ptr);
  return 0;
}

  return 0;
}
```

A.1.1.2 API laser tracker interface

```
/*  Project Tracker API
  Automated Precision, Inc.
  Copyright  1998-2001 by Automated Precision, In
   SUBSYSTEM:    TrackerAPI.dll Application
   FILE:         TrackerAPI.cpp
   AUTHOR:       Christopher Eunsoo  LEE
   OVERVIEW
   ========
   Header file for implementation of CTracker Class.
*/
#ifndef _TRACKERAPI_
#define _TRACKERAPI_
// Standard DLL Macro Stuff
//
#if !defined (__TRACKER)
#define TRACKERAPI __declspec(dllimport)
#else
#define TRACKERAPI __declspec(dllexport)
#endif
#include <time.h>
#include <windows.h>
#define No_of_SysParams 135
#define Number_of_Factors 24
#define Number_of_Functions 14
static int ErrorMessageNumber;         // Error Message Number
#pragma pack(8)
typedef struct {
  bool       Warm_Up_Time;      //  true:System Warming-Up false:System is ready.
  bool       Laser_Path_Error;    //  true:Laser Beam-Path Error false:Laser Beam-Path is O.K.
  bool       Laser_Dist_Error;    //  true:Laser Distance Status Error false:Distance Status is O.K.
  bool       External_Switch;     //  true:External Switch Contact false:External Switch Open
  bool       Filtering_Switch;    //  true:Filtering Switch On false:Filtering Switch Off
  clock_t      Time_Stamp;         //  Tracker Time Stamp in ms
  unsigned char Operation_Mode;      //  Tracker Operation Mode
                      //  0 -> Servo Free Mode (Idle)
                      //  1 -> Servo Engaged Mode (Servo)
                      //  2 -> Tracking Mode (Track)
                      //  3 -> Losing target during Tracking (Track Idle)
                      //  4 -> Not Used by API user (Internal Use Only)
                      //  5 -> Searching the Encoder Index (Index Searching)
                      //  6 -> Not Used by API user (Internal Use Only)
                      //  7 -> Target Scan Search Mode (Search)
                      //  8 -> AZ Axis Motor Run-Away
                      //  9 -> EL Axis Motor Run-Away
  float      Laser_Intensity;     //  The Laser intensity is between 0.0-1.0
  float      Laser_Distance;      //  If the tracker mode is not in the Tracking Mode, the Laser Distance is 0.
  float      Current_Position_AZ;  //  Azimuth in Degree
  float      Current_Position_EL;  //  Elevation in Degree
  float      Air_Temperature;     //  Weather Sensor Information : Air Temperature (Centigrade)
  float      Air_Pressure;      //  Weather Sensor Information : Air Pressure  (mm/Hg)
  float      Current_Position_X;   //  X in mm
  float      Current_Position_Y;   //  Y in mm
  float      Current_Position_Z;   //  Z in mm
  float      Target_Velocity;     //  Velocity (mm/sec)
  float      Photo_X;          //  Calibrated Photo Sensor Eccentric in mm (X-direction)
  float      Photo_Y;          //  Calibrated Photo Sensor Eccentric in mm (Y-direction)
  float      Level_X;          //  Calibrated Level Sensor in ArcSec (X-direction)
  float      Level_Y;          //  Calibrated Level Sensor in ArcSec (Y-direction)
  bool       LevelX_OverLimit;    //  true:Out of the limit false:Within limit
  bool       LevelY_OverLimit;    //  true:Out of the limit false:Within limit
} REALTIME_INFO;
#pragma pack(8)
typedef struct {
  unsigned long Captured_Points;    // Captured Number Of Points in the FIFO.
  unsigned long Retrieved_Points;   // Retrieved Number Of Points from the FIFO.
} FIFO_INFO;
#pragma pack(8)
typedef struct {
  clock_t      Time_Stamp;         //  Tracker Time Stamp in ms
  unsigned char Operation_Mode;      //  Tracker Operation Mode
                      //  0 -> Servo Free Mode (Idle)
```

```
                        //  1 -> Servo Engaged Mode (Servo)
                        //  2 -> Tracking Mode (Track)
                        //  3 -> Losing target during Tracking (Track Idle)
                        //  4 -> Not Used by API user (Internal Use Only)
                        //  5 -> Searching the Encoder Index (Index Searching)
                        //  6 -> Not Used by API user (Internal Use Only)
                        //  7 -> Target Scan Search Mode (Search)
  float     Laser_Distance;     //  If the tracker mode is not in the Tracking Mode, the
    Laser Distance is 0.
  float     Current_Position_AZ;  //  Azimuth in Degree
  float     Current_Position_EL;  //  Elevation in Degree
  float     Current_Position_X;   //  X in mm
  float     Current_Position_Y;   //  Y in mm
  float     Current_Position_Z;   //  Z in mm
  float     Level_X;        //  Calibrated Level Sensor in ArcSec (X-direction)
  float     Level_Y;        //  Calibrated Level Sensor in ArcSec (Y-direction)
  bool      LevelX_OverLimit;   //  true:Out of the limit false:Within limit
  bool      LevelY_OverLimit;   //  true:Out of the limit false:Within limit
} FIFO_RECORD;
#pragma pack(8)
typedef struct {
  bool  JogMode;    // true:Absolute Jogging, false:Incremental Jogging
  float Azimuth;    // Azimuth Jogging Target
  float Elevation;    // Elevation Jogging Target
} TARGET;
#pragma pack(8)
typedef struct {
  bool  JogMode;    // true:Absolute Jogging, false:Incremental Jogging
  bool  InPosition;   // true:Check the in-positioning, false:Fly on the jog operation
  float Azimuth;    // Azimuth Jogging Target
  float Elevation;    // Elevation Jogging Target
} TARGET_EXTENDED;
#pragma pack(8)
typedef struct {
  float     Reset_Laser_Distance; //  If the tracker mode is not in the Tracking Mode,
    the Laser Distance is 0.
  float     Reset_Position_AZ;    //  Azimuth in Degree
  float     Reset_Position_EL;    //  Elevation in Degree
} RESET_ANGULAR, ANGULAR_DATA;
#pragma pack(8)
typedef struct {
  float     Reset_Position_X;   //  X in mm
  float     Reset_Position_Y;   //  Y in mm
  float     Reset_Position_Z;   //  Z in mm
} RESET_CARTESIAN, CARTESIAN_DATA;
#pragma pack(8)
typedef struct {
  double x;
  double y;
  double z;
  double w;
} D_VECTOR4;
#pragma pack(8)
typedef struct {
  double x;
  double y;
  double z;
} D_VECTOR3;
#pragma pack(8)
typedef struct {
  double x;
  double y;
} D_VECTOR2;

//////////////////////////////////////////////////////////////////////////////
// CTracker
// See TrackerAPI.cpp for the implementation of this class
//
class TRACKERAPI CTracker
{
// For the Internal Use Purpose Only.
private:
  unsigned char SerialPortIndex, BaudRateIndex;
  unsigned char Successful_Initialization;
  PVOID pAPItracker;
  REALTIME_INFO TrackerMonitoringBuffer;
  REALTIME_INFO Real_Time_Data;
  bool  RealTime_Capturing;
```

```
unsigned long m_lNumber_of_Points;
unsigned long m_lCapturedPoints;
unsigned long m_lRetrievedPoints;
float m_fRequiredParameter;
clock_t m_tWaitingDelay, TriggerTimer;
float m_fMinimumTriggerDistance;
float m_fVelocityBand;
bool  m_bSemaphore_CriticalSection;
unsigned char ProcedureSequencer[Number_of_Functions];
unsigned char StabilityCounter;
double  Reset_Inposition_Band;
D_VECTOR4 Reset_Point;
D_VECTOR4 Photo_Diff;
D_VECTOR2 Current_AZEL;
double  PriorValues[8];
unsigned long m_lPoints;
int   ExponentialFilterWeight;
int   Number_of_AZ_Data, Number_of_EL_Data;
float*  pElevation_Table;
float*  pAzimuth_Table;
void* pTrackerSystemParam;
char  PrmFileName[16], ModelNumber[16], SerialNumber[16], LicenseOwner[32];
float PhotoIntensity_X, PhotoIntensity_Y,
    CalibrationFactors[Number_of_Factors], CalculatedCalibrationFactors[6],
    Uncalibrated_Distance, Reset_Distance;
D_VECTOR4 CurrentAngle;
double  BirdBathAngle, BirdBathDistance,
    Gimbal_Coeff, Gimbal_Temp, In_Position[2], Softlimits[4];
FIFO_INFO FIFO_Info;
clock_t ReadingTimer, DelayTimer, WeatherUpdate, WeatherUpdateInterval;
bool  FilterApply, LaserDistanceError, FreshQVCdata, Level_Mode;
int   NumberOfLevelTransDataPoints;
float*  pLevelTrans_Xv;       // Spline function discrete points for the Level Sensor
double* pLevelTrans_Coeff;      // Spline function Coefficients for the Level Sensor
int   NumberOfLevelBeamDataPoints;
float*  pLevelBeam_Xv;        // Spline function discrete points for the Level Sensor
double* pLevelBeam_Coeff;     // Spline function Coefficients for the Level Sensor
bool  bTriggerProcedure;
CARTESIAN_DATA* pTemporaryPointer;
CARTESIAN_DATA temp_ResetPosition;
FIFO_RECORD* pRetrievalPointer;
float*  pAllocatedMemory;
D_VECTOR4* pReading_Buffer;
D_VECTOR4  Averaging_Buffer;
D_VECTOR2  InPositionDelta;
unsigned int Point_Counter;
unsigned char FacingStepNumber;
float VerifyPoints[3*4*5];
float Repeatabilities[4*2];
int   DataSetNumber;
bool  ErrorFlag, TriggerFired, TogglingSwitch;
unsigned char FacingSequencer;
float FacingAngles[2];
D_VECTOR4 FirstFacePoint, SecondFacePoint, BackSightPoint;
float Laser_Reset_Distance;
bool  GoBackToBirdBath, Turning_Direction, Blocked_Beam_Path;
double  In_Position_Check;

D_VECTOR4 Sample_Points[4];
unsigned int initial_counter;
double  MinDistance;
bool  FacingStatus;
D_VECTOR2 saveSightAngles;
D_VECTOR4 SightPointsCopy;
float Averaging_Counter;
HANDLE  hAPI_EventHandle, hWorkerThread;
DWORD dwWorkerId, dwExitCode;
static  DWORD RealTime_Monitoring(CTracker*); // Realtime Data Reading Thread
void  SetupDataStructure(float*, void*, float*);
double  exponential_filter(int, bool, double, int);
float calibrate_encoder(double, int, float*);
bool  FacingOperation(unsigned char*, float*, unsigned char, float*, double*, bool);
void  CalibrationProcedure(float*, double*, double*, float*);
int   mid_value(float, float, float);
bool  GoToBirdBath(unsigned char*);
bool  TwoFaceOperation(int, unsigned char*, unsigned char*, unsigned char*, unsigned
   char*, float*, int*, bool*, bool*, unsigned char*);
unsigned char m_iRTCapture;
bool  TrackerParametersSetup(char*, int*);
void  TrackerPrmFileCopy(void);
void  LogFileOperation(int, float*, float*, float*);
void  CubicSplines(float*, float*, int, double*);
float CalcSpline(bool, float*, double*, int, float, bool*);
```

```
  void  tridiag(double*, int, double*);
  double  PolyCalc(double, double*, int);
  void  ConvertCartesianToSphericalCoordinates(CARTESIAN_DATA*);
protected:
  HANDLE  TaskHandle;
public:
  CTracker(void);
  ~CTracker(void);
//  Tracker API Identification Information
  void getTrackerApiObject(char*    /* receiver */);  // Get Tracker API ObjectName
  void getTrackerApiRevision(char*  /* receiver */);  // Get Tracker API RevisionNumber
  void getTrackerApiRelease(char*   /* receiver */);  // Get Tracker API ReleaseDate
  void getControlVersion(char*    /* receiver */);  // Get Controller Firmware Version String
  void getModelNumber(char*     /* receiver */);  // Get Model Number String
  void getSerialNumber(char*      /* receiver */);  // Get Sensor Serial Number String
  void getRemoteTriggerID(char*   /* receiver */);  // Get Remote Trigger Identifier String
  bool setRemoteTriggerID(int,    /* RemoteTriggerID in Decimal(2 digits) */
             int* = &ErrorMessageNumber  /* Error Message Number */);
  void getLicenseOwner(char*      /* receiver */);  // Get License Owner
  virtual char* errorMsg(int  /* Error Number */);  // Returns Error Message string pointer
  virtual bool TrackerInitialization  (int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerInitialization  (char*,          // Configuration File Path+Name
                   int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerInitialization  (LPCTSTR, /* ComPort */ // Controller initialization function
                   char*,          // Configuration File Path+Name
                   int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerInitialization  (LPCTSTR, /* ComPort */ // Controller initialization function
                   DWORD,   /* BaudRate */// Serial Communication Baudrate
                   char*,          // Configuration File Path+Name
                   int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerInitialization  (bool*,    /* true:Initializing Sequence-Done,
                           false:Initializing Sequence-On Initializing Operation */
                   bool,     /* true:Abort the Initializing Procedure,
                           false:On Initializing Operation */
                   int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerInitialization  (bool*,    /* true:Initializing Sequence-Done,
                           false:Initializing Sequence-On Initializing Operation */
                   bool,     /* true:Abort the Initializing Procedure,
                           false:On Initializing Operation */
                   char*,   // Configuration File Path+Name
                   int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerInitialization  (bool*,    /* true:Initializing Sequence-Done,
                           false:Initializing Sequence-On Initializing Operation */
                   bool,     /* true:Abort the Initializing Procedure,
                           false:On Initializing Operation */
                   LPCTSTR, /* ComPort */ // Controller initialization function
                   char*,   // Configuration File Path+Name
                   int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerInitialization  (bool*,    /* true:Initializing Sequence-Done,
                           false:Initializing Sequence-On Initializing Operation */
                   bool,     /* true:Abort the Initializing Procedure,
                           false:On Initializing Operation */
                   LPCTSTR, /* ComPort */ // Controller initialization function
                   DWORD,   /* BaudRate */// Serial Communication Baudrate
                   char*,   // Configuration File Path+Name
                   int* = &ErrorMessageNumber /* Error Message Number */);
  virtual void TrackerResetPosition (RESET_ANGULAR*);  /* Tracker Reseting Position in Angular Coordinates */
  virtual bool TrackerResetOperation  (bool*, /* true:Resetting Sequence done, false:On Resetting Operation */
                   bool,  /* true:Abort the Resetting Procedure, false:On Resetting Operation */
                   RESET_ANGULAR*,  /* Tracker Reseting Position in Angular Coordinates */
                   int* = &ErrorMessageNumber /* Error Message Number */);
  virtual void TrackerResetPosition (RESET_CARTESIAN*);  /* Tracker Reseting Position in XYZ Cartesian Coordinates */
  virtual bool TrackerResetOperation  (bool*, /* true:Resetting Sequence done, false:On Resetting Operation */
                   bool,  /* true:Abort the Resetting Procedure, false:On Resetting Operation */
```

```
                     RESET_CARTESIAN*,  /* Tracker Reseting Position in XYZ Cartesian Coordinates */
                     int* = &ErrorMessageNumber /* Error Message Number */);
virtual bool TrackerTargetSearch  (bool*, /* true:Searching Sequence done, false:On Searching Operation */
                     bool,  /* true:Abort the Searching Procedure, false:On Searching Operation */
                     float, /* target searching frequency */
                     float, /* target searching multiplier */
                     int* = &ErrorMessageNumber /* Error Message Number */);
virtual bool TrackerHomingOperation (bool*, /* true:Homing Sequence done, false:On Homing Operation */
                     bool,  /* true:Abort the Homing Procedure, false:On Homing Operation */
                     int* = &ErrorMessageNumber /* Error Message Number */);
// Return the pointer for the Realtime-based Data
virtual REALTIME_INFO* TrackerMonitoringData(int /* Exponential Filter Strength from 1: No filter,
                            the bigger, then the heavier filtration and the slower reponses*/);
virtual bool TrackerChangingMode  (int, /* Operation Mode Number */
                        //  0 -> Servo Free Mode (Idle)
                        //  1 -> Servo Engaged Mode (Servo)
                        //  2 -> Tracking Mode (Track)
                        //  3 -> Losing target during Tracking (Track Idle)
                        //  4 -> Not Used by API user (Internal Use Only)
                        //  5 -> Searching the Encoder Index (Index Searching)
                        //  6 -> Not Used by API user (Internal Use Only)
                        //  7 -> Target Scan Search Mode (Search)
                     int* = &ErrorMessageNumber /* Error Message Number */);
virtual bool TrackerJoggingMotion (TARGET*, /*  Jogging Mode and Target Point */
                     int* = &ErrorMessageNumber /* Error Message Number */);
virtual bool TrackerJoggingMotion (bool*,   /* true:Jogging Sequence done, false:On Jogging Operation */
                     bool,    /* true:Abort the Jogging Procedure, false:On Jogging Operation */
                     TARGET_EXTENDED*, /* Jogging Mode and Target Point */
                     int* = &ErrorMessageNumber /* Error Message Number */);
virtual bool TrackerCapturingOperation(int, /* CaptureMode  -3:Unacceptable
                               -2:Unacceptable
                               -1:Unacceptable
                                0:Unacceptable
                                1:Unacceptable
                                2:PC Realtime Static FrontSight Capturing
                                3:PC Realtime Static FrontBackSight Capturing
                                4:PC Realtime Temporal Capturing
                                5:Unacceptable */
                       FIFO_RECORD*,  /* Memory Pointer for the Data Retrieved */
                       unsigned long, /* Number of Points to be captured */
                       unsigned long, /* In case of Realtime Static Capturing : Averaging Number of Points
                              In case of Realtime Temporal Capturing : Capturing Interval(ms) */
                       int* = &ErrorMessageNumber /* Error Message Number */);
virtual bool TrackerCapturingOperation(int, /* CaptureMode  -3:PC Realtime Automatic FrontBackSight Capturing
                               -2:PC Realtime Automatic FrontSight Capturing
                               -1:Unacceptable
                                0:Unacceptable
                                1:Unacceptable
                                2:Unacceptable
                                3:Unacceptable
                                4:Unacceptable
                                5:Unacceptable */
                       FIFO_RECORD*,  /* Memory Pointer for the Data Retrieved */
                       unsigned long, /* Number of Points to be captured */
                       clock_t,     /* Waiting Delay in ms */
                       float,     /* Minimum Trigger Distance in mm */
                       float,     /* Velocity Band in mm/sec */
                       unsigned long, /* Averaging Number of Points */
                       int* = &ErrorMessageNumber /* Error Message Number */);
virtual bool TrackerCapturingOperation(int, /* CaptureMode  -3:Unacceptable
                               -2:Unacceptable
                               -1:Unacceptable
                                0:Unacceptable
                                1:Unacceptable
                                2:Unacceptable
                                3:Unacceptable
                                4:Unacceptable
```

```
                                 5:PC Realtime Spatial Capturing  */
                       FIFO_RECORD*,  /* Memory Pointer for the Data Retrieved */
                       unsigned long, /* Number of Points to be captured */
                       float,     /* In case of Realtime Spatial Capturing  : Capturing
    Distance(mm) */
                       int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerCapturingOperation(int, /* CaptureMode  -3:Unacceptable
                                -2:Unacceptable
                                -1:Unacceptable
                                 0:Controller Static Capturing
                                 1:Controller Temporal Capturing
                                 2:Unacceptable
                                 3:Unacceptable
                                 4:Unacceptable
                                 5:Unacceptable */
                       unsigned long, /* Number of Points to be captured */
                       unsigned long, /* In case of FIFO Static Capturing   : Averaging
    Duration(ms)[1-3000]
                               In case of FIFO Temporal Capturing   : Capturing Interval(
    ms)[1-3000] */
                       int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerCommandTrigger  (int* = &ErrorMessageNumber /* Error Message Number
     */);
  virtual bool TrackerFifoRetrieval (bool*, /* true:FIFO Retrieval Sequence done, false:
    On FIFO Retrieving Operation */
                     bool,  /* true:Abort the Retrieving Procedure, false:On Retrieving
    Operation */
                     FIFO_RECORD*,  /* Memory Pointer for the Data Retrieved */
                     int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerTaskingStop   (int* = &ErrorMessageNumber /* Error Message Number
    */);
  virtual void TrackerFIFOinformation(FIFO_INFO*, /* returned Fifo Information */
                    int* = &ErrorMessageNumber  /* Error Message Number */);
  virtual bool TrackerVerifyingOperation(bool*, /* true:Verifying Sequence-Done,
                           false:Verifying Sequence-On Verifying Operation */
                       bool,  /* true:Abort the Verifying Procedure, false:On Verifying
    Operation */
                       bool*, /* In Position flag for the AZ,EL
                           Point A : true if in-pos is within +/-0.5deg(EL 0.0)
                           Point B : true if in-pos is within +/-0.5deg(EL 0.0)
                           Point C : true if in-pos is within +/-5.0deg(EL +55.0)
                           Point D : true if in-pos is within +/-5.0deg(EL -55.0) */
                       bool*, /* Data PickUp flag  true(command):read the current point,
    false(return):operation done */
                       unsigned char*, /* Position Indicator
                           Point A : return value = 1
                           Point B : return value = 2
                           Point C : return value = 3
                           Point D : return value = 4 */
                       float*, /* Calibration Result - Single Dimensional 4th order float
     Array
                              The final result is returned when the Calibration Procedure
     has been done.
                            The ratio is between 0.0 and 1.0.
                            Array [0] : Squareness Calibration Result
                            Array [1] : T-axis offset Calibration Result
                            Array [2] : Z-axis offset Calibration Result
                            Array [3] : T-Beam Deviation Calibration Result
                            Array [4] : Z-Beam Deviation Calibration Result
                            Array [5] : T-V Distance Calibration Result */
                       int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerVerifyingOperation(int, /* QVC Method - 0 : Single Cycle QVC
    Operation, 1 : Three Cycle QVC Operation */
                       bool*, /* true:Verifying Sequence-Done,
                           false:Verifying Sequence-On Verifying Operation */
                       bool,  /* true:Abort the Verifying Procedure, false:On Verifying
    Operation */
                       bool*, /* In Position flag for the AZ,EL
                           Point A : true if in-pos is within +/-0.5deg(EL 0.0)
                           Point B : true if in-pos is within +/-0.5deg(EL 0.0)
                           Point C : true if in-pos is within +/-5.0deg(EL +55.0)
                           Point D : true if in-pos is within +/-5.0deg(EL -55.0) */
                       bool*, /* Data PickUp flag  true(command):read the current point,
    false(return):operation done */
                       unsigned char*, /* Position Indicator
                           Point A : return value = 1
                           Point B : return value = 2
                           Point C : return value = 3
```

```
                                    Point D : return value = 4 */
                                float*, /* Calibration Result - Single Dimensional 4th order float Array
                                        The final result is returned when the Calibration Procedure has been done.
                                     The ratio is between 0.0 and 1.0.
                                     Array [0] : Squareness Calibration Result
                                     Array [1] : T-axis offset Calibration Result
                                     Array [2] : Z-axis offset Calibration Result
                                     Array [3] : T-Beam Deviation Calibration Result
                                     Array [4] : Z-Beam Deviation Calibration Result
                                     Array [5] : T-V Distance Calibration Result */
                                int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerQvcUpdate(int* = &ErrorMessageNumber  /* Error Message Number */);
  virtual void TrackerDefaultRecovery(void);
  virtual bool TrackerCheckingOperation(bool*, /* true:Checking Sequence-Done,
                                  false:Checking Sequence-On Checking Operation */
                               bool,  /* true:Abort the Checking Procedure, false:On Checking Operation */
                               unsigned char, /* Averaging Time in sec for the Position Data (recommendation: 5 sec)*/
                               float*, /* Backsight Result - Single Dimensional 2nd order float Array
                                    The final result is returned when the Backsight Procedure has been done.
                                    Array [0] : Checking Results for Azimuth BackSight Difference
                                    Array [1] : Checking Results for Elevation BackSight Difference */
                               int* = &ErrorMessageNumber  /* Error Message Number */);
  virtual bool TrackerBacksightOperation(bool*, /* true:BackSighting Sequence-Done,
                                    false:BackSighting Sequence-On BackSighting Operation */
                                bool,  /* true:Abort the BackSighting Procedure, false:On BackSighting Operation */
                                int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerToolingBall   (bool*, /* true:Calculating Sequence-Done,
                                    false:Calculating Sequence-On Calculating Operation */
                              bool,  /* true:Abort the Calculating Procedure, false:On Calculating Operation */
                              float, /* SMR Diameter(mm) */
                              float, /* Tooling Ball Diameter(mm) */
                              float, /* Center Error Tolerance(mm) */
                              unsigned int, /* Number of Calculation (10 - 1500) */
                              float*, /* Calculation Progress(%) */
                              CARTESIAN_DATA*, /* Tooling Ball Center point */
                              float*, /* Average Error */
                              float*, /* Maximum Error */
                              float*, /* RMS Error */
                              int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerNestReading   (bool*, /* true:Readinging Sequence-Done,
                                    false:Reading Sequence-On Reading Operation */
                              bool,  /* true:Abort the Reading Procedure, false:On Reading Operation */
                              unsigned int, /* Number of Readings for the averaging operation (10 - 1500) */
                              float*, /* Reading Progress(%) */
                              CARTESIAN_DATA*, /* SMR Center point */
                              float*, /* Average Error */
                              float*, /* Maximum Error */
                              float*, /* RMS Error */
                              int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerSetEnvironment  (float, /* Manual Air Pressure in mm/Hg. If it is 0.0, then the Automatic Sensor value will be applied.(580.0mm/Hg - 800.0mm/Hg) */
                              float, /* Manual Air Temperature in centigrade. If it is 0.0, then the Automatic Sensor value will be applied.(5.0C - 45.0C) */
                              float, /* Relative Humidity in precentage. It should be set correctly whenever this function is called. (1%-100%) */
                              int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerSetEnvironment  (unsigned char, /* Updating time interval (1sec - 60sec). If it is 0sec, then the Updating Data is disabled. */
                              float, /* Manual Air Pressure in mm/Hg. If it is 0.0, then the Automatic Sensor value will be applied.(580.0mm/Hg - 800.0mm/Hg) */
                              float, /* Manual Air Temperature in centigrade. If it is 0.0, then the Automatic Sensor value will be applied.(5.0C - 45.0C) */
                              float, /* Relative Humidity in precentage. It should be set correctly whenever this function is called. (1%-100%) */
                              int* = &ErrorMessageNumber /* Error Message Number */);
```

```
  virtual bool TrackerGetEnvironment  (float*, /* Air Pressure in mm/Hg. If it is over
    the range (580.0mm/Hg - 800.0mm/Hg), then Error. */
                     float*, /* Air Temperature in centigrade. If it is over the range
    (5.0C - 45.0C), then Error. */
                     float*, /* Relative Humidity in precentage. If it is over the range
    (1%-100%), then Error.  */
                     int* = &ErrorMessageNumber /* Error Message Number */);
  virtual bool TrackerClosingOperation(int* = &ErrorMessageNumber /* Error Message Number
     */);
};
//
    ////////////////////////////////////////////////////////////////////////////////////////
#endif
```

A.1.2 Nomad Chassis Drivers

The code documented in this section belongs to the Nomad robot chassis driver. The chassis uses differential drive model with L298n chipset. IMU is provided by the openCR board. There is also a servo on the chassis which is used for rotating the LiDAR to move its FOV.

A.1.2.1 Anduino ino file for openCR

```
#include "turtlebot3_core_config.h"
/*******************************
* Setup function
*******************************/
void setup()
{
  DEBUG_SERIAL.begin(9600);
  // Initialize ROS node handle, advertise and subscribe the topics
  nh.initNode();
  nh.getHardware()->setBaud(115200);
  nh.subscribe(cmd_vel_sub);
  nh.subscribe(sound_sub);
  nh.subscribe(joint_position_sub);
  nh.subscribe(reset_sub);
  nh.advertise(sensor_state_pub);
  nh.advertise(version_info_pub);
  nh.advertise(imu_pub);
  nh.advertise(odom_pub);
  nh.advertise(joint_states_pub);
  nh.advertise(battery_state_pub);
  nh.advertise(mag_pub);
  tf_broadcaster.init(nh);
  // Setting for Dynamixel motors
//  motor_driver.init(NAME);/
  // Setting for IMU
  sensors.init();
  // Init diagnosis
  diagnosis.init();
  // Setting for ROBOTIS RC100 remote controller and cmd_vel
  controllers.init(MAX_LINEAR_VELOCITY, MAX_ANGULAR_VELOCITY);
  // Setting for SLAM and navigation (odometry, joint states, TF)
  initOdom();
  initJointStates();
  prev_update_time = millis();
  pinMode(LED_WORKING_CHECK, OUTPUT);
  setup_end = true;
/*******************************
* MOTORS
*******************************/
  pinMode(L_PWM, OUTPUT);
  pinMode(L_FORW, OUTPUT);
  pinMode(L_BACK, OUTPUT);
  pinMode(R_PWM, OUTPUT);
```

```
  pinMode(R_FORW, OUTPUT);
  pinMode(R_BACK, OUTPUT);
  stop();
/*******************************
* dex servo init
*******************************/

  bool result = false;
  const char *log;
  uint16_t model_number = 0;
  result = dxl_wb.init(DEVICE_NAME, BAUDRATE, &log);
  if (result == false)
  {
    Serial.println(log);
    Serial.println("Failed to init");
  }
  else
  {
    Serial.print("Succeeded to init : ");
    Serial.println(BAUDRATE);
  }
  result = dxl_wb.ping(DXL_ID, &model_number, &log);
  if (result == false)
  {
    Serial.println(log);
    Serial.println("Failed to ping");
  }
  else
  {
    Serial.println("Succeeded to ping");
    Serial.print("id : ");
    Serial.print(DXL_ID);
    Serial.print(" model_number : ");
    Serial.println(model_number);
  }
  result = dxl_wb.jointMode(DXL_ID, 0, 0, &log);
  if (result == false)
  {
    Serial.println(log);
    Serial.println("Failed to change joint mode");
  }
  else
  {
    Serial.println("Succeed to change joint mode");
    dxl_wb.led(DXL_ID, true, &log);
  }
/*******************************
* encoder
*******************************/
  initEncoders();
  resetEncoders();
}
/*******************************
* Loop function
*******************************/
void loop()
{
  uint32_t t = millis();
  updateTime();
  updateVariable(nh.connected());
  updateTFPrefix(nh.connected());

  if ((t-tTime[0]) >= (1000 / CONTROL_MOTOR_SPEED_FREQUENCY))
  {
    updateGoalVelocity();
    if ((t-tTime[5]) > CONTROL_MOTOR_TIMEOUT)
    {
//      l298_motor_driver();/
//      stop();/
    }
    else {
      l298_motor_driver();
    }
    tTime[0] = t;
  }
  if ((t-tTime[1]) >= (1000 / ENCODER_SAMPLE_FREQUENCY))
  {
```

```
    updateEncoder();
    tTime[1] = t;
  }
  if ((t-tTime[2]) >= (1000 / DRIVE_INFORMATION_PUBLISH_FREQUENCY))
  {
    publishSensorStateMsg();
    publishBatteryStateMsg();
    publishDriveInformation();
    tTime[2] = t;
  }
  if ((t-tTime[3]) >= (1000 / IMU_PUBLISH_FREQUENCY))
  {
    publishImuMsg();
    publishMagMsg();
    tTime[3] = t;
  }
//  if ((t-tTime[4]) >= (1000 / JOINT_CONTROL_FREQEUNCY))
//  {
//    jointControl();
//    tTime[4] = t;
//  }
#ifdef DEBUG
  if ((t-tTime[4]) >= (1000 / DEBUG_LOG_FREQUENCY))
  {
    sendDebuglog();
    tTime[4] = t;
  }
#endif
  // Send log message after ROS connection
  sendLogMsg();
  // Check push button pressed for simple test drive
  driveTest(diagnosis.getButtonPress(3000));
  // Update the IMU unit
  sensors.updateIMU();
  // TODO
  // Update sonar data
  // sensors.updateSonar(t);
  // Start Gyro Calibration after ROS connection
  updateGyroCali(nh.connected());
  // Show LED status
  diagnosis.showLedStatus(nh.connected());
  // Update Voltage
  battery_state = diagnosis.updateVoltageCheck(setup_end);
  // Call all the callbacks waiting to be called at that point in time
  nh.spinOnce();
  // Wait the serial link time to process
  waitForSerialLink(nh.connected());
}
/*******************************
* Callback function for cmd_vel msg
*******************************/
void commandVelocityCallback(const geometry_msgs::Twist& cmd_vel_msg)
{
  goal_velocity_from_cmd[LINEAR]  = cmd_vel_msg.linear.x;
  goal_velocity_from_cmd[ANGULAR] = cmd_vel_msg.angular.z;
  goal_velocity_from_cmd[LINEAR]  = constrain(goal_velocity_from_cmd[LINEAR],
    MIN_LINEAR_VELOCITY, MAX_LINEAR_VELOCITY);
  goal_velocity_from_cmd[ANGULAR] = constrain(goal_velocity_from_cmd[ANGULAR],
    MIN_ANGULAR_VELOCITY, MAX_ANGULAR_VELOCITY);
  tTime[6] = millis();
}
void jointTrajectoryPointCallback(const std_msgs::Int32& joint_pointing_msg)
{
//  if (is_moving == false)
//  {
    const char *log;
    joint_trajectory_point = joint_pointing_msg.data;
    dxl_wb.goalPosition(DXL_ID, (int32_t)joint_trajectory_point, &log);
//    is_moving = true;
//  }
}
void l298_motor_driver()
{
  float x = max(min(goal_velocity_from_cmd[LINEAR], 1.0f), -1.0f);
  float z = max(min(goal_velocity_from_cmd[ANGULAR], 1.0f), -1.0f);
  // Calculate the intensity of left and right wheels. Simple version.
```

```
  // Taken from https://hackernoon.com/unicycle-to-differential-drive-courseras-control-
    of-mobile-robots-with-ros-and-rosbots-part-2-6d27d15f2010#1e59
  float l = (goal_velocity_from_cmd[LINEAR] - goal_velocity_from_cmd[ANGULAR]) / 2;
  float r = (goal_velocity_from_cmd[LINEAR] + goal_velocity_from_cmd[ANGULAR]) / 2;
  // Then map those values to PWM intensities. PWMRANGE = full speed, while PWM_MIN = the
     minimal amount of power at which the motors begin moving.
  uint16_t lPwm = mapPwm(fabs(l), PWM_MIN, PWMRANGE);
  uint16_t rPwm = mapPwm(fabs(r), PWM_MIN, PWMRANGE);
  // Set direction pins and PWM
  digitalWrite(L_FORW, l > 0);
  digitalWrite(L_BACK, l < 0);
  digitalWrite(R_FORW, r > 0);
  digitalWrite(R_BACK, r < 0);
  analogWrite(L_PWM, lPwm);
  analogWrite(R_PWM, rPwm);
}
float mapPwm(float x, float out_min, float out_max)
{
  return x * (out_max - out_min) + out_min;
}
/*********************************
* Callback function for sound msg
*********************************/
void soundCallback(const turtlebot3_msgs::Sound& sound_msg)
{
  sensors.makeSound(sound_msg.value);
}
/*********************************
* Callback function for motor_power msg
*********************************/
//void motorPowerCallback(const std_msgs::Bool& power_msg)
//{
//  bool dxl_power = power_msg.data;
//
//  motor_driver.setTorque(dxl_power);
//}
/*********************************
* Callback function for reset msg
*********************************/
void resetCallback(const std_msgs::Empty& reset_msg)
{
  char log_msg[50];
  (void)(reset_msg);
  sprintf(log_msg, "Start Calibration of Gyro");
  nh.loginfo(log_msg);
  sensors.calibrationGyro();
  sprintf(log_msg, "Calibration End");
  nh.loginfo(log_msg);
  initOdom();
  sprintf(log_msg, "Reset Odometry");
  nh.loginfo(log_msg);
}
/*********************************
* Publish msgs (IMU data: angular velocity, linear acceleration, orientation)
*********************************/
void publishImuMsg(void)
{
  imu_msg = sensors.getIMU();
  imu_msg.header.stamp    = rosNow();
  imu_msg.header.frame_id = imu_frame_id;
  imu_pub.publish(&imu_msg);
}
/*********************************
* Publish msgs (Magnetic data)
*********************************/
void publishMagMsg(void)
{
  mag_msg = sensors.getMag();
  mag_msg.header.stamp    = rosNow();
  mag_msg.header.frame_id = mag_frame_id;
  mag_pub.publish(&mag_msg);
}
/*********************************
* Publish msgs
*********************************/
void publishSensorStateMsg(void)
{
```

```
  bool dxl_comm_result = false;
  sensor_state_msg.header.stamp = rosNow();
  sensor_state_msg.battery = sensors.checkVoltage();
  dxl_comm_result = readEncoder(sensor_state_msg.left_encoder, sensor_state_msg.
    right_encoder);
  if (dxl_comm_result == true)
    updateMotorInfo(sensor_state_msg.left_encoder, sensor_state_msg.right_encoder);
  else
    return;
  sensor_state_msg.bumper = sensors.checkPushBumper();
  sensor_state_msg.cliff = sensors.getIRsensorData();
  // TODO
  // sensor_state_msg.sonar = sensors.getSonarData();
  sensor_state_msg.illumination = sensors.getIlluminationData();
  sensor_state_msg.button = sensors.checkPushButton();
//  sensor_state_msg.torque = mot/or_driver.getTorque();
  sensor_state_pub.publish(&sensor_state_msg);
}
/*******************************
* Publish msgs (version info)
*******************************/
void publishVersionInfoMsg(void)
{
  version_info_msg.hardware = "0.0.0";
  version_info_msg.software = "0.0.0";
  version_info_msg.firmware = FIRMWARE_VER;
  version_info_pub.publish(&version_info_msg);
}
/*******************************
* Publish msgs (battery_state)
*******************************/
void publishBatteryStateMsg(void)
{
  battery_state_msg.header.stamp = rosNow();
  battery_state_msg.design_capacity = 1.8f; //Ah
  battery_state_msg.voltage = sensors.checkVoltage();
  battery_state_msg.percentage = (float)(battery_state_msg.voltage / 11.1f);
  if (battery_state == 0)
    battery_state_msg.present = false;
  else
    battery_state_msg.present = true;
  battery_state_pub.publish(&battery_state_msg);
}
/*******************************
* Publish msgs (odometry, joint states, tf)
*******************************/
void publishDriveInformation(void)
{
  unsigned long time_now = millis();
  unsigned long step_time = time_now - prev_update_time;
  prev_update_time = time_now;
  ros::Time stamp_now = rosNow();
  // calculate odometry
  calcOdometry((double)(step_time * 0.001));
  // odometry
  updateOdometry();
  odom.header.stamp = stamp_now;
  odom_pub.publish(&odom);
  // odometry tf
  updateTF(odom_tf);
  odom_tf.header.stamp = stamp_now;
  tf_broadcaster.sendTransform(odom_tf);
  // joint states
  updateJointStates();
  joint_states.header.stamp = stamp_now;
  joint_states_pub.publish(&joint_states);
}
/*******************************
* Update TF Prefix
*******************************/
void updateTFPrefix(bool isConnected)
{
  static bool isChecked = false;
  char log_msg[50];
  if (isConnected)
  {
    if (isChecked == false)
    {
```

```
      nh.getParam("~tf_prefix", &get_tf_prefix);
      if (!strcmp(get_tf_prefix, ""))
      {
        sprintf(odom_header_frame_id, "odom");
        sprintf(odom_child_frame_id, "base_footprint");
        sprintf(imu_frame_id, "imu_link");
        sprintf(mag_frame_id, "mag_link");
        sprintf(joint_state_header_frame_id, "base_link");
      }
      else
      {
        strcpy(odom_header_frame_id, get_tf_prefix);
        strcpy(odom_child_frame_id, get_tf_prefix);
        strcpy(imu_frame_id, get_tf_prefix);
        strcpy(mag_frame_id, get_tf_prefix);
        strcpy(joint_state_header_frame_id, get_tf_prefix);
        strcat(odom_header_frame_id, "/odom");
        strcat(odom_child_frame_id, "/base_footprint");
        strcat(imu_frame_id, "/imu_link");
        strcat(mag_frame_id, "/mag_link");
        strcat(joint_state_header_frame_id, "/base_link");
      }
      sprintf(log_msg, "Setup TF on Odometry [%s]", odom_header_frame_id);
      nh.loginfo(log_msg);
      sprintf(log_msg, "Setup TF on IMU [%s]", imu_frame_id);
      nh.loginfo(log_msg);
      sprintf(log_msg, "Setup TF on MagneticField [%s]", mag_frame_id);
      nh.loginfo(log_msg);
      sprintf(log_msg, "Setup TF on JointState [%s]", joint_state_header_frame_id);
      nh.loginfo(log_msg);
      isChecked = true;
    }
  }
  else
  {
    isChecked = false;
  }
}
/*******************************
* Update the odometry
*******************************/
void updateOdometry(void)
{
  odom.header.frame_id = odom_header_frame_id;
  odom.child_frame_id  = odom_child_frame_id;
  odom.pose.pose.position.x = odom_pose[0];
  odom.pose.pose.position.y = odom_pose[1];
  odom.pose.pose.position.z = 0;
  odom.pose.pose.orientation = tf::createQuaternionFromYaw(odom_pose[2]);
  odom.twist.twist.linear.x  = odom_vel[0];
  odom.twist.twist.angular.z = odom_vel[2];
}
/*******************************
* Update the wheel states
*******************************/
void updateJointStates(void)
{
  static float joint_states_pos[WHEEL_NUM] = {0.0, 0.0};
  static float joint_states_vel[WHEEL_NUM] = {0.0, 0.0};
  //static float joint_states_eff[WHEEL_NUM] = {0.0, 0.0};
  joint_states_pos[LEFT]  = last_rad[LEFT];
  joint_states_pos[RIGHT] = last_rad[RIGHT];
  joint_states_vel[LEFT]  = last_velocity[LEFT];
  joint_states_vel[RIGHT] = last_velocity[RIGHT];
  joint_states.position = joint_states_pos;
  joint_states.velocity = joint_states_vel;
}
/*******************************
* CalcUpdateulate the TF
*******************************/
void updateTF(geometry_msgs::TransformStamped& odom_tf)
{
  odom_tf.header = odom.header;
  odom_tf.child_frame_id = odom.child_frame_id;
  odom_tf.transform.translation.x = odom.pose.pose.position.x;
  odom_tf.transform.translation.y = odom.pose.pose.position.y;
  odom_tf.transform.translation.z = odom.pose.pose.position.z;
  odom_tf.transform.rotation      = odom.pose.pose.orientation;
```

```
}
/*********************************
* Update motor information
*********************************/
void updateMotorInfo(int32_t left_tick, int32_t right_tick)
{
  int32_t current_tick = 0;
  static int32_t last_tick[WHEEL_NUM] = {0, 0};
  if (init_encoder)
  {
    for (int index = 0; index < WHEEL_NUM; index++)
    {
      last_diff_tick[index] = 0;
      last_tick[index]      = 0;
      last_rad[index]       = 0.0;
      last_velocity[index]  = 0.0;
    }
    last_tick[LEFT] = left_tick;
    last_tick[RIGHT] = right_tick;
    init_encoder = false;
    return;
  }
  current_tick = left_tick;
  last_diff_tick[LEFT] = current_tick - last_tick[LEFT];
  last_tick[LEFT]      = current_tick;
  last_rad[LEFT]       += TICK2RAD * (double)last_diff_tick[LEFT];
  current_tick = right_tick;
  last_diff_tick[RIGHT] = current_tick - last_tick[RIGHT];
  last_tick[RIGHT]      = current_tick;
  last_rad[RIGHT]       += TICK2RAD * (double)last_diff_tick[RIGHT];
}
/*********************************
* Calculate the odometry
*********************************/
bool calcOdometry(double diff_time)
{
  float* orientation;
  double wheel_l, wheel_r;      // rotation value of wheel [rad]
  double delta_s, theta, delta_theta;
  static double last_theta = 0.0;
  double v, w;                  // v = translational velocity [m/s], w = rotational
    velocity [rad/s]
  double step_time;
  wheel_l = wheel_r = 0.0;
  delta_s = delta_theta = theta = 0.0;
  v = w = 0.0;
  step_time = 0.0;
  step_time = diff_time;
  if (step_time == 0)
    return false;
  wheel_l = TICK2RAD * (double)last_diff_tick[LEFT];
  wheel_r = TICK2RAD * (double)last_diff_tick[RIGHT];
  if (isnan(wheel_l))
    wheel_l = 0.0;
  if (isnan(wheel_r))
    wheel_r = 0.0;
  delta_s     = WHEEL_RADIUS * (wheel_r + wheel_l) / 2.0;
  // theta = WHEEL_RADIUS * (wheel_r - wheel_l) / WHEEL_SEPARATION;
  orientation = sensors.getOrientation();
  theta       = atan2f(orientation[1]*orientation[2] + orientation[0]*orientation[3],
                0.5f - orientation[2]*orientation[2] - orientation[3]*orientation[3]);
  delta_theta = theta - last_theta;
  // compute odometric pose
  odom_pose[0] += delta_s * cos(odom_pose[2] + (delta_theta / 2.0));
  odom_pose[1] += delta_s * sin(odom_pose[2] + (delta_theta / 2.0));
  odom_pose[2] += delta_theta;
  // compute odometric instantaneous velocity
  v = delta_s / step_time;
  w = delta_theta / step_time;
  odom_vel[0] = v;
  odom_vel[1] = 0.0;
  odom_vel[2] = w;
  last_velocity[LEFT]  = wheel_l / step_time;
  last_velocity[RIGHT] = wheel_r / step_time;
  last_theta = theta;
  return true;
}
/*********************************
* Turtlebot3 test drive using push buttons
```

```
********************************/
void driveTest(uint8_t buttons)
{
  static bool move[2] = {false, false};
  static int32_t saved_tick[2] = {0, 0};
  static double diff_encoder = 0.0;
  int32_t current_tick[2] = {0, 0};
  readEncoder(current_tick[LEFT], current_tick[RIGHT]);
  if (buttons & (1<<0))
  {
    move[LINEAR] = true;
    saved_tick[RIGHT] = current_tick[RIGHT];
    diff_encoder = TEST_DISTANCE / (0.207 / 4096); // (Circumference of Wheel) / (The
    number of tick per revolution)
    tTime[6] = millis();
  }
  else if (buttons & (1<<1))
  {
    move[ANGULAR] = true;
    saved_tick[RIGHT] = current_tick[RIGHT];
    diff_encoder = (TEST_RADIAN * TURNING_RADIUS) / (0.207 / 4096);
    tTime[6] = millis();
  }
  if (move[LINEAR])
  {
    if (abs(saved_tick[RIGHT] - current_tick[RIGHT]) <= diff_encoder)
    {
      goal_velocity_from_button[LINEAR]  = 0.05;
      tTime[6] = millis();
    }
    else
    {
      goal_velocity_from_button[LINEAR]  = 0.0;
      move[LINEAR] = false;
    }
  }
  else if (move[ANGULAR])
  {
    if (abs(saved_tick[RIGHT] - current_tick[RIGHT]) <= diff_encoder)
    {
      goal_velocity_from_button[ANGULAR]= -0.7;
      tTime[6] = millis();
    }
    else
    {
      goal_velocity_from_button[ANGULAR]  = 0.0;
      move[ANGULAR] = false;
    }
  }
}
/********************************
* Update variable (initialization)
********************************/
void updateVariable(bool isConnected)
{
  static bool variable_flag = false;
  if (isConnected)
  {
    if (variable_flag == false)
    {
      sensors.initIMU();
      initOdom();
      variable_flag = true;
    }
  }
  else
  {
    variable_flag = false;
  }
}
/********************************
* Wait for Serial Link
********************************/
void waitForSerialLink(bool isConnected)
{
  static bool wait_flag = false;
  if (isConnected)
```

```
  {
    if (wait_flag == false)
    {
      delay(10);
      wait_flag = true;
    }
  }
  else
  {
    wait_flag = false;
  }
}
/*******************************
* Update the base time for interpolation
*******************************/
void updateTime()
{
  current_offset = millis();
  current_time = nh.now();
}
/*******************************
* ros::Time::now() implementation
*******************************/
ros::Time rosNow()
{
  return nh.now();
}
/*******************************
* Time Interpolation function (deprecated)
*******************************/
ros::Time addMicros(ros::Time & t, uint32_t _micros)
{
  uint32_t sec, nsec;
  sec  = _micros / 1000 + t.sec;
  nsec = _micros % 1000000000 + t.nsec;
  return ros::Time(sec, nsec);
}
/*******************************
* Start Gyro Calibration
*******************************/
void updateGyroCali(bool isConnected)
{
  static bool isEnded = false;
  char log_msg[50];
  (void)(isConnected);
  if (nh.connected())
  {
    if (isEnded == false)
    {
      sprintf(log_msg, "Start Calibration of Gyro");
      nh.loginfo(log_msg);
      sensors.calibrationGyro();
      sprintf(log_msg, "Calibration End");
      nh.loginfo(log_msg);
      isEnded = true;
    }
  }
  else
  {
    isEnded = false;
  }
}
/*******************************
* Send log message
*******************************/
void sendLogMsg(void)
{
  static bool log_flag = false;
  char log_msg[100];
  String name             = NAME;
  String firmware_version = FIRMWARE_VER;
  String bringup_log      = "This core(v" + firmware_version + ") is compatible with TB3
    " + name;
  const char* init_log_data = bringup_log.c_str();
  if (nh.connected())
  {
    if (log_flag == false)
    {
```

```
      sprintf(log_msg, "--------------------------");
      nh.loginfo(log_msg);
      sprintf(log_msg, "Connected to OpenCR board!");
      nh.loginfo(log_msg);
      sprintf(log_msg, init_log_data);
      nh.loginfo(log_msg);
      sprintf(log_msg, "--------------------------");
      nh.loginfo(log_msg);
      log_flag = true;
    }
  }
  else
  {
    log_flag = false;
  }
}
/*******************************
* Initialization odometry data
*******************************/
void initOdom(void)
{
  init_encoder = true;
  for (int index = 0; index < 3; index++)
  {
    odom_pose[index] = 0.0;
    odom_vel[index]  = 0.0;
  }
  odom.pose.pose.position.x = 0.0;
  odom.pose.pose.position.y = 0.0;
  odom.pose.pose.position.z = 0.0;
  odom.pose.pose.orientation.x = 0.0;
  odom.pose.pose.orientation.y = 0.0;
  odom.pose.pose.orientation.z = 0.0;
  odom.pose.pose.orientation.w = 0.0;
  odom.twist.twist.linear.x  = 0.0;
  odom.twist.twist.angular.z = 0.0;
}
/*******************************
* Initialization joint states data
*******************************/
void initJointStates(void)
{
  static char *joint_states_name[] = {(char*)"wheel_left_joint", (char*)"
    wheel_right_joint"};
  joint_states.header.frame_id = joint_state_header_frame_id;
  joint_states.name            = joint_states_name;
  joint_states.name_length     = WHEEL_NUM;
  joint_states.position_length = WHEEL_NUM;
  joint_states.velocity_length = WHEEL_NUM;
  joint_states.effort_length   = WHEEL_NUM;
}
/*******************************
* Update Goal Velocity
*******************************/
void updateGoalVelocity(void)
{
  goal_velocity[LINEAR]  = goal_velocity_from_button[LINEAR]  + goal_velocity_from_cmd[
    LINEAR];
  goal_velocity[ANGULAR] = goal_velocity_from_button[ANGULAR] + goal_velocity_from_cmd[
    ANGULAR];
  sensors.setLedPattern(goal_velocity[LINEAR], goal_velocity[ANGULAR]);
}

/*******************************
* Encoders function
*******************************/
void initEncoders(){
  pinMode(LencoderPinA, INPUT_PULLUP);
  pinMode(LencoderPinB, INPUT_PULLUP);
  pinMode(RencoderPinA, INPUT_PULLUP);
  pinMode(RencoderPinB, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(LencoderPinA), encoderLeftISR,  CHANGE);
//  attachInterrupt(digitalPinToInterrupt(3), encoderLeftISR,  CHANGE);
  attachInterrupt(digitalPinToInterrupt(RencoderPinA), encoderRightISR, CHANGE);
//  attachInterrupt(digitalPinToInterrupt(7), encoderRightISR, CHANGE);
}

void encoderLeftISR(){
  if (digitalRead (LencoderPinA))
```

```
    Lup = digitalRead (LencoderPinB);
  else
    Lup = !digitalRead (LencoderPinB);
  Lfired = true;
}
void encoderRightISR(){
  if (digitalRead (RencoderPinA))
    Rup = !digitalRead (RencoderPinB);
  else
    Rup = digitalRead (RencoderPinB);
  Rfired = true;
}

/* Wrap the encoder reset function */
void resetEncoder(int i) {
  if (i == LEFT){
    noInterrupts();
    encoderLeft = 0L;
    interrupts();
  }else {
    noInterrupts();
    encoderRight = 0L;
     interrupts();
  }
}
/* Wrap the encoder reset function */
void resetEncoders() {
  resetEncoder(LEFT);
  resetEncoder(RIGHT);
}
void updateEncoder(){
  char log_msg[50];
  if (Lfired)
  {
    if (Lup)
      encoderLeft++;
    else
      encoderLeft--;
    Lfired = false;
//    DEBUG_SERIAL.print("left: ");
//    DEBUG_SERIAL.print(encoderLeft);
//    DEBUG_SERIAL.print("; right: ");
//    DEBUG_SERIAL.println(encoderRight);
   }  // end if fired
  if (Rfired)
  {
    if (Rup)
      encoderRight++;
    else
      encoderRight--;
    Rfired = false;
//    DEBUG_SERIAL.print("left: ");
//    DEBUG_SERIAL.print(encoderLeft);
//    DEBUG_SERIAL.print("; right: ");
//    DEBUG_SERIAL.println(encoderRight);
  }  // end if fired
  //sprintf(log_msg, "Left Encoder [%d], Right Encoder [%d]", encoderLeft, encoderRight);
  nh.loginfo(log_msg);
}
bool readEncoder(int32_t &left_value, int32_t &right_value){
  left_value = encoderLeft;
  right_value = encoderRight;
  return true;
}
/********************************
* LiDAR dex servo rotation  -----------tobe done
********************************/
void jointControl(void)
{
  dxl_wb.goalPosition(DXL_ID, (int32_t)2048);
}
/********************************
* Send Debug data
********************************/
void sendDebuglog(void)
{
  DEBUG_SERIAL.println("---------------------------------------");
  DEBUG_SERIAL.println("EXTERNAL SENSORS");
```

```
  DEBUG_SERIAL.println("---------------------------------------");
  DEBUG_SERIAL.print("Bumper : "); DEBUG_SERIAL.println(sensors.checkPushBumper());
  DEBUG_SERIAL.print("Cliff : "); DEBUG_SERIAL.println(sensors.getIRsensorData());
  DEBUG_SERIAL.print("Sonar : "); DEBUG_SERIAL.println(sensors.getSonarData());
  DEBUG_SERIAL.print("Illumination : "); DEBUG_SERIAL.println(sensors.getIlluminationData
    ());
  DEBUG_SERIAL.println("---------------------------------------");
  DEBUG_SERIAL.println("OpenCR SENSORS");
  DEBUG_SERIAL.println("---------------------------------------");
  DEBUG_SERIAL.print("Battery : "); DEBUG_SERIAL.println(sensors.checkVoltage());
  DEBUG_SERIAL.println("Button : " + String(sensors.checkPushButton()));
  float* quat = sensors.getOrientation();
  DEBUG_SERIAL.println("IMU : ");
  DEBUG_SERIAL.print("    w : "); DEBUG_SERIAL.println(quat[0]);
  DEBUG_SERIAL.print("    x : "); DEBUG_SERIAL.println(quat[1]);
  DEBUG_SERIAL.print("    y : "); DEBUG_SERIAL.println(quat[2]);
  DEBUG_SERIAL.print("    z : "); DEBUG_SERIAL.println(quat[3]);
  DEBUG_SERIAL.println("---------------------------------------");
  DEBUG_SERIAL.println("DYNAMIXELS");
  DEBUG_SERIAL.println("---------------------------------------");
//  DEBUG_SERIAL.println("Torque : " /+ String(motor_driver.getTorque()));
  int32_t encoder[WHEEL_NUM] = {0, 0};
  readEncoder(encoder[LEFT], encoder[RIGHT]);
  DEBUG_SERIAL.println("Encoder(left) : " + String(encoder[LEFT]));
  DEBUG_SERIAL.println("Encoder(right) : " + String(encoder[RIGHT]));
  DEBUG_SERIAL.println("---------------------------------------");
  DEBUG_SERIAL.println("TurtleBot3");
  DEBUG_SERIAL.println("---------------------------------------");
  DEBUG_SERIAL.println("Odometry : ");
  DEBUG_SERIAL.print("         x : "); DEBUG_SERIAL.println(odom_pose[0]);
  DEBUG_SERIAL.print("         y : "); DEBUG_SERIAL.println(odom_pose[1]);
  DEBUG_SERIAL.print("     theta : "); DEBUG_SERIAL.println(odom_pose[2]);
}
/********************************
Motor controll
********************************/
void stop()
{
  digitalWrite(L_FORW, 0);
  digitalWrite(L_BACK, 0);
  digitalWrite(R_FORW, 0);
  digitalWrite(R_BACK, 0);
  analogWrite(L_PWM, 0);
  analogWrite(R_PWM, 0);
}
```

A.1.2.2 Servo controller

```
 #include <DynamixelWorkbench.h>

/*******************************************************************************
* encoder
*******************************************************************************/
#define WHEEL_NUM                        2
#define LEFT                             0
#define RIGHT                            1
#define LINEAR                           0
#define ANGULAR                          1
#define DEG2RAD(x)                       (x * 0.01745329252)  // *PI/180
#define RAD2DEG(x)                       (x * 57.2957795131)  // *180/PI
#define TICK2RAD                         0.001533981  // 0.087890625[deg] * 3.14159265359
     / 180 = 0.001533981f
#define TEST_DISTANCE                    0.300     // meter
#define TEST_RADIAN                      3.14      // 180 degree
/*******************************************************************************
* servo
*******************************************************************************/
#define DEVICE_NAME ""
#define BAUDRATE  1000000

#define LEFT            0
#define RIGHT           1
#define FORWARDS true
#define BACKWARDS false
#define PWM_MIN 110
#define PWMRANGE 255
/*******************************************************************************
```

```
* Joint servo
*****************************************************************************/
void jointTrajectoryPointCallback(const std_msgs::Int32& joint_trajectory_point_msg);
ros::Subscriber<std_msgs::Int32> joint_position_sub("livox_joint",
    jointTrajectoryPointCallback);
DynamixelWorkbench dxl_wb;
bool is_moving        = false;
int joint_trajectory_point;
uint8_t DXL_ID = 1;
/*****************************************************************************
* encoder
*****************************************************************************/
// add in the next 3 lines to fix min max bug
#undef min
inline int min(int a, int b) { return ((a)<(b) ? (a) : (b)); }
inline double min(double a, double b) { return ((a)<(b) ? (a) : (b)); }
#undef max
inline int max(int a, int b) { return ((a)>(b) ? (a) : (b)); }
inline double max(double a, double b) { return ((a)>(b) ? (a) : (b)); }
uint16_t lPwm;
uint16_t rPwm;
float l;
float r;
const uint8_t L_PWM = 9;
const uint8_t L_BACK = 4;
const uint8_t L_FORW = 5;
const uint8_t R_BACK = 6;
const uint8_t R_FORW = 7;
const uint8_t R_PWM = 10;
volatile long encoderLeft = 0L;
volatile long encoderRight = 0L;
const byte LencoderPinA = 2;
const byte LencoderPinB = 14;
const byte RencoderPinA = 3;
const byte RencoderPinB = 15;
volatile bool Lfired;
volatile bool Lup;
volatile bool Rfired;
volatile bool Rup;

/***************Dynamix Servo API memo***************
bool init(const char* device_name = "/dev/ttyUSB0",
        uint32_t baud_rate = 57600,
        const char **log = NULL);
bool begin(const char* device_name = "/dev/ttyUSB0",
        uint32_t baud_rate = 57600,
        const char **log = NULL);
bool setPortHandler(const char *device_name, const char **log = NULL);
bool setBaudrate(uint32_t baud_rate, const char **log = NULL);
bool setPacketHandler(float protocol_version, const char **log = NULL);
float getProtocolVersion(void);
uint32_t getBaudrate(void);
const char * getModelName(uint8_t id, const char **log = NULL);
uint16_t getModelNumber(uint8_t id, const char **log = NULL);
const ControlItem *getControlTable(uint8_t id, const char **log = NULL);
const ControlItem *getItemInfo(uint8_t id, const char *item_name, const char **log = NULL
    );
uint8_t getTheNumberOfControlItem(uint8_t id, const char **log = NULL);
const ModelInfo* getModelInfo(uint8_t id, const char **log = NULL);
uint8_t getTheNumberOfSyncWriteHandler(void);
uint8_t getTheNumberOfSyncReadHandler(void);
uint8_t getTheNumberOfBulkReadParam(void);
bool scan(uint8_t *get_id,
        uint8_t *get_the_number_of_id,
        uint8_t range = 253,
        const char **log = NULL);
bool scan(uint8_t *get_id,
        uint8_t *get_the_number_of_id,
        uint8_t start_number,
        uint8_t end_number,
        const char **log = NULL);
bool ping(uint8_t id,
        uint16_t *get_model_number,
        const char **log = NULL);
bool ping(uint8_t id,
        const char **log = NULL);
bool clearMultiTurn(uint8_t id, const char **log = NULL);
bool reboot(uint8_t id, const char **log = NULL);
bool reset(uint8_t id, const char **log = NULL);
bool writeRegister(uint8_t id, uint16_t address, uint16_t length, uint8_t* data, const
    char **log = NULL);
```

```
bool writeRegister(uint8_t id, const char *item_name, int32_t data, const char **log =
    NULL);
bool writeOnlyRegister(uint8_t id, uint16_t address, uint16_t length, uint8_t *data,
    const char **log = NULL);
bool writeOnlyRegister(uint8_t id, const char *item_name, int32_t data, const char **log
    = NULL);
bool readRegister(uint8_t id, uint16_t address, uint16_t length, uint32_t *data, const
    char **log = NULL);
bool readRegister(uint8_t id, const char *item_name, int32_t *data, const char **log =
    NULL);
void getParam(int32_t data, uint8_t *param);
bool addSyncWriteHandler(uint16_t address, uint16_t length, const char **log = NULL);
bool addSyncWriteHandler(uint8_t id, const char *item_name, const char **log = NULL);
bool syncWrite(uint8_t index, int32_t *data, const char **log = NULL);
bool syncWrite(uint8_t index, uint8_t *id, uint8_t id_num, int32_t *data, uint8_t
    data_num_for_each_id, const char **log = NULL);
bool addSyncReadHandler(uint16_t address, uint16_t length, const char **log = NULL);
bool addSyncReadHandler(uint8_t id, const char *item_name, const char **log = NULL);
bool syncRead(uint8_t index, const char **log = NULL);
bool syncRead(uint8_t index, uint8_t *id, uint8_t id_num, const char **log = NULL);
bool getSyncReadData(uint8_t index, int32_t *data, const char **log = NULL);
bool getSyncReadData(uint8_t index, uint8_t *id, uint8_t id_num, int32_t *data, const
    char **log = NULL);
bool getSyncReadData(uint8_t index, uint8_t *id, uint8_t id_num, uint16_t address,
    uint16_t length, int32_t *data, const char **log = NULL);
bool initBulkWrite(const char **log = NULL);
bool addBulkWriteParam(uint8_t id, uint16_t address, uint16_t length, int32_t data, const
     char **log = NULL);
bool addBulkWriteParam(uint8_t id, const char *item_name, int32_t data, const char **log
    = NULL);
bool bulkWrite(const char **log = NULL);
bool initBulkRead(const char **log = NULL);
bool addBulkReadParam(uint8_t id, uint16_t address, uint16_t length, const char **log =
    NULL);
bool addBulkReadParam(uint8_t id, const char *item_name, const char **log = NULL);
bool bulkRead(const char **log = NULL);
bool getBulkReadData(int32_t *data, const char **log = NULL);
bool getBulkReadData(uint8_t *id, uint8_t id_num, uint16_t *address, uint16_t *length,
    int32_t *data, const char **log = NULL);
bool clearBulkReadParam(void);
bool torque(uint8_t id, bool onoff, const char **log = NULL);
bool torqueOn(uint8_t id, const char **log = NULL);
bool torqueOff(uint8_t id, const char **log = NULL);
bool changeID(uint8_t id, uint8_t new_id, const char **log = NULL);
bool changeBaudrate(uint8_t id, uint32_t new_baudrate, const char **log = NULL);
bool changeProtocolVersion(uint8_t id, uint8_t version, const char **log = NULL);
bool itemWrite(uint8_t id, const char *item_name, int32_t data, const char **log = NULL);
bool itemRead(uint8_t id, const char *item_name, int32_t *data, const char **log = NULL);
bool led(uint8_t id, bool onoff, const char **log = NULL);
bool ledOn(uint8_t id, const char **log = NULL);
bool ledOff(uint8_t id, const char **log = NULL);
bool setNormalDirection(uint8_t id, const char **log = NULL);
bool setReverseDirection(uint8_t id, const char **log = NULL);
bool setVelocityBasedProfile(uint8_t id, const char **log = NULL);
bool setTimeBasedProfile(uint8_t id, const char **log = NULL);
bool setSecondaryID(uint8_t id, uint8_t secondary_id, const char **log = NULL);
bool setCurrentControlMode(uint8_t id, const char **log = NULL);
bool setTorqueControlMode(uint8_t id, const char **log = NULL);
bool setVelocityControlMode(uint8_t id, const char **log = NULL);
bool setPositionControlMode(uint8_t id, const char **log = NULL);
bool setExtendedPositionControlMode(uint8_t id, const char **log = NULL);
bool setMultiTurnControlMode(uint8_t id, const char **log = NULL);
bool setCurrentBasedPositionControlMode(uint8_t id, const char **log = NULL);
bool setPWMControlMode(uint8_t id, const char **log = NULL);
bool setOperatingMode(uint8_t id, uint8_t index, const char **log = NULL);
bool jointMode(uint8_t id, int32_t velocity = 0, int32_t acceleration = 0, const char **
    log = NULL);
bool wheelMode(uint8_t id, int32_t acceleration = 0, const char **log = NULL);
bool currentBasedPositionMode(uint8_t id, int32_t current = 0, const char **log = NULL);
bool goalPosition(uint8_t id, int32_t value, const char **log = NULL);
bool goalPosition(uint8_t id, float radian, const char **log = NULL);
bool goalVelocity(uint8_t id, int32_t value, const char **log = NULL);
bool goalVelocity(uint8_t id, float velocity, const char **log = NULL);
bool getPresentPositionData(uint8_t id, int32_t* data, const char **log = NULL);
bool getRadian(uint8_t id, float* radian, const char **log = NULL);
```

```
bool getPresentVelocityData(uint8_t id, int32_t* data, const char **log = NULL);
bool getVelocity(uint8_t id, float* velocity, const char **log = NULL);
int32_t convertRadian2Value(uint8_t id, float radian);
float convertValue2Radian(uint8_t id, int32_t value);
int32_t convertRadian2Value(float radian, int32_t max_position, int32_t min_position,
    float max_radian, float min_radian);
float convertValue2Radian(int32_t value, int32_t max_position, int32_t min_position,
    float max_radian, float min_radian);
int32_t convertVelocity2Value(uint8_t id, float velocity);
float convertValue2Velocity(uint8_t id, int32_t value);
int16_t convertCurrent2Value(float current);
float convertValue2Current(int16_t value);
float convertValue2Load(int16_t value);
*****************************************************/
```

A.1.2.3 IMU, L298N controller and rosserial

```
#ifndef TURTLEBOT3_CORE_CONFIG_H_
#define TURTLEBOT3_CORE_CONFIG_H_
#include <ros.h>
#include <ros/time.h>
#include <std_msgs/Bool.h>
#include <std_msgs/Empty.h>
#include <std_msgs/Int32.h>
#include <std_msgs/Float64.h>
#include <sensor_msgs/Imu.h>
#include <sensor_msgs/JointState.h>
#include <sensor_msgs/BatteryState.h>
#include <sensor_msgs/MagneticField.h>
#include <geometry_msgs/Vector3.h>
#include <geometry_msgs/Twist.h>
#include <tf/tf.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <turtlebot3_msgs/SensorState.h>
#include <turtlebot3_msgs/Sound.h>
#include <turtlebot3_msgs/VersionInfo.h>
#include <TurtleBot3.h>
#include "turtlebot3_waffle.h"
#include "lidar_joint.h"

#include <stdarg.h>
#include <math.h>
#define FIRMWARE_VER "1.2.3"
#define CONTROL_MOTOR_SPEED_FREQUENCY          30   //hz
#define ENCODER_SAMPLE_FREQUENCY               100   //hz
#define CONTROL_MOTOR_TIMEOUT                  500  //ms
#define IMU_PUBLISH_FREQUENCY                  100  //hz
#define CMD_VEL_PUBLISH_FREQUENCY              30   //hz
#define DRIVE_INFORMATION_PUBLISH_FREQUENCY    30   //hz
#define JOINT_CONTROL_FREQEUNCY                10    //hz
#define DEBUG_LOG_FREQUENCY                    10   //hz
// #define DEBUG
//#define DEBUG_SERIAL                     Serial4 /

/*******************************************************************************
*******************************************************************************/
// Callback function prototypes
void commandVelocityCallback(const geometry_msgs::Twist& cmd_vel_msg);
void soundCallback(const turtlebot3_msgs::Sound& sound_msg);
//void motorPowerCallback(const std_msgs::Bool& power_msg);/
void resetCallback(const std_msgs::Empty& reset_msg);

// Function prototypes
void publishCmdVelFromRC100Msg(void);
void publishImuMsg(void);
void publishMagMsg(void);
void publishSensorStateMsg(void);
void publishVersionInfoMsg(void);
void publishBatteryStateMsg(void);
void publishDriveInformation(void);
ros::Time rosNow(void);
ros::Time addMicros(ros::Time & t, uint32_t _micros); // deprecated
void updateVariable(bool isConnected);
void updateMotorInfo(int32_t left_tick, int32_t right_tick);
void updateTime(void);
```

```
void updateOdometry(void);
void updateJoint(void);
void updateTF(geometry_msgs::TransformStamped& odom_tf);
void updateGyroCali(bool isConnected);
void updateGoalVelocity(void);
void updateTFPrefix(bool isConnected);
void initOdom(void);
void initJointStates(void);
bool calcOdometry(double diff_time);
void sendLogMsg(void);
void waitForSerialLink(bool isConnected);
/*******************************************************************************
* ROS NodeHandle
*******************************************************************************/
ros::NodeHandle nh;
ros::Time current_time;
uint32_t current_offset;
/*******************************************************************************
* ROS Parameter
*******************************************************************************/
char get_prefix[10];
char* get_tf_prefix = get_prefix;
char odom_header_frame_id[30];
char odom_child_frame_id[30];
char imu_frame_id[30];
char mag_frame_id[30];
char joint_state_header_frame_id[30];
/*******************************************************************************
* Subscriber
*******************************************************************************/
ros::Subscriber<geometry_msgs::Twist> cmd_vel_sub("cmd_vel", commandVelocityCallback);
ros::Subscriber<turtlebot3_msgs::Sound> sound_sub("sound", soundCallback);
//ros::Subscriber<std_msgs::Bool> motor_power_sub("motor_power", motorPowerCallback);/
ros::Subscriber<std_msgs::Empty> reset_sub("reset", resetCallback);


/*******************************************************************************
* Publisher
*******************************************************************************/
// Bumpers, cliffs, buttons, encoders, battery of Turtlebot3
turtlebot3_msgs::SensorState sensor_state_msg;
ros::Publisher sensor_state_pub("sensor_state", &sensor_state_msg);
// Version information of Turtlebot3
turtlebot3_msgs::VersionInfo version_info_msg;
ros::Publisher version_info_pub("firmware_version", &version_info_msg);
// IMU of Turtlebot3
sensor_msgs::Imu imu_msg;
ros::Publisher imu_pub("imu", &imu_msg);
// Command velocity of Turtlebot3 using RC100 remote controller
geometry_msgs::Twist cmd_vel_rc100_msg;
ros::Publisher cmd_vel_rc100_pub("cmd_vel_rc100", &cmd_vel_rc100_msg);
// Odometry of Turtlebot3
nav_msgs::Odometry odom;
ros::Publisher odom_pub("odom", &odom);
// Joint(Dynamixel) state of Turtlebot3
sensor_msgs::JointState joint_states;
ros::Publisher joint_states_pub("joint_states", &joint_states);
// Battey state of Turtlebot3
sensor_msgs::BatteryState battery_state_msg;
ros::Publisher battery_state_pub("battery_state", &battery_state_msg);
// Magnetic field
sensor_msgs::MagneticField mag_msg;
ros::Publisher mag_pub("magnetic_field", &mag_msg);
/*******************************************************************************
* Transform Broadcaster
*******************************************************************************/
// TF of Turtlebot3
geometry_msgs::TransformStamped odom_tf;
tf::TransformBroadcaster tf_broadcaster;
/*******************************************************************************
* SoftwareTimer of Turtlebot3
*******************************************************************************/
static uint32_t tTime[10];
/*******************************************************************************
* Declaration for motor
*******************************************************************************/
//Turtlebot3MotorDriver motor_driver;
/*******************************************************************************
* Calculation for odometry
```

```
*************************************************************************/
bool init_encoder = true;
int32_t last_diff_tick[WHEEL_NUM] = {0, 0};
double  last_rad[WHEEL_NUM]       = {0.0, 0.0};
/*************************************************************************
* Update Joint State
*************************************************************************/
double  last_velocity[WHEEL_NUM]  = {0.0, 0.0};
/*************************************************************************
* Declaration for sensors
*************************************************************************/
Turtlebot3Sensor sensors;
/*************************************************************************
* Declaration for controllers
*************************************************************************/
Turtlebot3Controller controllers;
float zero_velocity[WHEEL_NUM] = {0.0, 0.0};
float goal_velocity[WHEEL_NUM] = {0.0, 0.0};
float goal_velocity_from_button[WHEEL_NUM] = {0.0, 0.0};
float goal_velocity_from_cmd[WHEEL_NUM] = {0.0, 0.0};
/*************************************************************************
* Declaration for diagnosis
*************************************************************************/
Turtlebot3Diagnosis diagnosis;
/*************************************************************************
* Declaration for SLAM and navigation
*************************************************************************/
unsigned long prev_update_time;
float odom_pose[3];
double odom_vel[3];
/*************************************************************************
* Declaration for Battery
*************************************************************************/
bool setup_end        = false;
uint8_t battery_state = 0;
//LiDAR_joint_Driver LiDAR_driver;
#endif // TURTLEBOT3_CORE_CONFIG_H_
```

A.1.2.4 Configurations

```
#ifndef TURTLEBOT3_WAFFLE_H_
#define TURTLEBOT3_WAFFLE_H_
#define NAME                             "Waffle or Waffle Pi"
#define WHEEL_RADIUS                     0.1            // meter
#define WHEEL_SEPARATION                 0.34           // meter (BURGER : 0.160, WAFFLE : 0.287)
#define TURNING_RADIUS                   0.1435         // meter (BURGER : 0.080, WAFFLE : 0.1435)
#define ROBOT_RADIUS                     0.220          // meter (BURGER : 0.105, WAFFLE : 0.220)
#define ENCODER_MIN                      -2147483648    // raw
#define ENCODER_MAX                      2147483648     // raw
#define MAX_LINEAR_VELOCITY              (WHEEL_RADIUS * 2 * 3.14159265359 * 3000/ 60) // m/s  (BURGER : 61[rpm], WAFFLE : 77[rpm])
#define MAX_ANGULAR_VELOCITY             (MAX_LINEAR_VELOCITY / TURNING_RADIUS)       // rad/s
#define MIN_LINEAR_VELOCITY              -MAX_LINEAR_VELOCITY
#define MIN_ANGULAR_VELOCITY             -MAX_ANGULAR_VELOCITY
#endif  //TURTLEBOT3_WAFFLE_H_
```

A.2 Hector SLAM codes

This section documented the source code of modified Hector SLAM described in Chapter 3 and 4, including the functionality of ITM matching algorithm, timestamped grid map, and multi-level grid map correction.

A.2.1 Laser tracker receiver

```cpp
#include <ros/ros.h>
#include <tf2_ros/transform_broadcaster.h>
#include <tf2/transform_datatypes.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2/LinearMath/Matrix3x3.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2/convert.h>
#include <math.h>
#include "std_msgs/String.h"
#include <nav_msgs/Path.h>
#include <nav_msgs/Odometry.h>
#include <sensor_msgs/NavSatFix.h>
#include <gps_common/GPSFix.h>
#include <gps_common/conversions.h>
#include <pcl_ros/point_cloud.h>
#include <geometry_msgs/Vector3Stamped.h>
#include <geometry_msgs/Point.h>
#include <geometry_msgs/Transform.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/registration/icp.h>
#include <hector_icp/PLICP_Trans.h>//include custom msg type
// #include <iostream>
#include "icpPointToPoint.h"
#include <iostream>
#include <fstream>
using namespace std;

class icp_iter_class{
public:

  icp_iter_class(){
    tracker_location_init = 0;
    laser_count = 0;
    hactor_count = 0;
    laser_offset = 0;
    hactor_offset = 0;
    laser_stamp_min.fromSec(0.0);
    laser_stamp_max.fromSec(0.0);
    first_call_escaper = 0;
    orientation_aline_escaper = 0;
    gps_location_init = false;
    path_flag = 1;
    min_residual = 1.00;
    round_hector_points = 0;
    round_tracker_points = 0;
    sampling_round = 5;
    sampling_num = 200;
    sub_ = n_.subscribe("/trajectory", 50, &icp_iter_class::hector_path_Callback, this);
    tracker_sub = n_.subscribe("/Tracker_xyz", 50, &icp_iter_class::tacker_callback, this
    );
    gps_sub = n_.subscribe("/solution", 50, &icp_iter_class::slu_callback, this);
    pose_sub = n_.subscribe("/Ground_Truth", 50, &icp_iter_class::pose_callback, this);
    timer  = n_.createTimer(ros::Duration(1), &icp_iter_class::icp_iter, this);
    icp_path_pub = n_.advertise<pcl::PointCloud<pcl::PointXYZ> >("icp_path", 1);
    hector_path_pub = n_.advertise<pcl::PointCloud<pcl::PointXYZ> >("Hector_path", 1);
    tracker_path_pub = n_.advertise<pcl::PointCloud<pcl::PointXYZ> >("Tracker_path", 1);
    PCL_Trans_pub = n_.advertise<hector_icp::PLICP_Trans>("PCL_Trans", 1);
    // slu_pub = n_.advertise<geometry_msgs::PoseStamped>("Ground_Truth", 1);
  }

  void icp_iter(const ros::TimerEvent&) {
    if (first_call_escaper < 2){
      ROS_INFO("Skip 5 sec %d", first_call_escaper); // with /clock some reason ROS will
    call timer event twice when init.
      first_call_escaper++;
      }
    else{
        if(laser_count >= path_flag*sampling_num && path_flag <= sampling_round){
          path_flag ++;
          ROS_INFO("Correctiong orientation");
          float factor;
          int32_t dim = 2;
          // int32_t num = 100;
          int32_t i = 0;
          int32_t j = 0;
```

```
        double* M = (double*)calloc(3*hactor_count,sizeof(double));
        double* T = (double*)calloc(3*laser_count,sizeof(double));
        //publisher for laser trajectory
        pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Tracker (new pcl::PointCloud<pcl::PointXYZ>);
        cloud_Tracker->width    = 50;
        cloud_Tracker->height   = 50;
        cloud_Tracker->points.resize (cloud_Tracker->width * cloud_Tracker->height);
        pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Hector (new pcl::PointCloud<pcl::PointXYZ>);
        cloud_Hector->width    = 50;
        cloud_Hector->height   = 50;
        cloud_Hector->points.resize (cloud_Hector->width * cloud_Hector->height);
        round_hector_points = hactor_count-hactor_offset;
        ROS_INFO("Processing hactor points from %d to %d, processed [%f] points", hactor_offset, hactor_count, round_hector_points);
        while(hactor_offset < Hactor_Path.poses.size() ){
          M[i*dim+0] = Hactor_Path.poses[hactor_offset].pose.position.x;
          M[i*dim+1] = Hactor_Path.poses[hactor_offset].pose.position.y;
          // M[i*dim+2] = Hactor_Path.poses[hactor_offset].pose.position.z;
          cloud_Hector->points[i].x = Hactor_Path.poses[hactor_offset].pose.position.x;
          cloud_Hector->points[i].y = Hactor_Path.poses[hactor_offset].pose.position.y;
          hactor_offset++;
          i++;
        }
        // ROS_INFO_STREAM("Hector cloud is" << std::endl << *M << std::endl);
        // hactor_offset = Hactor_Path.poses.size();
        laser_stamp_min = Laser_Path.poses[laser_offset].header.stamp;
        laser_stamp_max = Laser_Path.poses[laser_count-1].header.stamp;
        round_tracker_points = laser_count-laser_offset;
        factor = round_tracker_points / round_hector_points;
        // ROS_INFO("Checking########## %f, %f, %f", round_hector_points, round_tracker_points, factor);
        ROS_INFO("Processing Tracker points from %i to %i, processed [%f] points.", laser_offset, laser_count, round_tracker_points);
        ROS_INFO("From %f to %f.", laser_stamp_min.toSec(), laser_stamp_max.toSec());
        while(laser_offset < laser_count){
          T[j*dim+0] = Laser_Path.poses[laser_offset].pose.position.x;
          T[j*dim+1] = Laser_Path.poses[laser_offset].pose.position.y;
          //for display only
          cloud_Tracker->points[j].x = Laser_Path.poses[laser_offset].pose.position.x;
          cloud_Tracker->points[j].y = Laser_Path.poses[laser_offset].pose.position.y;
          // T[j*dim+2] = Laser_Path.poses[laser_offset].pose.position.z;
          laser_offset = ceil(factor + laser_offset);
          j++;
        }
        ROS_INFO("Tracker points downsamping from %d to %d points.", (laser_offset-laser_count), j);
        laser_offset = laser_count;
        //downsamping to match number of points

        pcl_conversions::toPCL(ros::Time::now(), cloud_Hector->header.stamp);
        cloud_Hector->header.frame_id = "/map";
        hector_path_pub.publish(cloud_Hector);
        pcl_conversions::toPCL(ros::Time::now(), cloud_Tracker->header.stamp);
        cloud_Tracker->header.frame_id = "/map";
        tracker_path_pub.publish(cloud_Tracker);

        // ROS_INFO_STREAM("Laser cloud is" << std::endl << *T << std::endl);
        // start with identity as initial transformation
        // in practice you might want to use some kind of prediction here
        Matrix R = Matrix::eye(dim);
        Matrix t(dim,1);
        // run point-to-plane ICP (-1 = no outlier threshold)
        ROS_INFO("Running ICP (point-to-plane, no outliers)");
        IcpPointToPoint icp(T,j,dim);
        double residual = icp.fit(M,i,R,t,-1);//T*R+t = M
        ROS_INFO("Residual is: %f", residual);
        // ROS_INFO_STREAM("Correspondences: " << icp.correspondences << std::endl << " inlierNum: "<< icp.inlierNum << std::endl<< "inlierIdx: "<< icp.inlierIdx);
        // if success and residual is bigger than something!? lucas
        if (residual < min_residual){
          min_residual = residual;
          // ROS_INFO("ICP calculated");
          ROS_INFO("##################### Alineing Reference Frame ###################### \n");
          // ROS_INFO("C Posted");
```

```
        // ROS_INFO("F Posted");
        ROS_INFO_STREAM("Transformation is" << std::endl << t << std::endl);
        ROS_INFO_STREAM("Rotation is" << std::endl << R << std::endl);
        path_rotation = R;
        path_translation = t;
        rotatePoint(M, i, R, t, dim);
        // ROS_INFO_STREAM("Points after Rotation is Stored @" << std::endl << T << std::endl);
        pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_ICP (new pcl::PointCloud<pcl::PointXYZ>);
        cloud_ICP->width    = 100;
        cloud_ICP->height   = 100;
        cloud_ICP->points.resize (cloud_ICP->width * cloud_ICP->height);
        for (int c = 0; c < i; ++c) {
          // ROS_INFO("Writing %d",c);
          cloud_ICP->points[c].x = M[dim * c + 0];
          cloud_ICP->points[c].y = M[dim * c + 1];
          }
        ROS_INFO("Write into point cloud");
        // ROS_INFO("T Posted");
        pcl_conversions::toPCL(ros::Time::now(), cloud_ICP->header.stamp);
        cloud_ICP->header.frame_id = "map";
        icp_path_pub.publish(cloud_ICP);
      }
      else{
        ROS_INFO("\n######Residual not valid, ICP has not converged.######\n");
      }
      // free memory
      free(M);
      free(T);
      if(path_flag == sampling_round){//last prepare cycle && first piblish
        t.getData(val_tm);
        Matrix Rr = Matrix::inv(R);
        Rr.getData(val_Rm);
        //publish trans
        //sending transformation: no matter last icp fited or not.
        hector_icp::PLICP_Trans new_frame_trans;
        new_frame_trans.header.stamp = ros::Time::now();
        new_frame_trans.header.frame_id = "map";
        new_frame_trans.child_frame_id = "pcl_transed";
        new_frame_trans.transform.data.resize(2);
        new_frame_trans.transform.data[0]= val_tm[0];
        new_frame_trans.transform.data[1]= val_tm[1];
        new_frame_trans.rotation.data.resize(4);
        new_frame_trans.rotation.data[0]= val_Rm[0];
        new_frame_trans.rotation.data[1]= val_Rm[1];
        new_frame_trans.rotation.data[2]= val_Rm[2];
        new_frame_trans.rotation.data[3]= val_Rm[3];
        new_frame_trans.begin_stamp = laser_stamp_min;
        new_frame_trans.end_stamp = laser_stamp_max;
        PCL_Trans_pub.publish(new_frame_trans);//msg TF transform
        ROS_INFO("Current Trans Time: %f to %f\n", laser_stamp_min.toSec(), laser_stamp_max.toSec());
      }
    }
  else if(laser_count >= path_flag*sampling_num && path_flag > sampling_round){
      // exit (0);
      path_flag++;
      // myfileH.open("H.txt");
      // myfileT.open("T.txt");
      // myfileI.open("I.txt");
      ROS_INFO("ICP CALLED");
      int32_t dim = 2;
      // int32_t num = 100;
      int32_t i = 0;
      int32_t j = 0;
      double* M = (double*)calloc(3*hactor_count,sizeof(double));
      double* T = (double*)calloc(3*laser_count,sizeof(double));
      //publisher for laser trajectory
      pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Tracker (new pcl::PointCloud<pcl::PointXYZ>);
      cloud_Tracker->width    = 30;
      cloud_Tracker->height   = 30;
      cloud_Tracker->points.resize (cloud_Tracker->width * cloud_Tracker->height);
      pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Hector (new pcl::PointCloud<pcl::PointXYZ>);
      cloud_Hector->width    = 30;
      cloud_Hector->height   = 30;
```

```
        cloud_Hector->points.resize (cloud_Hector->width * cloud_Hector->height);
        ROS_INFO("Processing hactor points from %d to %d, processed [%d] points", hactor_offset, hactor_count, (hactor_count-hactor_offset));
        while(hactor_offset < Hactor_Path.poses.size() ){
          M[i*dim+0] = Hactor_Path.poses[hactor_offset].pose.position.x;
          M[i*dim+1] = Hactor_Path.poses[hactor_offset].pose.position.y;
          // M[i*dim+2] = Hactor_Path.poses[hactor_offset].pose.position.z;
          cloud_Hector->points[i].x = Hactor_Path.poses[hactor_offset].pose.position.x;
          cloud_Hector->points[i].y = Hactor_Path.poses[hactor_offset].pose.position.y;
          // myfileH << cloud_Hector->points[i].x << " " <<cloud_Hector->points[i].y << std::endl;
          hactor_offset++;
          i++;
        }
        // ROS_INFO_STREAM("Hector cloud is" << std::endl << *M << std::endl);
        // hactor_offset = Hactor_Path.poses.size();

        laser_stamp_min = Laser_Path.poses[laser_offset].header.stamp;
        laser_stamp_max = Laser_Path.poses[laser_count-1].header.stamp;
        ROS_INFO("Processing Tracker points from %d to %d, processed [%d] points, from %f to %f.", laser_offset, laser_count, (laser_count-laser_offset), laser_stamp_min.toSec(), laser_stamp_max.toSec());
        int laser_offset_copy = laser_offset;
        while(laser_offset < laser_count){
          T[j*dim+0] = Laser_Path.poses[laser_offset].pose.position.x;
          T[j*dim+1] = Laser_Path.poses[laser_offset].pose.position.y;
          //for display only
          // cloud_Tracker->points[j].x = Laser_Path.poses[laser_offset].pose.position.x;
          // cloud_Tracker->points[j].y = Laser_Path.poses[laser_offset].pose.position.y;
          // T[j*dim+2] = Laser_Path.poses[laser_offset].pose.position.z;
          // myfileT << cloud_Tracker->points[j].x << " " << cloud_Tracker->points[j].y << std::endl;
          laser_offset++;
          j++;
        }
        // rotatePoint(T, j, path_rotation, path_translation, dim);//orientation correction from init
        j = 0;
        while(laser_offset_copy < laser_count){
          //for display only
          cloud_Tracker->points[j].x = T[j*dim+0];
          cloud_Tracker->points[j].y = T[j*dim+1];
          // T[j*dim+2] = Laser_Path.poses[laser_offset].pose.position.z;
          // myfileT << cloud_Tracker->points[j].x << " " << cloud_Tracker->points[j].y << std::endl;
          laser_offset_copy++;
          j++;
        }
        pcl_conversions::toPCL(ros::Time::now(), cloud_Hector->header.stamp);
        cloud_Hector->header.frame_id = "/map";
        hector_path_pub.publish(cloud_Hector);
        pcl_conversions::toPCL(ros::Time::now(), cloud_Tracker->header.stamp);
        cloud_Tracker->header.frame_id = "/map";
        tracker_path_pub.publish(cloud_Tracker);

        // ROS_INFO_STREAM("Laser cloud is" << std::endl << *T << std::endl);
        // start with identity as initial transformation
        // in practice you might want to use some kind of prediction here
        Matrix R = Matrix::eye(dim);
        Matrix t(dim,1);
        // run point-to-plane ICP (-1 = no outlier threshold)
        ROS_INFO("Running ICP (point-to-plane, no outliers)");
        IcpPointToPoint icp(T,j,dim);
        double residual = icp.fit(M,i,R,t,-1);

        // if success and residual is bigger than something!? lucas
        if (residual < 0.01){
          // ROS_INFO("ICP calculated");
          ROS_INFO("###################### Sending Rotation Matrix ###################### \n");
          ROS_INFO("Residual is: %f", residual);
          // ROS_INFO("C Posted");
          // ROS_INFO("F Posted");
          ROS_INFO_STREAM("Transformation is" << std::endl << t << std::endl);
          ROS_INFO_STREAM("Rotation is" << std::endl << R << std::endl);
          rotatePoint(M, i, R, t, dim);
```

```
            // ROS_INFO_STREAM("Points after Rotation is Stored @" << std::endl << T << std::endl);
            pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_ICP (new pcl::PointCloud<pcl::PointXYZ>);
            cloud_ICP->width    = 30;
            cloud_ICP->height   = 30;
            cloud_ICP->points.resize (cloud_ICP->width * cloud_ICP->height);
            for (int c = 0; c < i; ++c) {
              // ROS_INFO("Writing %d",c);
              cloud_ICP->points[c].x = M[dim * c + 0];
              cloud_ICP->points[c].y = M[dim * c + 1];
              // myfileI << cloud_ICP->points[c].x << " " << cloud_ICP->points[c].y << std::endl;
            }
            ROS_INFO("Write into point cloud");
            // ROS_INFO("T Posted");
            pcl_conversions::toPCL(ros::Time::now(), cloud_ICP->header.stamp);
            cloud_ICP->header.frame_id = "map";
            icp_path_pub.publish(cloud_ICP);
            t.getData(val_tm);
            Matrix Rr = Matrix::inv(R);
            Rr.getData(val_Rm);
            // ROS_INFO_STREAM(std::endl << val_Rm[0] << val_Rm[1] << val_Rm[2] << val_Rm[3]  << std::endl << val_tm[0] << val_tm[1] << std::endl);
            //publish trans
            //sending transformation: no matter last icp fited or not.
            hector_icp::PLICP_Trans new_frame_trans;
            new_frame_trans.header.stamp = ros::Time::now();
            new_frame_trans.header.frame_id = "map";
            new_frame_trans.child_frame_id = "pcl_transed";
            new_frame_trans.transform.data.resize(2);
            new_frame_trans.transform.data[0]= val_tm[0];
            new_frame_trans.transform.data[1]= val_tm[1];
            new_frame_trans.rotation.data.resize(4);
            new_frame_trans.rotation.data[0]= val_Rm[0];
            new_frame_trans.rotation.data[1]= val_Rm[1];
            new_frame_trans.rotation.data[2]= val_Rm[2];
            new_frame_trans.rotation.data[3]= val_Rm[3];
            new_frame_trans.begin_stamp = laser_stamp_min;
            new_frame_trans.end_stamp = laser_stamp_max;
            PCL_Trans_pub.publish(new_frame_trans);//msg TF transform
            ROS_INFO("Current Trans Time: %f to %f\n", laser_stamp_min.toSec(), laser_stamp_max.toSec());
            // myfileH.close();
            // myfileT.close();
            // myfileI.close();
          }
          else{
            ROS_INFO("######Residual not valid, ICP has not converged.######");
          }
          // free memory
          free(M);
          free(T);
        }
      else{
          ROS_INFO("waiting for more Reference points");
        }
      }
    }
  void rotatePoint(double *&T, int num2, Matrix R, Matrix t, int32_t dim) {
    for (int i = 0; i < num2; ++i) {
      FLOAT *val = new FLOAT[dim];
      val[0] = (FLOAT)(T[dim * i + 0]);
      val[1] = (FLOAT)(T[dim * i + 1]);
      Matrix point(dim, 1, val);
      Matrix pointout = R * point + t;
      // Matrix pointout = R * point;
      T[dim * i + 0] = pointout.val[0][0];
      T[dim * i + 1] = pointout.val[1][0];
    }
  }
  void hector_path_Callback(const nav_msgs::Path& msg)
  {
    Hactor_Path = msg;
    hactor_count = Hactor_Path.poses.size();
    // ROS_INFO("Saved [%d] Hactor points", hactor_count);
  }
```

```
void tacker_callback( const geometry_msgs::Vector3Stamped& laser_msg){
  if (tracker_location_init = 0){
      tracker_location_init_x = laser_msg.vector.x;
      tracker_location_init_y = laser_msg.vector.y;
      tracker_location_init_z = laser_msg.vector.z;
      tracker_location_init ++;
      windowsXP_offset = ros::Time::now() - laser_msg.header.stamp;
    }
  // else if(tracker_location_init == 4){
  //   tracker_location_init_x += laser_msg.vector.x;
  //   tracker_location_init_y += laser_msg.vector.y;
  //   tracker_location_init_z += laser_msg.vector.z;
  //
  //   tracker_location_init_x = tracker_location_init_x/5;
  //   tracker_location_init_y = tracker_location_init_y/5;
  //   tracker_location_init_z = tracker_location_init_z/5;
  //   tracker_location_init ++;
  // }
  else{
    //callback every time the leica's xyz is received
    ros::Time current_time = ros::Time::now();
    geometry_msgs::PoseStamped current_point;
    current_point.header.stamp = laser_msg.header.stamp + windowsXP_offset;
    current_point.header.frame_id = "map";
    current_point.pose.position.x = (laser_msg.vector.x - tracker_location_init_x)+2;
    current_point.pose.position.y = (laser_msg.vector.y - tracker_location_init_y)+1;
    current_point.pose.position.z = (laser_msg.vector.z - tracker_location_init_z)+1.5;
  //only 2D pose for now
    // ROS_INFO("Time header [%f]", current_point.header.stamp.toSec());
    Laser_Path.header.stamp = current_point.header.stamp;
    Laser_Path.header.frame_id = "map";
    Laser_Path.poses.push_back(current_point);
    laser_count++;
  }
  // ROS_INFO("Saved [%d] Tracker points", laser_count);
}
void slu_callback(const nav_msgs::Odometry& slu) {
  // if (fix.status.status == sensor_msgs::NavSatStatus::STATUS_NO_FIX) {
  //   ROS_INFO("No fix.");
  //   return;
  // }
  // if (path_flag != true && slu.header.stamp == ros::Time(0)) {
  //   return;
  // }
  geometry_msgs::PoseStamped current_point;
  current_point.header.stamp = slu.header.stamp;
  current_point.header.frame_id = "map";
  // ROS_INFO("GPS TO MAP Coords: %f & %f\n", slu.pose.pose.position.x, Hactor_Path.
  poses[Hactor_Path.poses.size()].pose.position.y);
  current_point.pose.position.x = slu.pose.pose.position.x;
  current_point.pose.position.y = slu.pose.pose.position.y;
  current_point.pose.position.z = slu.pose.pose.position.z;
  // slu_pub.publish(current_point);
  // ROS_INFO("Time header [%f]", current_point.header.stamp.toSec());
  // ROS_INFO("GPS TO MAP Coords: %f & %f\n", current_point.pose.position.x,
  current_point.pose.position.y);
  Laser_Path.header.stamp = slu.header.stamp;
  Laser_Path.header.frame_id = "map";
  Laser_Path.poses.push_back(current_point);
  laser_count++;
}
void pose_callback(const geometry_msgs::PoseStamped& pose) {
  // Current_Pose = pose;
  geometry_msgs::PoseStamped current_point;
  current_point.header.stamp = pose.header.stamp;
  current_point.header.frame_id = "map";
  // ROS_INFO("GPS TO MAP Coords: %f & %f\n", slu.pose.pose.position.x, Hactor_Path.
  poses[Hactor_Path.poses.size()].pose.position.y);
  current_point.pose.position.x = pose.pose.position.x;
  current_point.pose.position.y = pose.pose.position.y;
  current_point.pose.position.z = pose.pose.position.z;
  // slu_pub.publish(current_point);
  // ROS_INFO("Time header [%f]", current_point.header.stamp.toSec());
  // ROS_INFO("GPS TO MAP Coords: %f & %f\n", current_point.pose.position.x,
  current_point.pose.position.y);
  Laser_Path.header.stamp = pose.header.stamp;
  Laser_Path.header.frame_id = "map";
  Laser_Path.poses.push_back(current_point);
  laser_count++;
```

```
  }
  // void cloud_resize(float round_tracker_points,float round_hector_points){
  //   factor = round_tracker_points / round_hector_points;
  // }
// private:
  ros::NodeHandle n_;
  ros::Subscriber sub_;
  ros::Subscriber tracker_sub;
  ros::Subscriber gps_sub;
  ros::Subscriber pose_sub;
  ros::Timer timer;
  ros::Publisher icp_path_pub;
  ros::Publisher hector_path_pub;
  ros::Publisher tracker_path_pub;
  ros::Publisher PCL_Trans_pub;
  ros::Publisher slu_pub;
  nav_msgs::Path Hactor_Path;
  nav_msgs::Path Laser_Path;
  // geometry_msgs::PoseStamped Current_Pose;
  tf2_ros::TransformBroadcaster pcl_br;
  int tracker_location_init;
  float tracker_location_init_x;
  float tracker_location_init_y;
  float tracker_location_init_z;
  bool gps_location_init;
  float gps_location_init_x;
  float gps_location_init_y;
  float gps_location_init_z;
  int laser_count;
  int hactor_count;
  int laser_offset;
  int hactor_offset;
  ros::Time laser_stamp_min;
  ros::Time laser_stamp_max;
  int first_call_escaper;
  int orientation_aline_escaper;
  ros::Duration windowsXP_offset;
  int path_flag;
  double min_residual;
  Matrix path_rotation;
  Matrix path_translation;
  int sampling_round;
  int sampling_num;
  float  round_hector_points;
  float  round_tracker_points;
  float val_tm[2*2];  //two matrix for sending transform msg
  float val_Rm[2*2];
  // ofstream myfileH;
  // ofstream myfileT;
  // ofstream myfileI;
};

int main(int argc, char **argv)
{
  ros::init(argc, argv, "path_icp");
  icp_iter_class my_pcl;
  ros::spin();
  return 0;
}
```

A.2.2 Hector main method

```
#ifndef _hectorslamprocessor_h__
#define _hectorslamprocessor_h__
#include "../map/GridMap.h"
#include "../map/OccGridMapUtilConfig.h"
#include "../matcher/ScanMatcher.h"
#include "../scan/DataPointContainer.h"
#include "../util/UtilFunctions.h"
#include "../util/DrawInterface.h"
#include "../util/HectorDebugInfoInterface.h"
#include "../util/MapLockerInterface.h"
#include "MapRepresentationInterface.h"
#include "MapRepMultiMap.h"

#include <float.h>
namespace hectorslam{
class HectorSlamProcessor
{
public:
```

```
HectorSlamProcessor(float mapResolution, int mapSizeX, int mapSizeY , const Eigen::
  Vector2f& startCoords, int multi_res_size, DrawInterface* drawInterfaceIn = 0,
  HectorDebugInfoInterface* debugInterfaceIn = 0)
    : drawInterface(drawInterfaceIn)
    , debugInterface(debugInterfaceIn)
  {
    mapRep = new MapRepMultiMap(mapResolution, mapSizeX, mapSizeY, multi_res_size,
    startCoords, drawInterfaceIn, debugInterfaceIn);
    this->reset();
    this->setMapUpdateMinDistDiff(0.4f *1.0f);
    this->setMapUpdateMinAngleDiff(0.13f * 1.0f);
  }
  ~HectorSlamProcessor()
  {
    delete mapRep;
  }
  void update(const DataContainer& dataContainer, const Eigen::Vector3f& poseHintWorld,
    bool map_without_matching = false)
  {
    //std::cout << "\nph:\n" << poseHintWorld << "\n";
    Eigen::Vector3f newPoseEstimateWorld;
    if (!map_without_matching){
        newPoseEstimateWorld = (mapRep->matchData(poseHintWorld, dataContainer,
    lastScanMatchCov));
    }else{
        newPoseEstimateWorld = poseHintWorld;
    }
    lastScanMatchPose = newPoseEstimateWorld;
    //std::cout << "\nt1:\n" << newPoseEstimateWorld << "\n";
    //std::cout << "\n1";
    //std::cout << "\n" << lastScanMatchPose << "\n";
    if(util::poseDifferenceLargerThan(newPoseEstimateWorld, lastMapUpdatePose,
    paramMinDistanceDiffForMapUpdate, paramMinAngleDiffForMapUpdate) ||
    map_without_matching){
      mapRep->updateByScan(dataContainer, newPoseEstimateWorld);
      mapRep->onMapUpdated();
      lastMapUpdatePose = newPoseEstimateWorld;
    }
    if(drawInterface){
      const GridMap& gridMapRef (mapRep->getGridMap());
      drawInterface->setColor(1.0, 0.0, 0.0);
      drawInterface->setScale(0.15);
      drawInterface->drawPoint(gridMapRef.getWorldCoords(Eigen::Vector2f::Zero()));
      drawInterface->drawPoint(gridMapRef.getWorldCoords((gridMapRef.getMapDimensions().
    array()-1).cast<float>()));
      drawInterface->drawPoint(Eigen::Vector2f(1.0f, 1.0f));
      drawInterface->sendAndResetData();
    }
    if (debugInterface)
    {
      debugInterface->sendAndResetData();
    }
  }
  void updateByLaser(int *SetOccupie, int *SetFree, Eigen::Vector3f *lastpose){// lucas
    this is getting wild, so many places need to be modified
    mapRep->updateByLaser(SetOccupie, SetFree, lastpose);
    // ROS_INFO("Current Pose: X %f, Y %f, Z %f,", lastpose[0][0], lastpose[0][1],
    lastpose[0][2]);
    mapRep->onMapUpdated();
    lastScanMatchPose = Eigen::Vector3f(lastpose[0][0], lastpose[0][1], lastpose[0][2]);
    // ROS_INFO("Current Pose: X %f, Y %f, Z %f,", lastScanMatchPose[0],
    lastScanMatchPose[1], lastScanMatchPose[2]);
    ROS_INFO("Pose Updated");
    // lastMapUpdatePose = lastScanMatchPose;
  }
  void reset()
  {
    lastMapUpdatePose = Eigen::Vector3f(FLT_MAX, FLT_MAX, FLT_MAX);
    lastScanMatchPose = Eigen::Vector3f::Zero();
    //lastScanMatchPose.x() = -10.0f;
    //lastScanMatchPose.y() = -15.0f;
    //lastScanMatchPose.z() = M_PI*0.15f;
    mapRep->reset();
  }
  const Eigen::Vector3f& getLastScanMatchPose() const { return lastScanMatchPose; };
  const Eigen::Matrix3f& getLastScanMatchCovariance() const { return lastScanMatchCov; };
```

```
  float getScaleToMap() const { return mapRep->getScaleToMap(); };
  int getMapLevels() const { return mapRep->getMapLevels(); };
  const GridMap& getGridMap(int mapLevel = 0) const { return mapRep->getGridMap(mapLevel)
    ; };
  void addMapMutex(int i, MapLockerInterface* mapMutex) { mapRep->addMapMutex(i, mapMutex
    ); };
  MapLockerInterface* getMapMutex(int i) { return mapRep->getMapMutex(i); };
  void setUpdateFactorFree(float free_factor) { mapRep->setUpdateFactorFree(free_factor);
     };
  void setUpdateFactorOccupied(float occupied_factor) { mapRep->setUpdateFactorOccupied(
    occupied_factor); };
  void setMapUpdateMinDistDiff(float minDist) { paramMinDistanceDiffForMapUpdate =
    minDist; };
  void setMapUpdateMinAngleDiff(float angleChange) { paramMinAngleDiffForMapUpdate =
    angleChange; };
protected:
  MapRepresentationInterface* mapRep;
  Eigen::Vector3f lastMapUpdatePose;
  Eigen::Vector3f lastScanMatchPose;
  Eigen::Matrix3f lastScanMatchCov;
  float paramMinDistanceDiffForMapUpdate;
  float paramMinAngleDiffForMapUpdate;
  DrawInterface* drawInterface;
  HectorDebugInfoInterface* debugInterface;
};
}
#endif
```

A.2.3 Timestamped grid map

```
#ifndef __GridMapTimerCount_h_
#define __GridMapTimerCount_h_
#include <cmath>
/**
 * Provides a log odds of occupancy probability representation for cells in a occupancy
    grid map.
 */
class TimerCountCell
{
public:
  TimerCountCell(){
    // mapPoseCount = 99;
    timeStamp.fromSec(0.0);
    shifted = false;
  }
  /*
  void setOccupied()
  {
    logOddsVal += 0.7f;
  };
  void setFree()
  {
    logOddsVal -= 0.4f;
  };
  */

  /**
   * Sets the cell value to val.
   * @param val The log odds value.
   */
  void set(float val)
  {
    logOddsVal = val;
  }
  ros::Time getTimeStamp() const //lucas
  {
    return timeStamp;
  }
  void setTimeStamp()//lucas
  {
    timeStamp = ros::Time::now();
    // ROS_INFO("Current map time: %f",timeStamp.toSec());
  }
  /**
   * Returns the value of the cell.
   * @return The log odds value.
   */
```

```
  float getValue() const
  {
    return logOddsVal;
  }
  /**
   * Returns wether the cell is occupied.
   * @return Cell is occupied
   */
  bool isOccupied() const
  {
    return logOddsVal > 0.0f;
  }
  bool isFree() const
  {
    return logOddsVal < 0.0f;
  }
  bool isshifted() const
  {
    return shifted;
  }
  /**
   * Reset Cell to prior probability.
   */
  void resetGridCell()
  {
    logOddsVal = -0.1f;
    updateIndex = -1;
    timeStamp.fromSec(0.0);//lucas
    shifted = false;//lucas
  }
  //protected:
public:
  float logOddsVal; ///< The log odds representation of occupancy probability.
  int updateIndex;
  ros::Time timeStamp;//lucas
  bool shifted;//lucas
  // Eigen::Vector3f mapPose[5];
  // int mapPoseCount;
};
/**
 * Provides functions related to a log odds of occupancy probability respresentation for cells in a occupancy grid map.
 */
class GridMapTimerCountFunctions
{
public:
  /**
   * Constructor, sets parameters like free and occupied log odds ratios.
   */
  GridMapTimerCountFunctions()
  {
    this->setUpdateFreeFactor(0.4f);
    this->setUpdateOccupiedFactor(0.6f);
    /*
    //float probOccupied = 0.6f;
    float probOccupied = 0.9f;
    float oddsOccupied = probOccupied / (1.0f - probOccupied);
    logOddsOccupied = log(oddsOccupied);
    float probFree = 0.4f;
    float oddsFree = probFree / (1.0f - probFree);
    logOddsFree = log(oddsFree);
    */
  }
  /**
   * Update cell as occupied
   * @param cell The cell.
   */
  void updateSetOccupied(TimerCountCell& cell) const
  {
    if (cell.logOddsVal < 50.0f){
      cell.logOddsVal += logOddsOccupied;
      // cell.timeStamp = TimeStamp; no need as called in other functions lucas
    }
  }
  void updateTimeStamp(TimerCountCell& cell) const//lucas mappose remoced
  {
    cell.setTimeStamp();
    // if (cell.mapPoseCount == 99 || cell.mapPoseCount == 5){
```

```
    //   cell.mapPoseCount = 0;
    //   cell.mapPose[cell.mapPoseCount] = mapPose ;
    //
    //   // std::cout << "X " << cell.mapPose[cell.mapPoseCount][0] << " Y " <<cell.
    mapPose[cell.mapPoseCount][1] << " H " << cell.mapPose[cell.mapPoseCount][2] << "\n";
    //
    //   cell.mapPoseCount++;
    //   // std::cout << "No count\n";
    // }
    // else{
    //   cell.mapPose[cell.mapPoseCount] = mapPose ;
    //   cell.mapPoseCount++;
    //   // std::cout << "Yes count\n";
    // }
    // std::cout.precision(20);
    // std::cout << "Time Stamped\n" << std::fixed << cell.timeStamp;
  }
  /**
   * Update cell as free
   * @param cell The cell.
   */
  void updateSetFree(TimerCountCell& cell) const
  {
    cell.logOddsVal += logOddsFree;
  }
  void updateUnsetFree(TimerCountCell& cell) const
  {
    cell.logOddsVal -= logOddsFree;
    //cell.timeStamp = TimeStamp; no need as called in other functions lucas
  }
  /**
   * Get the probability value represented by the grid cell.
   * @param cell The cell.
   * @return The probability
   */
  float getGridProbability(const TimerCountCell& cell) const
  {
    float odds = exp(cell.logOddsVal);
    return odds / (odds + 1.0f);
    /*
    float val = cell.logOddsVal;
    //prevent #IND when doing exp(large number).
    if (val > 50.0f) {
      return 1.0f;
    } else {
      float odds = exp(val);
      return odds / (odds + 1.0f);
    }
    */
    //return 0.5f;
  }
  void laser_updateSetOccupied(TimerCountCell& cell) const
  {
    if (cell.logOddsVal < 50.0f){
      // cell.logOddsVal += logOddsOccupied;
      cell.logOddsVal = 50.0f;
      cell.setTimeStamp(); //no need as called in other functions lucas
    }
  }
  void laser_updateSetFree(TimerCountCell& cell) const
  {
    // cell.logOddsVal += logOddsFree;
    cell.resetGridCell();
  }
  void setshifted(TimerCountCell& cell) const
  {
    cell.shifted = true;
  }
  void setUpdateFreeFactor(float factor)
  {
    logOddsFree = probToLogOdds(factor);
  }
  void setUpdateOccupiedFactor(float factor)
  {
    logOddsOccupied = probToLogOdds(factor);
  }
protected:
```

```
  float probToLogOdds(float prob)
  {
    float odds = prob / (1.0f - prob);
    return log(odds);
  }
  float logOddsOccupied; /// < The log odds representation of probability used for
    updating cells as occupied
  float logOddsFree;     /// < The log odds representation of probability used for
    updating cells as free
  // ros::Time  TimeStamp;//lucas
  // Eigen::Vector3f mapPose;//lucas
  // int mapPoseCount;
};

#endif
```

A.2.4 Timestamped grid map header

```
#ifndef __GridMapBase_h_
#define __GridMapBase_h_
#include <Eigen/Geometry>
#include <Eigen/LU>
#include "MapDimensionProperties.h"
#include "hector_icp/PLICP_Trans.h"//include custom msg type
#include "../util/matrix.h"
namespace hectorslam {
/**
 * GridMapBase provides basic grid map functionality (creates grid , provides
    transformation from/to world coordinates).
 * It serves as the base class for different map representations that may extend it's
    functionality.
 */
template<typename ConcreteCellType>
class GridMapBase
{
public:
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW
  /**
   * Indicates if given x and y are within map bounds
   * @return True if coordinates are within map bounds
   */
  bool hasGridValue(int x, int y) const
  {
    return (x >= 0) && (y >= 0) && (x < this->getSizeX()) && (y < this->getSizeY());
  }
  const Eigen::Vector2i& getMapDimensions() const { return mapDimensionProperties.
    getMapDimensions(); };
  int getSizeX() const { return mapDimensionProperties.getSizeX(); };
  int getSizeY() const { return mapDimensionProperties.getSizeY(); };
  bool pointOutOfMapBounds(const Eigen::Vector2f& pointMapCoords) const
  {
    return mapDimensionProperties.pointOutOfMapBounds(pointMapCoords);
  }
  virtual void reset()
  {
    this->clear();
  }
  /**
   * Resets the grid cell values by using the resetGridCell() function.
   */
  void clear()
  {
    int size = this->getSizeX() * this->getSizeY();
    for (int i = 0; i < size; ++i) {
      this->mapArray[i].resetGridCell();
    }
    //this->mapArray[0].set(1.0f);
    //this->mapArray[size-1].set(1.0f);
  }

  const MapDimensionProperties& getMapDimProperties() const { return
    mapDimensionProperties; };
  /**
   * Constructor, creates grid representation and transformations.
   */
```

```
GridMapBase(float mapResolution, const Eigen::Vector2i& size, const Eigen::Vector2f&
  offset)
  : mapArray(0)
  , lastUpdateIndex(-1)
{
  Eigen::Vector2i newMapDimensions (size);
  this->setMapGridSize(newMapDimensions);
  sizeX = size[0];
  setMapTransformation(offset, mapResolution);
  this->clear();
}
/**
 * Destructor
 */
virtual ~GridMapBase()
{
  deleteArray();
}
/**
 * Allocates memory for the two dimensional pointer array for map representation.
 */
void allocateArray(const Eigen::Vector2i& newMapDims)
{
  int sizeX = newMapDims.x();
  int sizeY = newMapDims.y();
  mapArray = new ConcreteCellType [sizeX*sizeY];
  mapDimensionProperties.setMapCellDims(newMapDims);
}
void deleteArray()
{
  if (mapArray != 0){
    delete[] mapArray;
    mapArray = 0;
    mapDimensionProperties.setMapCellDims(Eigen::Vector2i(-1,-1));
  }
}
ConcreteCellType& getCell(int x, int y)
{
  return mapArray[y * sizeX + x];
}
const ConcreteCellType& getCell(int x, int y) const
{
  return mapArray[y * sizeX + x];
}
ConcreteCellType& getCell(int index)
{
  return mapArray[index];
}
const ConcreteCellType& getCell(int index) const
{
  return mapArray[index];
}
void setMapGridSize(const Eigen::Vector2i& newMapDims)
{
  if (newMapDims != mapDimensionProperties.getMapDimensions() ){
   deleteArray();
   allocateArray(newMapDims);
   this->reset();
  }
}
/**
 * Copy Constructor, only needed if pointer members are present.
 */
GridMapBase(const GridMapBase& other)
{
  allocateArray(other.getMapDimensions());
  *this = other;
}
/**
 * Assignment operator, only needed if pointer members are present.
 */
GridMapBase& operator=(const GridMapBase& other)
{
  if ( !(this->mapDimensionProperties == other.mapDimensionProperties)){
    this->setMapGridSize(other.mapDimensionProperties.getMapDimensions());
  }
```

```
    this->mapDimensionProperties = other.mapDimensionProperties;
    this->worldTmap = other.worldTmap;
    this->mapTworld = other.mapTworld;
    this->worldTmap3D = other.worldTmap3D;
    this->scaleToMap = other.scaleToMap;
    //@todo potential resize
    int sizeX = this->getSizeX();
    int sizeY = this->getSizeY();
    size_t concreteCellSize = sizeof(ConcreteCellType);
    memcpy(this->mapArray, other.mapArray, sizeX*sizeY*concreteCellSize);
    return *this;
  }
  /**
   * Returns the world coordinates for the given map coords.
   */
  inline Eigen::Vector2f getWorldCoords(const Eigen::Vector2f& mapCoords) const
  {
    return worldTmap * mapCoords;
  }
  /**
   * Returns the map coordinates for the given world coords.
   */
  inline Eigen::Vector2f getMapCoords(const Eigen::Vector2f& worldCoords) const
  {
    return mapTworld * worldCoords;
  }
  /**
   * Returns the world pose for the given map pose.
   */
  inline Eigen::Vector3f getWorldCoordsPose(const Eigen::Vector3f& mapPose) const
  {
    Eigen::Vector2f worldCoords (worldTmap * mapPose.head<2>());
    return Eigen::Vector3f(worldCoords[0], worldCoords[1], mapPose[2]);
  }
  /**
   * Returns the map pose for the given world pose.
   */
  inline Eigen::Vector3f getMapCoordsPose(const Eigen::Vector3f& worldPose) const
  {
    Eigen::Vector2f mapCoords (mapTworld * worldPose.head<2>());
    return Eigen::Vector3f(mapCoords[0], mapCoords[1], worldPose[2]);
  }
  void setDimensionProperties(const Eigen::Vector2f& topLeftOffsetIn, const Eigen::
    Vector2i& mapDimensionsIn, float cellLengthIn)
  {
    setDimensionProperties(MapDimensionProperties(topLeftOffsetIn,mapDimensionsIn,
    cellLengthIn));
  }
  void setDimensionProperties(const MapDimensionProperties& newMapDimProps)
  {
    //Grid map cell number has changed
    if (!newMapDimProps.hasEqualDimensionProperties(this->mapDimensionProperties)){
      this->setMapGridSize(newMapDimProps.getMapDimensions());
    }
    //Grid map transformation/cell size has changed
    if(!newMapDimProps.hasEqualTransformationProperties(this->mapDimensionProperties)){
      this->setMapTransformation(newMapDimProps.getTopLeftOffset(), newMapDimProps.
    getCellLength());
    }
  }
  /**
   * Set the map transformations
   * @param xWorld The origin of the map coordinate system on the x axis in world
    coordinates
   * @param yWorld The origin of the map coordinate system on the y axis in world
    coordinates
   * @param The cell length of the grid map
   */
  void setMapTransformation(const Eigen::Vector2f& topLeftOffset, float cellLength)
  {
    mapDimensionProperties.setCellLength(cellLength);
    mapDimensionProperties.setTopLeftOffset(topLeftOffset);
    scaleToMap = 1.0f / cellLength;
    mapTworld = Eigen::AlignedScaling2f(scaleToMap, scaleToMap) * Eigen::Translation2f(
    topLeftOffset[0], topLeftOffset[1]);
    worldTmap3D = Eigen::AlignedScaling3f(scaleToMap, scaleToMap, 1.0f) * Eigen::
    Translation3f(topLeftOffset[0], topLeftOffset[1], 0);
    //std::cout << worldTmap3D.matrix() << std::endl;
```

```
    worldTmap3D = worldTmap3D.inverse();
    worldTmap = mapTworld.inverse();
  }

  /**
   * Returns the scale factor for one unit in world coords to one unit in map coords.
   * @return The scale factor
   */
  float getScaleToMap() const
  {
    return scaleToMap;
  }
  /**
   * Returns the cell edge length of grid cells in millimeters.
   * @return the cell edge length in millimeters.
   */
  float getCellLength() const
  {
    return mapDimensionProperties.getCellLength();
  }
  /**
   * Returns a reference to the homogenous 2D transform from map to world coordinates.
   * @return The homogenous 2D transform.
   */
  const Eigen::Affine2f& getWorldTmap() const
  {
    return worldTmap;
  }
  /**
   * Returns a reference to the homogenous 3D transform from map to world coordinates.
   * @return The homogenous 3D transform.
   */
  const Eigen::Affine3f& getWorldTmap3D() const
  {
    return worldTmap3D;
  }
  /**
   * Returns a reference to the homogenous 2D transform from world to map coordinates.
   * @return The homogenous 2D transform.
   */
  const Eigen::Affine2f& getMapTworld() const
  {
    return mapTworld;
  }
  void setUpdated() { lastUpdateIndex++; };
  int getUpdateIndex() const { return lastUpdateIndex; };
  /**
    * Returns the rectangle ([xMin,yMin],[xMax,xMax]) containing non-default cell values
    */
  bool getMapExtends(int& xMax, int& yMax, int& xMin, int& yMin) const
  {
    int lowerStart = -1;
    int upperStart = 10000;
    int xMaxTemp = lowerStart;
    int yMaxTemp = lowerStart;
    int xMinTemp = upperStart;
    int yMinTemp = upperStart;
    int sizeX = this->getSizeX();
    int sizeY = this->getSizeY();
    for (int x = 0; x < sizeX; ++x) {
      for (int y = 0; y < sizeY; ++y) {
        if (this->mapArray[x][y].getValue() != 0.0f) {
          if (x > xMaxTemp) {
            xMaxTemp = x;
          }
          if (x < xMinTemp) {
            xMinTemp = x;
          }
          if (y > yMaxTemp) {
            yMaxTemp = y;
          }
          if (y < yMinTemp) {
            yMinTemp = y;
          }
        }
      }
    }
    if ((xMaxTemp != lowerStart) &&
        (yMaxTemp != lowerStart) &&
```

```
        (xMinTemp != upperStart) &&
        (yMinTemp != upperStart)) {
      xMax = xMaxTemp;
      yMax = yMaxTemp;
      xMin = xMinTemp;
      yMin = yMinTemp;
      return true;
    } else {
      return false;
    }
  }
  void mapTrans(hector_icp::PLICP_Trans Trans, int *SetOccupie, int *SetFree, bool
    first_trans, Eigen::Vector3f *lastpose, float p_map_resolution_) const//lucas
  {
    const int dim = 2;//setting demenional
    float val_Rm[dim*dim];
    float val_tm[dim];
    for (int i = 0; i < 4; ++i){
      val_Rm[i] = Trans.rotation.data[i];
      // val_Rm[i] = 0.0f;
    }
    for (int i = 0; i < 2; ++i){
      // val_tm[i] = mapTrans[i];
      val_tm[i] = Trans.transform.data[i];
      // val_tm[i] = 0.0f;
    }
    Eigen::Vector2f mass_center(Trans.transform.data[2],Trans.transform.data[3]);
    Matrix R(dim, dim, val_Rm);
    Matrix t(dim, 1, val_tm);
    int SizeX = this->getSizeX();
    int SizeY = this->getSizeY();
    int size = SizeX * SizeY;
    int *rotated_map;
    rotated_map = new int [size*dim];
    int count = 1;
    if(first_trans){
      ROS_INFO("###############Initial Map Trans Received\n");
      for (int x = 0; x < SizeX; ++x) {
        for (int y = 0; y < SizeY; ++y) {
          if (this->mapArray[x*SizeY + y].isOccupied()) {
          // if(true){
              rotated_map[dim * count + 0] = x;
              rotated_map[dim * count + 1] = y;
              // rotated_map[dim * count + 2] = z;
              SetFree[count] = x*SizeY + y; //Optimization
              count++;
          }
        }
      }
      SetFree[0] = count;
      ROS_INFO("###############Initial Map Trans Finished\n");
    }
    else{
      Eigen::Vector2f MapZero = getMapCoords(mass_center);
      ros::Duration d(0.1);//Durations can be negative. for test, processing time unknown
      // for (int x = 0; x < SizeX; ++x) {
      //   for (int y = 0; y < SizeY; ++y) {
      //     if (this->mapArray[x*SizeY + y].isOccupied()
      //         && !this->mapArray[x*SizeY + y].isshifted()
      //         && Trans.begin_stamp - d <= this->mapArray[x*SizeY + y].getTimeStamp()
      //         && this->mapArray[x*SizeY + y].getTimeStamp() <= Trans.end_stamp + d) {
      //         rotated_map[dim * count + 0] = x;
      //         rotated_map[dim * count + 1] = y;
      //         // rotated_map[dim * count + 2] = z;
      //         SetFree[count] = x*SizeY + y; //Optimization
      //         count++;
      //     }
      //   }
      // }
      float zoom = (SizeX/200)/p_map_resolution_;
      int x_zoom_low, x_zoom_high, y_zoom_low, y_zoom_high;
      MapZero[1]-zoom > 0 ? x_zoom_low = MapZero[1]-zoom : x_zoom_low = 0;
      MapZero[0]-zoom > 0 ? y_zoom_low = MapZero[0]-zoom : y_zoom_low = 0;
      MapZero[1]+zoom > SizeX ? x_zoom_high = SizeX : x_zoom_high = MapZero[1]+zoom;
      MapZero[0]+zoom > SizeY ? y_zoom_high = SizeY : y_zoom_high = MapZero[0]+zoom;
      ROS_INFO("Map size is %i X %i; ZOOM box is  X_l %i, X_h %i, Y_l %i, Y_h %i", SizeX,
     SizeY, x_zoom_low, x_zoom_high, y_zoom_low, y_zoom_high);
```

```
      for (int x = x_zoom_low; x < x_zoom_high; ++x) {
        for (int y = y_zoom_low; y < y_zoom_high; ++y) {
          if (this->mapArray[x*SizeY + y].isOccupied()
              && !this->mapArray[x*SizeY + y].isshifted()//skip sifited cell
              && Trans.begin_stamp - d <= this->mapArray[x*SizeY + y].getTimeStamp()
              && this->mapArray[x*SizeY + y].getTimeStamp() <= Trans.end_stamp + d) {
              rotated_map[dim * count + 0] = x;
              rotated_map[dim * count + 1] = y;
              // rotated_map[dim * count + 2] = z;
              SetFree[count] = x*SizeY + y; //Optimization
              count++;
          }
        }
      }
      SetFree[0] = count;//use first cell to store list size
    }
    // R = Matrix::eye(2);
    rotatePose(lastpose, R, t);
    rotatePoint(rotated_map, count, R, t, dim, p_map_resolution_, mass_center);
    int cc = 0;
    while (count > 0){
      if (int(rotated_map[dim * count + 0]) <= SizeX
          && int(rotated_map[dim * count + 1]) <= SizeY){
        SetOccupie[cc] = int(rotated_map[dim * count + 0])*SizeY
        + int(rotated_map[dim * count + 1]); //Optimization
        cc ++;
      }
      count--;
    }
    SetOccupie[0] = cc;//use first cell to store list size
    // ROS_INFO("Current Pose: X %f, Y %f, Z %f,", lastpose[0][0], lastpose[0][1], lastpose[0][2]);
    // *lastpose = Eigen::Vector3f(lastpose[0][0]+1, lastpose[0][1]+1, 3.14);
    // for (size_t i = 0; i < size; i++) {
    //   this->mapArray[i] = waitList[i];
    //   // waitList[x*this->getSizeY() + y].resetGridCell();
    //   // std::cout << "Cell merged\n";
    // }
    // ROS_INFO("**********************************************Go BACK\n");
  }
  void rotatePose(Eigen::Vector3f* lastpose, Matrix Rr, Matrix t) const{
    ROS_INFO("R: X %f, Y %f, Z %f",  lastpose[0][0], lastpose[0][1], lastpose[0][2]);
    ROS_INFO("t: X %f, Y %f",  t.val[0][0], t.val[0][1]);
    // Eigen::Vector3f WorldPose(this->getWorldCoordsPose(*lastpose));
    FLOAT *val = new FLOAT[2];
    val[0] = lastpose[0][0];
    val[1] = lastpose[0][1];
    Matrix point(2, 1, val);
    // Matrix Rr = Matrix::inv(R);//for some reason the coordinate sys for map and world are inversed--need to check with paper/ROS
    Matrix pointout = Rr * point + t;
    // ROS_INFO("Orientation R1 %f, R2 %f, T %f",  R.val[1][1], R.val[0][1], lastpose[0][2]);
    FLOAT theta = atan2(Rr.val[1][0] , Rr.val[0][0]) + lastpose[0][2];
    *lastpose = Eigen::Vector3f(pointout.val[0][0], pointout.val[1][0], theta);
    ROS_INFO("Current Pose: X %f, Y %f, Z %f,", lastpose[0][0], lastpose[0][1], lastpose[0][2]);
  }
  void rotatePoint(int *&T, int num2, Matrix Rr, Matrix t, int32_t dim, float p_map_resolution_, Eigen::Vector2f& mass_center) const{
    ROS_INFO("Translation2m: X %f, Y %f", t.val[0][0], t.val[0][1]);
    // Eigen::Vector2f map_center = getMapCoords(0,0);
    // Eigen::Vector2f MapZero = getMapCoords(mass_center);
    // Eigen::Vector2f MapZero;
    // MapZero[0] = 1024 + ceil(mass_center[1]/p_map_resolution_);
    // MapZero[1] = 1024 + ceil(mass_center[0]/p_map_resolution_);
    // ROS_INFO("MassZero in world: X %f, Y %f, MassZero on map: X %f, Y %f, ", mass_center[0], mass_center[1], MapZero[0], MapZero[1]);
    for (int i = 0; i < num2; ++i) {
      FLOAT *val = new FLOAT[dim];
      Eigen::Vector2f mapCoords((T[dim * i + 1]), (T[dim * i + 0]));
      // ROS_INFO("Coord2mIn: X %f, Y %f", mapCoords[0], mapCoords[1]);
      // transfer from grid number to world coords
      Eigen::Vector2f worldCoords = getWorldCoords(mapCoords);
      val[0] = worldCoords[0];
      val[1] = worldCoords[1];
```

```
      // val[0] = mapCoords[1] - MapZero[1];
      // val[1] = mapCoords[0] - MapZero[0];
      // ROS_INFO("pointin: X %f, Y %f", val[0], val[1]);
      Matrix point(2, 1, val);
      // Matrix Rr = Matrix::inv(R);
      Matrix pointout = Rr * point + t;
      Eigen::Vector2f afterWorldCoords(pointout.val[0][0], pointout.val[1][0]);
      // Eigen::Vector2f afterWorldCoords(pointout.val[0][0]+MapZero[1], pointout.val[0][1]+MapZero[0]);
      // ROS_INFO("worldCoords: X %f, Y %f, afterWorldCoords: X %f, Y %f,", worldCoords[0], worldCoords[1], pointout.val[0][0], pointout.val[1][0]);
      // ROS_INFO("pointout: X %f, Y %f", afterWorldCoords[0], afterWorldCoords[1]);
      mapCoords = getMapCoords(afterWorldCoords);
      T[dim * i + 0] = mapCoords[1];
      T[dim * i + 1] = mapCoords[0];
      // T[dim * i + 0] = ceil(afterWorldCoords[1]);
      // T[dim * i + 1] = ceil(afterWorldCoords[0]);
      // ROS_INFO("Coord2mOUT: X %f, Y %f", mapCoords[0], mapCoords[1]);
    }
  }
protected:
  ConcreteCellType *mapArray;    ///< Map representation used with plain pointer array.
  float scaleToMap;              ///< Scaling factor from world to map.
  Eigen::Affine2f worldTmap;     ///< Homogenous 2D transform from map to world coordinates.
  Eigen::Affine3f worldTmap3D;   ///< Homogenous 3D transform from map to world coordinates.
  Eigen::Affine2f mapTworld;     ///< Homogenous 2D transform from world to map coordinates.
  MapDimensionProperties mapDimensionProperties;
  int sizeX;
private:
  int lastUpdateIndex;
};
}
#endif
```

A.2.5 Map container

```
#ifndef __OccGridMapBase_h_
#define __OccGridMapBase_h_
#include "GridMapBase.h"
#include "../scan/DataPointContainer.h"
#include "../util/UtilFunctions.h"
#include <Eigen/Geometry>
namespace hectorslam {
template<typename ConcreteCellType, typename ConcreteGridFunctions>
class OccGridMapBase
  : public GridMapBase<ConcreteCellType>
{
public:
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW
  OccGridMapBase(float mapResolution, const Eigen::Vector2i& size, const Eigen::Vector2f& offset)
    : GridMapBase<ConcreteCellType>(mapResolution, size, offset)
    , currUpdateIndex(0)
    , currMarkOccIndex(-1)
    , currMarkFreeIndex(-1)
  {}
  virtual ~OccGridMapBase() {}
  void updateSetOccupied(int index)
  {
    concreteGridFunctions.updateSetOccupied(this->getCell(index));
  }
  void updateSetFree(int index)
  {
    concreteGridFunctions.updateSetFree(this->getCell(index));
  }
  void updateTimeStamp(int index)//lucas
  {
    concreteGridFunctions.updateTimeStamp(this->getCell(index));
  }
  void updateUnsetFree(int index)
  {
    concreteGridFunctions.updateUnsetFree(this->getCell(index));
  }
```

```
float getGridProbabilityMap(int index) const
{
  return concreteGridFunctions.getGridProbability(this->getCell(index));
}
// float getCellTimeStamp(int index) const//lucas
// {
//   return concreteGridFunctions.updateTimeStamp(this->getCell(index));
// }
bool isOccupied(int xMap, int yMap) const
{
  return (this->getCell(xMap,yMap).isOccupied());
}
bool isFree(int xMap, int yMap) const
{
  return (this->getCell(xMap,yMap).isFree());
}
bool isOccupied(int index) const
{
  return (this->getCell(index).isOccupied());
}
bool isFree(int index) const
{
  return (this->getCell(index).isFree());
}
float getObstacleThreshold() const
{
  ConcreteCellType temp;
  temp.resetGridCell();
  return concreteGridFunctions.getGridProbability(temp);
}
void setUpdateFreeFactor(float factor)
{
  concreteGridFunctions.setUpdateFreeFactor(factor);
}
void setUpdateOccupiedFactor(float factor)
{
  concreteGridFunctions.setUpdateOccupiedFactor(factor);
}
/**
 * Updates the map using the given scan data and robot pose
 * @param dataContainer Contains the laser scan data
 * @param robotPoseWorld The 2D robot pose in world coordinates
 */
void updateByScan(const DataContainer& dataContainer, const Eigen::Vector3f&
  robotPoseWorld)
{
  // ROS_INFO("Current robotPoseWorld: X %f, Y %f, Z %f,", robotPoseWorld[0],
  robotPoseWorld[1], robotPoseWorld[2]);
  currMarkFreeIndex = currUpdateIndex + 1;
  currMarkOccIndex = currUpdateIndex + 2;
  //Get pose in map coordinates from pose in world coordinates
  Eigen::Vector3f mapPose(this->getMapCoordsPose(robotPoseWorld));
  //Get a 2D homogenous transform that can be left-multiplied to a robot coordinates
  vector to get world coordinates of that vector
  Eigen::Affine2f poseTransform((Eigen::Translation2f(
                                   mapPose[0], mapPose[1]) * Eigen::Rotation2Df(
  mapPose[2])));
  //Get start point of all laser beams in map coordinates (same for alle beams, stored
  in robot coords in dataContainer)
  Eigen::Vector2f scanBeginMapf(poseTransform * dataContainer.getOrigo());
  //Get integer vector of laser beams start point
  Eigen::Vector2i scanBeginMapi(scanBeginMapf[0] + 0.5f, scanBeginMapf[1] + 0.5f);
  //Get number of valid beams in current scan
  int numValidElems = dataContainer.getSize();
  //std::cout << "\n maxD: " << maxDist << " num: " << numValidElems << "\n";
  //Iterate over all valid laser beams
  for (int i = 0; i < numValidElems; ++i) {
    //Get map coordinates of current beam endpoint
    Eigen::Vector2f scanEndMapf(poseTransform * (dataContainer.getVecEntry(i)));
    //std::cout << "\ns\n" << scanEndMapf << "\n";
    //add 0.5 to beam endpoint vector for following integer cast (to round, not
  truncate)
    scanEndMapf.array() += (0.5f);
    //Get integer map coordinates of current beam endpoint
    Eigen::Vector2i scanEndMapi(scanEndMapf.cast<int>());
    //Update map using a bresenham variant for drawing a line from beam start to beam
  endpoint in map coordinates
```

```
      if (scanBeginMapi != scanEndMapi){
        updateLineBresenhami(scanBeginMapi, scanEndMapi);
      }
    }
    //Tell the map that it has been updated
    this->setUpdated();
    //Increase update index (used for updating grid cells only once per incoming scan)
    currUpdateIndex += 3;
  }
  inline void updateLineBresenhami( const Eigen::Vector2i& beginMap, const Eigen::
    Vector2i& endMap, unsigned int max_length = UINT_MAX){
    int x0 = beginMap[0];
    int y0 = beginMap[1];
    //check if beam start point is inside map, cancel update if this is not the case
    if ((x0 < 0) || (x0 >= this->getSizeX()) || (y0 < 0) || (y0 >= this->getSizeY())) {
      return;
    }
    int x1 = endMap[0];
    int y1 = endMap[1];
    //std::cout << " x: "<< x1 << " y: " << y1 << " length: " << length << "     ";
    //check if beam end point is inside map, cancel update if this is not the case
    if ((x1 < 0) || (x1 >= this->getSizeX()) || (y1 < 0) || (y1 >= this->getSizeY())) {
      return;
    }
    int dx = x1 - x0;
    int dy = y1 - y0;
    unsigned int abs_dx = abs(dx);
    unsigned int abs_dy = abs(dy);
    int offset_dx = util::sign(dx);
    int offset_dy = util::sign(dy) * this->sizeX;
    unsigned int startOffset = beginMap.y() * this->sizeX + beginMap.x();
    //if x is dominant
    if(abs_dx >= abs_dy){
      int error_y = abs_dx / 2;
      bresenham2D(abs_dx, abs_dy, error_y, offset_dx, offset_dy, startOffset);
    }else{
      //otherwise y is dominant
      int error_x = abs_dy / 2;
      bresenham2D(abs_dy, abs_dx, error_x, offset_dy, offset_dx, startOffset);
    }
    unsigned int endOffset = endMap.y() * this->sizeX + endMap.x();
    this->bresenhamCellOcc(endOffset);//lucas
  }
  inline void bresenhamCellFree(unsigned int offset)
  {
    ConcreteCellType& cell (this->getCell(offset));
    if (cell.updateIndex < currMarkFreeIndex) {
      concreteGridFunctions.updateSetFree(cell);
      cell.updateIndex = currMarkFreeIndex;
    }
  }
  inline void bresenhamCellOcc(unsigned int offset)
  {
    ConcreteCellType& cell (this->getCell(offset));
    if (cell.updateIndex < currMarkOccIndex) {
      //if this cell has been updated as free in the current iteration, revert this
      if (cell.updateIndex == currMarkFreeIndex) {
        concreteGridFunctions.updateUnsetFree(cell);
        concreteGridFunctions.updateTimeStamp(cell);// mappose removed, not sure why here
     lucas
      }
      concreteGridFunctions.updateSetOccupied(cell);
      concreteGridFunctions.updateTimeStamp(cell);//lucas
      //std::cout << " setOcc " << "\n";
      cell.updateIndex = currMarkOccIndex;
    }
  }
  inline void bresenham2D( unsigned int abs_da, unsigned int abs_db, int error_b, int
    offset_a, int offset_b, unsigned int offset){
    this->bresenhamCellFree(offset);
    unsigned int end = abs_da-1;
    for(unsigned int i = 0; i < end; ++i){
      offset += offset_a;
      error_b += abs_db;
      if((unsigned int)error_b >= abs_da){
        offset += offset_b;
        error_b -= abs_da;
```

```
      }
      this->bresenhamCellFree(offset);
    }
  }
  inline void updateByLaser(int *SetOccupie, int *SetFree, Eigen::Vector3f *lastpose){//
    not sure why inline will make this unstable
    for (int x = 1; x < SetOccupie[0]; ++x){//skip 1st. its used for passing arry length
      this->laserICPCellOcc(SetOccupie[x]);
      // p++;
    }
    for (int x = 1; x < SetFree[0]; ++x){//skip 1st. its used for passing arry length
      this->laserICPCellFree(SetFree[x]);
      // p++;
    }
    ROS_INFO("Map: %i cells Transed, %i cell Freed", SetOccupie[0], SetFree[0]);
    // ROS_INFO("Map: %i cells Transed", p);
    // ROS_INFO("Current Pose: X %f, Y %f, Z %f,", lastpose[0][0], lastpose[0][1],
    lastpose[0][2]);
  }
  inline void laserICPCellOcc(unsigned int offset)
  {
    ConcreteCellType& cell (this->getCell(offset));
    concreteGridFunctions.laser_updateSetOccupied(cell);
    concreteGridFunctions.setshifted(cell);
    // ROS_INFO("Map Editing");
    // concreteGridFunctions.laser_updateSetFree(cell);// mappose removed, not sure why
    here lucas
    // cell.updateIndex = currMarkOccIndex;
  }
  inline void laserICPCellFree(unsigned int offset)
  {
    ConcreteCellType& cell (this->getCell(offset));
    if (!cell.isshifted()){//if this cell not touched by my ICP
      concreteGridFunctions.laser_updateSetFree(cell);
    }
    // concreteGridFunctions.laser_updateSetFree(cell);
    // ROS_INFO("Map Editing");
    // concreteGridFunctions.laser_updateSetFree(cell);// mappose removed, not sure why
    here lucas
    // cell.updateIndex = currMarkOccIndex;
  }
protected:
  ConcreteGridFunctions concreteGridFunctions;
  int currUpdateIndex;
  int currMarkOccIndex;
  int currMarkFreeIndex;
};

}
#endif
```

A.2.6 Map class

```
#ifndef __GridMap_h_
#define __GridMap_h_
#include "OccGridMapBase.h"
#include "GridMapLogOdds.h"
#include "GridMapReflectanceCount.h"
#include "GridMapSimpleCount.h"
#include "GridMapTimerCount.h"
namespace hectorslam {
// typedef OccGridMapBase<LogOddsCell, GridMapLogOddsFunctions> GridMap;
//typedef OccGridMapBase<SimpleCountCell, GridMapSimpleCountFunctions> GridMap;
//typedef OccGridMapBase<ReflectanceCell, GridMapReflectanceFunctions> GridMap;
typedef OccGridMapBase<TimerCountCell, GridMapTimerCountFunctions> GridMap;//
    lucasModified
}
#endif
```

A.2.7 Multi-level map class

```
#ifndef _hectormaprepmultimap_h__
#define _hectormaprepmultimap_h__
```

```
#include "MapRepresentationInterface.h"
#include "MapProcContainer.h"
#include "../map/GridMap.h"
#include "../map/OccGridMapUtilConfig.h"
#include "../matcher/ScanMatcher.h"
#include "../util/DrawInterface.h"
#include "../util/HectorDebugInfoInterface.h"
namespace hectorslam{
class MapRepMultiMap : public MapRepresentationInterface
{
public:
  MapRepMultiMap(float mapResolution, int mapSizeX, int mapSizeY, unsigned int numDepth, const Eigen::Vector2f& startCoords, DrawInterface* drawInterfaceIn, HectorDebugInfoInterface* debugInterfaceIn)
  {
    //unsigned int numDepth = 3;
    Eigen::Vector2i resolution(mapSizeX, mapSizeY);
    float totalMapSizeX = mapResolution * static_cast<float>(mapSizeX);
    float mid_offset_x = totalMapSizeX * startCoords.x();
    float totalMapSizeY = mapResolution * static_cast<float>(mapSizeY);
    float mid_offset_y = totalMapSizeY * startCoords.y();
    for (unsigned int i = 0; i < numDepth; ++i){
      std::cout << "HectorSM map lvl " << i << ": cellLength: " << mapResolution << " res x:" << resolution.x() << " res y: " << resolution.y() << "\n";
      GridMap* gridMap = new hectorslam::GridMap(mapResolution,resolution, Eigen::Vector2f(mid_offset_x, mid_offset_y));
      OccGridMapUtilConfig<GridMap>* gridMapUtil = new OccGridMapUtilConfig<GridMap>(gridMap);
      ScanMatcher<OccGridMapUtilConfig<GridMap> >* scanMatcher = new hectorslam::ScanMatcher<OccGridMapUtilConfig<GridMap> >(drawInterfaceIn, debugInterfaceIn);
      mapContainer.push_back(MapProcContainer(gridMap, gridMapUtil, scanMatcher));
      resolution /= 2;
      mapResolution*=2.0f;
    }
    dataContainers.resize(numDepth-1);
  }
  virtual ~MapRepMultiMap()
  {
    unsigned int size = mapContainer.size();
    for (unsigned int i = 0; i < size; ++i){
      mapContainer[i].cleanup();
    }
  }
  virtual void reset()
  {
    unsigned int size = mapContainer.size();
    for (unsigned int i = 0; i < size; ++i){
      mapContainer[i].reset();
    }
  }
  virtual float getScaleToMap() const { return mapContainer[0].getScaleToMap(); };
  virtual int getMapLevels() const { return mapContainer.size(); };
  virtual const GridMap& getGridMap(int mapLevel) const { return mapContainer[mapLevel].getGridMap(); };
  virtual void addMapMutex(int i, MapLockerInterface* mapMutex)
  {
    mapContainer[i].addMapMutex(mapMutex);
  }
  MapLockerInterface* getMapMutex(int i)
  {
    return mapContainer[i].getMapMutex();
  }
  virtual void onMapUpdated()
  {
    unsigned int size = mapContainer.size();
    for (unsigned int i = 0; i < size; ++i){
      mapContainer[i].resetCachedData();
    }
  }
  virtual Eigen::Vector3f matchData(const Eigen::Vector3f& beginEstimateWorld, const DataContainer& dataContainer, Eigen::Matrix3f& covMatrix)
  {
    size_t size = mapContainer.size();
    Eigen::Vector3f tmp(beginEstimateWorld);
    for (int index = size - 1; index >= 0; --index){
      //std::cout << " m " << i;
```

```
      if (index == 0){
        tmp  = (mapContainer[index].matchData(tmp, dataContainer, covMatrix, 5));
      }else{
        dataContainers[index-1].setFrom(dataContainer, static_cast<float>(1.0 / pow(2.0, static_cast<double>(index))));
        tmp  = (mapContainer[index].matchData(tmp, dataContainers[index-1], covMatrix, 3));
      }
    }
    return tmp;
  }
  virtual void updateByScan(const DataContainer& dataContainer, const Eigen::Vector3f& robotPoseWorld)
  {
    unsigned int size = mapContainer.size();
    for (unsigned int i = 0; i < size; ++i){
      //std::cout << " u " <<  i;
      if (i==0){
        mapContainer[i].updateByScan(dataContainer, robotPoseWorld);
      }else{
        mapContainer[i].updateByScan(dataContainers[i-1], robotPoseWorld);
      }
    }
    //std::cout << "\n";
  }
  virtual void updateByLaser(int *SetOccupie, int *SetFree, Eigen::Vector3f *lastpose)// lucas this is getting wild, so many places need to be modified
  {
    unsigned int size = mapContainer.size();
    for (unsigned int i = 0; i < size; ++i){
      //std::cout << " u " <<  i;
      if (i==0){
        mapContainer[i].updateByLaser(SetOccupie, SetFree, lastpose);
      }else{
        // mapContainer[i].updateByLaser(SetOccupie, SetFree, lastpose);
      }
    }
    //std::cout << "\n";
  }
  virtual void setUpdateFactorFree(float free_factor)
  {
    size_t size = mapContainer.size();
    for (unsigned int i = 0; i < size; ++i){
      GridMap& map = mapContainer[i].getGridMap();
      map.setUpdateFreeFactor(free_factor);
    }
  }
  virtual void setUpdateFactorOccupied(float occupied_factor)
  {
    size_t size = mapContainer.size();
    for (unsigned int i = 0; i < size; ++i){
      GridMap& map = mapContainer[i].getGridMap();
      map.setUpdateOccupiedFactor(occupied_factor);
    }
  }
protected:
  std::vector<MapProcContainer> mapContainer;
  std::vector<DataContainer> dataContainers;
};
}
#endif
```

A.2.8 Multi-level map interface

```
#ifndef _hectormaprepresentationinterface_h__
#define _hectormaprepresentationinterface_h__
class GridMap;
class ConcreteOccGridMapUtil;
class DataContainer;
namespace hectorslam{
class MapRepresentationInterface
{
public:
  virtual ~MapRepresentationInterface() {};
  virtual void reset() = 0;
```

```
  virtual float getScaleToMap() const = 0;
  virtual int getMapLevels() const = 0;
  virtual const GridMap& getGridMap(int mapLevel = 0) const = 0;
  virtual void addMapMutex(int i, MapLockerInterface* mapMutex) = 0;
  virtual MapLockerInterface* getMapMutex(int i) = 0;
  virtual void onMapUpdated() = 0;
  virtual Eigen::Vector3f matchData(const Eigen::Vector3f& beginEstimateWorld, const
    DataContainer& dataContainer, Eigen::Matrix3f& covMatrix) = 0;
  virtual void updateByScan(const DataContainer& dataContainer, const Eigen::Vector3f&
    robotPoseWorld) = 0;
  virtual void updateByLaser(int *SetOccupie, int *SetFree, Eigen::Vector3f *lastpose) =
    0;// lucas modified
  virtual void setUpdateFactorFree(float free_factor) = 0;
  virtual void setUpdateFactorOccupied(float occupied_factor) = 0;
};
}
#endif
```

A.3 LOAM ITM Codes

This section documented the source code of modified LOAM SLAM described in Chapter 4. The major improvements are focused on Slerp-based system state updates using ITM algorithm.

A.3.1 GPS and laser tracker receiver

```
//This code is created based on A-LOAM by Weichen WEI: weichen.wei@monash.edu
#include <iostream>
#include <ros/ros.h>
#include <tf2_ros/transform_broadcaster.h>
#include <tf2/transform_datatypes.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2/LinearMath/Matrix3x3.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2/convert.h>
#include <math.h>
#include "std_msgs/String.h"
#include <nav_msgs/Path.h>
#include <nav_msgs/Odometry.h>
#include <sensor_msgs/NavSatFix.h>
#include <gps_common/GPSFix.h>
#include <gps_common/conversions.h>
#include <pcl_ros/point_cloud.h>
#include <geometry_msgs/Vector3Stamped.h>
#include <geometry_msgs/Point.h>
#include <geometry_msgs/Transform.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <ceres/ceres.h>
#include <Eigen/Core>
#include <Eigen/Geometry>
#include <chrono>
#include "loam_itm/rotation.h"
using namespace std;
class icp_iter_class
{
public:
  icp_iter_class()
  {
    std::remove("Hector.txt");
    std::remove("Tracker.txt");
    std::remove("Icp.txt");
    std::remove("TrackerOrig.txt");
    n_.param<int>("sampling_round", p_sampling_round, 0);
    n_.param<int>("sampling_num", p_sampling_num, 100);
    n_.param<int>("iter_cycle", p_iter_cycle, 5);
    n_.param<float>("gap_limit", p_gap_limit, 0.0f);
```

```
    n_.param<float>("angle_limit", p_angle_limit, 0.02f);
    n_.param<int>("residual_limit", p_residual_limit, 300);
    reference_location_init = 0;
    laser_count = 0;
    hactor_count = 0;
    laser_offset = 0;
    hactor_offset = 0;
    laser_stamp_min.fromSec(0.0);
    laser_stamp_max.fromSec(0.0);
    first_call_escaper = 0;
    orientation_aline_escaper = 0;
    gps_location_init = false;
    path_flag = 1;
    min_residual = 10.00;
    round_hector_points = 0;
    round_tracker_points = 0;
    sub_ = n_.subscribe("/aft_mapped_path", 50, &icp_iter_class::hector_path_Callback,
    this);
    sub_kitti = n_.subscribe("/path_gt", 50, &icp_iter_class::kitti_path_Callback, this);
    timer = n_.createTimer(ros::Duration(p_iter_cycle), &icp_iter_class::icp_iter, this);
    icp_path_pub = n_.advertise<pcl::PointCloud<pcl::PointXYZ>>("icp_path", 1);
    align_path_pub = n_.advertise<pcl::PointCloud<pcl::PointXYZ>>("align_path", 1);
    hector_path_pub = n_.advertise<pcl::PointCloud<pcl::PointXYZ>>("Hector_path", 1);
    tracker_path_pub = n_.advertise<pcl::PointCloud<pcl::PointXYZ>>("Tracker_path", 1);
    PCL_Trans_pub = n_.advertise<geometry_msgs::TransformStamped>("PCL_Trans", 1);
    gps_pub = n_.advertise<geometry_msgs::PoseStamped>("GPS_path", 1);
  }
  struct cost_function_define
  {
    cost_function_define(Eigen::Vector3d p1, Eigen::Vector3d p2) : _p1(p1), _p2(p2) {}
    template <typename T>
    bool operator()(const T *const cere_r, T *residual) const
    {
      T p_1[3];
      T p_2[3];
      p_1[0] = T(_p1.x());
      p_1[1] = T(_p1.y());
      p_1[2] = T(_p1.z());
      AngleAxisRotatePoint(cere_r, p_1, p_2);
      p_2[0] += cere_r[3];
      p_2[1] += cere_r[4];
      p_2[2] += cere_r[5];
      T p_3[3];
      p_3[0] = T(_p2.x());
      p_3[1] = T(_p2.y());
      p_3[2] = T(_p2.z());
      residual[0] = p_2[0] - p_3[0];
      residual[1] = p_2[1] - p_3[1];
      residual[2] = p_2[2] - p_3[2];
      return true;
    }
    Eigen::Vector3d _p1, _p2;
  };
  void icp_iter(const ros::TimerEvent &)
  {
    if (first_call_escaper < 2)
    {
      ROS_INFO("Skip 5 sec %d", first_call_escaper);
      first_call_escaper++;
      ROS_INFO("pre-sampleling in %d rounds, every round with %d points, Iter every %d
    Seconds. ITM tigger if gap bigger than %f meters.", p_sampling_round, p_sampling_num,
     p_iter_cycle, p_gap_limit);
    }
    else
    {

      int dim = 3;

      int i = 0;
      int j = 0;
      double *M = (double *)calloc(3 * hactor_count, sizeof(double));
      double *T = (double *)calloc(3 * laser_count, sizeof(double));
      double *ceres_rot = new double[6];
      Eigen::Quaterniond R;
      Eigen::Vector3d t;
      Eigen::Vector3d mass_center(dim, 1);
      pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Tracker(new pcl::PointCloud<pcl::PointXYZ
    >);
```

```
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Hector(new pcl::PointCloud<pcl::PointXYZ
  >);
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_ICP(new pcl::PointCloud<pcl::PointXYZ>);
    if (laser_count >= path_flag * p_sampling_num && path_flag <= p_sampling_round)
    {
      double residual = pclAssembler(cloud_Hector, cloud_Tracker, i, j, M, T, R, t, dim
  , mass_center, ceres_rot);

      min_residual = residual;
      ROS_INFO("###################### Alineing Reference Frame ######################
  \n");
      path_rotation = R;
      path_translation = t;
      rotatePoint(T, j, R, t, dim, ceres_rot);
      ITMEncoder(cloud_ICP, T, j, dim);
      if (residual < min_residual)
      {
        align_path_pub.publish(cloud_ICP);
      }
      else
      {
        ROS_INFO("\n######Residual not valid, ICP has not converged.###### %f",
  residual);
      }



      path_flag++;
    }
    else if (laser_count >= path_flag * p_sampling_num && path_flag > p_sampling_round)
    {
      double residual = pclAssembler(cloud_Hector, cloud_Tracker, i, j, M, T, R, t, dim
  , mass_center, ceres_rot);

      rotatePoint(M, i, R, t, dim, ceres_rot);
      ITMEncoder(cloud_ICP, M, i, dim);
      if (residual < p_residual_limit)
      {
        ROS_INFO("###################### Sending Rotation Matrix #####################
   \n");
        ROS_INFO("Residual is: %f", residual);



        ROS_INFO("Rotation Mass Center: X %f, Y %f, Z %f", mass_center.x(), mass_center
  .y(), mass_center.z());


        if ((abs(t.x()) + abs(t.y()) + abs(t.z())) > p_gap_limit)
        {
          icp_path_pub.publish(cloud_ICP);
          ITMpub(laser_stamp_min, laser_stamp_max, ceres_rot, mass_center, dim);
        }
        else
        {
          align_path_pub.publish(cloud_ICP);
          ROS_INFO("Displance or Rotation is too Small, No need to Correct");
        }
      }
      else
      {
        ROS_INFO("######Residual not valid, ICP has not converged.######");
        align_path_pub.publish(cloud_ICP);
      }
      path_flag++;
    }
    else
    {
      ROS_INFO("waiting for more Reference points");
    }
    free(M);
    free(T);
  }
}
double pclAssembler(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Hector, pcl::PointCloud<
  pcl::PointXYZ>::Ptr cloud_Tracker, int &i, int &j, double *&M, double *&T, Eigen::
  Quaterniond &R, Eigen::Vector3d &t, int dim, Eigen::Vector3d &mass_center, double *
  ceres_rot)
```

```
{
  pointCloudResize(cloud_Hector);
  pointCloudResize(cloud_Tracker);
  round_hector_points = hactor_count - hactor_offset;
  round_tracker_points = laser_count - laser_offset;
  i = hectorEncoder(cloud_Hector, M, i, dim, mass_center);
  j = kittiEncoder(cloud_Tracker, T, j, dim);
  hector_path_pub.publish(cloud_Hector);
  tracker_path_pub.publish(cloud_Tracker);
  double residual = 0.0;
  if (path_flag <= p_sampling_round)
  {

  }
  else
  {
    ROS_INFO("Trajectory Alignment");
    residual = ceres_ICP(i, j, M, T, R, t, dim, residual, ceres_rot, mass_center);
    return residual;
  }
}
void pointCloudResize(pcl::PointCloud<pcl::PointXYZ>::Ptr targetPC)
{
  targetPC->width = 50;
  targetPC->height = 50;
  targetPC->points.resize(targetPC->width * targetPC->height);
  pcl_conversions::toPCL(ros::Time::now(), targetPC->header.stamp);
  targetPC->header.frame_id = "/camera_init";
}
int hectorEncoder(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Hector, double *&M, int i,
  int dim, Eigen::Vector3d &mass_center)
{

  float m_c_x = 0.0;
  float m_c_y = 0.0;
  float m_c_z = 0.0;
  pcl_conversions::toPCL(ros::Time::now(), cloud_Hector->header.stamp);
  cloud_Hector->header.frame_id = "/camera_init";
  ROS_INFO("Processing hactor points from %d to %d, processed [%f] points",
  hactor_offset, hactor_count, round_hector_points);
  while (hactor_offset < Hactor_Path.poses.size())
  {
    M[i * dim + 0] = Hactor_Path.poses[hactor_offset].pose.position.x;
    M[i * dim + 1] = Hactor_Path.poses[hactor_offset].pose.position.y;
    M[i * dim + 2] = Hactor_Path.poses[hactor_offset].pose.position.z;
    cloud_Hector->points[i].x = Hactor_Path.poses[hactor_offset].pose.position.x;
    cloud_Hector->points[i].y = Hactor_Path.poses[hactor_offset].pose.position.y;
    cloud_Hector->points[i].z = Hactor_Path.poses[hactor_offset].pose.position.z;

    hactor_offset++;

    m_c_x += M[i * dim + 0];
    m_c_y += M[i * dim + 1];
    m_c_z += M[i * dim + 2];
    i++;
  }
  mass_center.x() = m_c_x / i;
  mass_center.y() = m_c_y / i;
  mass_center.z() = m_c_z / i;
  ROS_INFO("Hector points downsamping to %d points.", i);
  return i;
}
int kittiEncoder(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_Tracker, double *&T, int j,
  int dim)
{
  double factor = round_tracker_points / round_hector_points;
  double cont = laser_offset;
  pcl_conversions::toPCL(ros::Time::now(), cloud_Tracker->header.stamp);
  cloud_Tracker->header.frame_id = "/camera_init";
  ROS_INFO("Processing Kitti_GPS points from %d to %d, processed [%f] points",
  laser_offset, laser_count, round_tracker_points);
  while (laser_offset < Laser_Path.poses.size())
  {
    T[j * dim + 0] = Laser_Path.poses[laser_offset].pose.position.x;
    T[j * dim + 1] = Laser_Path.poses[laser_offset].pose.position.y;
    T[j * dim + 2] = Laser_Path.poses[laser_offset].pose.position.z;
    cloud_Tracker->points[j].x = Laser_Path.poses[laser_offset].pose.position.x;
    cloud_Tracker->points[j].y = Laser_Path.poses[laser_offset].pose.position.y;
```

```
      cloud_Tracker->points[j].z = Laser_Path.poses[laser_offset].pose.position.z;
      cont += factor;
      laser_offset = ceil(cont);

      j++;
    }
    ROS_INFO("Kitti_GPS points downsamping to %d points.", j);
    return j;
  }
  void ITMEncoder(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_ICP, double *&M, int i, int
    dim)
  {
    pointCloudResize(cloud_ICP);
    pcl_conversions::toPCL(ros::Time::now(), cloud_ICP->header.stamp);
    cloud_ICP->header.frame_id = "camera_init";
    for (int c = 0; c < i; ++c)
    {

      cloud_ICP->points[c].x = M[dim * c + 0];
      cloud_ICP->points[c].y = M[dim * c + 1];
      cloud_ICP->points[c].z = M[dim * c + 2];
    }
  }
  void ITMpub(ros::Time laser_stamp_min, ros::Time laser_stamp_max, double *ceres_rot,
    Eigen::Vector3d mass_center, int dim)
  {
    double q_curr[4];
    geometry_msgs::Quaternion ICP_rot;
    geometry_msgs::Vector3 ICP_trans;
    AngleAxisToQuaternion(ceres_rot, q_curr);
    ICP_rot.w = q_curr[0];
    ICP_rot.x = q_curr[1];
    ICP_rot.y = q_curr[2];
    ICP_rot.z = q_curr[3];
    ICP_trans.x = ceres_rot[3];
    ICP_trans.y = ceres_rot[4];
    ICP_trans.z = ceres_rot[5];
    geometry_msgs::TransformStamped ICP_message;
    geometry_msgs::Transform new_frame_trans;
    new_frame_trans.translation = ICP_trans;
    new_frame_trans.rotation = ICP_rot;

    ICP_message.header.frame_id = "camera_init";
    ICP_message.transform = new_frame_trans;

    PCL_Trans_pub.publish(ICP_message);
    ROS_INFO("Current Trans Time: %f to %f\n", laser_stamp_min.toSec(), laser_stamp_max.
    toSec());
  }
  void rotatePoint(double *&points_in, int &points_num, Eigen::Quaterniond Rq, Eigen::
    Vector3d t, int dim, double *ceres_rot)
  {
    for (int i = 0; i < points_num; ++i)
    {
      double pointout[3] = {points_in[dim * i + 0], points_in[dim * i + 1], points_in[dim
     * i + 2]};
      AngleAxisRotatePoint(ceres_rot, pointout, pointout);

      points_in[dim * i + 0] = pointout[0] + ceres_rot[3];
      points_in[dim * i + 1] = pointout[1] + ceres_rot[4];
      points_in[dim * i + 2] = pointout[2] + ceres_rot[5];
    }
  }
  void hector_path_Callback(const nav_msgs::Path &msg)
  {
    Hactor_Path = msg;
    hactor_count = Hactor_Path.poses.size();

  }
  void kitti_path_Callback(const nav_msgs::Path &msg)
  {
    Laser_Path = msg;
    laser_count = Laser_Path.poses.size();

  }
  double ceres_ICP(int &i, int &j, double *&M, double *&T, Eigen::Quaterniond &R, Eigen::
    Vector3d &t, int dim, double residual, double *ceres_rot3, Eigen::Vector3d &
    mass_center)
  {
```

```
    double cere_r_t[6] = {0.0, 0, 0, 0, 0, 0};


    vector<Eigen::Vector3d> pts1, pts2;
    for (int i1 = 0; i1 < i; i1++)
    {
      Eigen::Vector3d cp1;
      cp1.x() = M[dim * i1 + 0];
      cp1.y() = M[dim * i1 + 1];
      cp1.z() = M[dim * i1 + 2];
      pts1.push_back(cp1);
    }
    for (int j1 = 0; j1 < j; j1++)
    {
      Eigen::Vector3d cp2;
      cp2.x() = T[dim * j1 + 0];
      cp2.y() = T[dim * j1 + 1];
      cp2.z() = T[dim * j1 + 2];
      pts2.push_back(cp2);
    }
    ceres::Problem problem;
    for (int conti = 0; conti < pts2.size(); conti++)
    {
      ceres::CostFunction *costfunction = new ceres::AutoDiffCostFunction<
    cost_function_define, 3, 6>(new cost_function_define(pts1[conti], pts2[conti]));
      problem.AddResidualBlock(costfunction, NULL, cere_r_t);
    }
    ceres::Solver::Options option;
    option.linear_solver_type = ceres::DENSE_QR;

    option.max_num_iterations =100;

    ceres::Solver::Summary summary;

    ceres::Solve(option, &problem, &summary);

    ceres_rot3[0] = cere_r_t[0];
    ceres_rot3[1] = cere_r_t[1];
    ceres_rot3[2] = cere_r_t[2];
    ceres_rot3[3] = cere_r_t[3];
    ceres_rot3[4] = cere_r_t[4];
    ceres_rot3[5] = cere_r_t[5];
    double q_curr[4];
    AngleAxisToQuaternion(cere_r_t, q_curr);
    cout << "R = " << cere_r_t[0] << ", " << cere_r_t[1] << ", " << cere_r_t[2] << endl;
    cout << "Q = " << q_curr[0] << ", " << q_curr[1] << ", " << q_curr[2] << "," <<
    q_curr[3] << endl;
    cout << "t = " << cere_r_t[3] << ", " << cere_r_t[4] << ", " << cere_r_t[5] << endl;
    cout << "Average Cost = " << summary.final_cost/p_sampling_num << endl;
    R.w() = q_curr[0];
    R.x() = q_curr[1];
    R.y() = q_curr[2];
    R.z() = q_curr[3];
    t.x() = cere_r_t[3];
    t.y() = cere_r_t[4];
    t.z() = cere_r_t[5];
    if(summary.IsSolutionUsable()==true){
      return summary.final_cost/p_sampling_num;
    }else{
      return 99999;
    }
  }

  ros::NodeHandle n_;
  ros::Subscriber sub_, sub_kitti;
  ros::Subscriber tracker_sub;
  ros::Subscriber gps_sub;
  ros::Subscriber pose_sub;
  ros::Timer timer;
  ros::Publisher icp_path_pub;
  ros::Publisher align_path_pub;
  ros::Publisher hector_path_pub;
  ros::Publisher tracker_path_pub;
  ros::Publisher PCL_Trans_pub;
  ros::Publisher gps_pub;
  nav_msgs::Path Hactor_Path;
  nav_msgs::Path Laser_Path;

  tf2_ros::TransformBroadcaster pcl_br;
  int reference_location_init;
```

```
  float reference_location_init_x;
  float reference_location_init_y;
  float reference_location_init_z;
  bool gps_location_init;
  float gps_location_init_x;
  float gps_location_init_y;
  float gps_location_init_z;
  int laser_count;
  int hactor_count;
  int laser_offset;
  int hactor_offset;
  ros::Time laser_stamp_min;
  ros::Time laser_stamp_max;
  int first_call_escaper;
  int orientation_aline_escaper;
  ros::Duration windowsXP_offset;
  int path_flag;
  double min_residual;
  Eigen::Quaterniond path_rotation;
  Eigen::Vector3d path_translation;
  int p_sampling_round;
  int p_sampling_num;
  int p_iter_cycle;
  float p_gap_limit;
  float p_angle_limit;
  int p_residual_limit;
  float round_hector_points;
  float round_tracker_points;
  float val_tm[3 * 3];
  float val_Rm[3 * 3];
};
int main(int argc, char **argv)
{
  ros::init(argc, argv, "path_icp");
  icp_iter_class my_pcl;
  ros::spin();
  return 0;
}
```

A.3.2 Scan-to-scan odometry

```
//This code is created based on A-LOAM by Weichen WEI: weichen.wei@monash.edu
#include <math.h>
#include <vector>
#include <loam_itm/common.h>
#include <nav_msgs/Odometry.h>
#include <nav_msgs/Path.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/TransformStamped.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <ros/ros.h>
#include <sensor_msgs/Imu.h>
#include <sensor_msgs/PointCloud2.h>
#include <tf/transform_datatypes.h>
#include <tf/transform_broadcaster.h>
#include <eigen3/Eigen/Dense>
#include <ceres/ceres.h>
#include <mutex>
#include <queue>
#include <thread>
#include <iostream>
#include <string>
#include "loam_itm/rotation.h"
#include "lidarFactor.hpp"
#include "loam_itm/common.h"
#include "loam_itm/tic_toc.h"
int frameCount = 0;
double timeLaserCloudCornerLast = 0;
double timeLaserCloudSurfLast = 0;
double timeLaserCloudFullRes = 0;
double timeLaserOdometry = 0;
int laserCloudCenWidth = 10;
int laserCloudCenHeight = 10;
int laserCloudCenDepth = 5;
const int laserCloudWidth = 21;
const int laserCloudHeight = 21;
```

```cpp
const int laserCloudDepth = 11;
const int laserCloudNum = laserCloudWidth * laserCloudHeight * laserCloudDepth;
int laserCloudValidInd[125];
int laserCloudSurroundInd[125];

pcl::PointCloud<PointType>::Ptr laserCloudCornerLast(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudSurfLast(new pcl::PointCloud<PointType>());

pcl::PointCloud<PointType>::Ptr laserCloudSurround(new pcl::PointCloud<PointType>());

pcl::PointCloud<PointType>::Ptr laserCloudCornerFromMap(new pcl::PointCloud<PointType>())
    ;
pcl::PointCloud<PointType>::Ptr laserCloudSurfFromMap(new pcl::PointCloud<PointType>());

pcl::PointCloud<PointType>::Ptr laserCloudFullRes(new pcl::PointCloud<PointType>());

pcl::PointCloud<PointType>::Ptr laserCloudCornerArray[laserCloudNum];
pcl::PointCloud<PointType>::Ptr laserCloudSurfArray[laserCloudNum];

pcl::KdTreeFLANN<PointType>::Ptr kdtreeCornerFromMap(new pcl::KdTreeFLANN<PointType>());
pcl::KdTreeFLANN<PointType>::Ptr kdtreeSurfFromMap(new pcl::KdTreeFLANN<PointType>());
double parameters[7] = {0, 0, 0, 1, 0, 0, 0};

Eigen::Map<Eigen::Quaterniond> q_w_curr(parameters);
Eigen::Map<Eigen::Vector3d> t_w_curr(parameters + 4);


Eigen::Quaterniond q_wmap_wodom(1, 0, 0, 0);
Eigen::Vector3d t_wmap_wodom(0, 0, 0);

Eigen::Quaterniond q_wodom_curr(1, 0, 0, 0);
Eigen::Vector3d t_wodom_curr(0, 0, 0);
Eigen::Quaterniond q_ITM_curr(1, 0, 0, 0);
Eigen::Vector3d t_ITM_curr(0, 0, 0);
int IMT_flag = 0;
int p_slerp = 0;
std::queue<sensor_msgs::PointCloud2ConstPtr> cornerLastBuf;
std::queue<sensor_msgs::PointCloud2ConstPtr> surfLastBuf;
std::queue<sensor_msgs::PointCloud2ConstPtr> fullResBuf;
std::queue<nav_msgs::Odometry::ConstPtr> odometryBuf;
std::mutex mBuf;
pcl::VoxelGrid<PointType> downSizeFilterCorner;
pcl::VoxelGrid<PointType> downSizeFilterSurf;
std::vector<int> pointSearchInd;
std::vector<float> pointSearchSqDis;
PointType pointOri, pointSel;
ros::Publisher pubLaserCloudSurround, pubLaserCloudMap, pubLaserCloudFullRes,
    pubOdomAftMapped, pubOdomAftMappedHighFrec, pubLaserAfterMappedPath;
nav_msgs::Path laserAfterMappedPath;




void correctionAssociateToMap()
{
  double slerp_step = p_slerp*1.0;
  Eigen::Quaterniond qres;
  Eigen::Quaterniond qa = Eigen::Quaterniond::Identity();
  Eigen::Vector3d t_slerp = t_ITM_curr / slerp_step;
  qres = qa.slerp(1.00 / slerp_step, q_ITM_curr);
  if (IMT_flag < slerp_step)
  {
    q_w_curr = qres * q_w_curr;
    t_w_curr = qres * t_w_curr + t_slerp;
    IMT_flag++;
  }
}

void transformAssociateToMap()
{
  q_w_curr = q_wmap_wodom * q_wodom_curr;
  t_w_curr = q_wmap_wodom * t_wodom_curr + t_wmap_wodom;
}

void transformUpdate()
{
  q_wmap_wodom = q_w_curr * q_wodom_curr.inverse();
  t_wmap_wodom = t_w_curr - q_wmap_wodom * t_wodom_curr;
}

void pointAssociateToMap(PointType const *const pi, PointType *const po)
```

```
{
  Eigen::Vector3d point_curr(pi->x, pi->y, pi->z);
  Eigen::Vector3d point_w = q_w_curr * point_curr + t_w_curr;
  po->x = point_w.x();
  po->y = point_w.y();
  po->z = point_w.z();
  po->intensity = pi->intensity;

}

void pointAssociateTobeMapped(PointType const *const pi, PointType *const po)
{
  Eigen::Vector3d point_w(pi->x, pi->y, pi->z);
  Eigen::Vector3d point_curr = q_w_curr.inverse() * (point_w - t_w_curr);
  po->x = point_curr.x();
  po->y = point_curr.y();
  po->z = point_curr.z();
  po->intensity = pi->intensity;
}
void laserCloudCornerLastHandler(const sensor_msgs::PointCloud2ConstPtr &
    laserCloudCornerLast2)
{
  mBuf.lock();
  cornerLastBuf.push(laserCloudCornerLast2);
  mBuf.unlock();
}
void laserCloudSurfLastHandler(const sensor_msgs::PointCloud2ConstPtr &
    laserCloudSurfLast2)
{
  mBuf.lock();
  surfLastBuf.push(laserCloudSurfLast2);
  mBuf.unlock();
}
void laserCloudFullResHandler(const sensor_msgs::PointCloud2ConstPtr &laserCloudFullRes2)
{
  mBuf.lock();
  fullResBuf.push(laserCloudFullRes2);
  mBuf.unlock();
}

void laserOdometryHandler(const nav_msgs::Odometry::ConstPtr &laserOdometry)
{
  mBuf.lock();
  odometryBuf.push(laserOdometry);
  mBuf.unlock();

  Eigen::Quaterniond q_wodom_curr;
  Eigen::Vector3d t_wodom_curr;
  q_wodom_curr.x() = laserOdometry->pose.pose.orientation.x;
  q_wodom_curr.y() = laserOdometry->pose.pose.orientation.y;
  q_wodom_curr.z() = laserOdometry->pose.pose.orientation.z;
  q_wodom_curr.w() = laserOdometry->pose.pose.orientation.w;
  t_wodom_curr.x() = laserOdometry->pose.pose.position.x;
  t_wodom_curr.y() = laserOdometry->pose.pose.position.y;
  t_wodom_curr.z() = laserOdometry->pose.pose.position.z;
  Eigen::Quaterniond q_w_curr = q_wmap_wodom * q_wodom_curr;
  Eigen::Vector3d t_w_curr = q_wmap_wodom * t_wodom_curr + t_wmap_wodom;
  nav_msgs::Odometry odomAftMapped;
  odomAftMapped.header.frame_id = "/camera_init";
  odomAftMapped.child_frame_id = "/aft_mapped";
  odomAftMapped.header.stamp = laserOdometry->header.stamp;
  odomAftMapped.pose.pose.orientation.x = q_w_curr.x();
  odomAftMapped.pose.pose.orientation.y = q_w_curr.y();
  odomAftMapped.pose.pose.orientation.z = q_w_curr.z();
  odomAftMapped.pose.pose.orientation.w = q_w_curr.w();
  odomAftMapped.pose.pose.position.x = t_w_curr.x();
  odomAftMapped.pose.pose.position.y = t_w_curr.y();
  odomAftMapped.pose.pose.position.z = t_w_curr.z();
  pubOdomAftMappedHighFrec.publish(odomAftMapped);
}
void laserTrackerCallBack(const geometry_msgs::TransformStamped Trans)
{
  IMT_flag = 0;
  Eigen::Quaterniond q_ITM_last;
  Eigen::Vector3d t_ITM_last;
  t_ITM_last.x() = Trans.transform.translation.x;
  t_ITM_last.y() = Trans.transform.translation.y;
  t_ITM_last.z() = Trans.transform.translation.z;
```

```
  q_ITM_last.w() = Trans.transform.rotation.w;
  q_ITM_last.x() = Trans.transform.rotation.x;
  q_ITM_last.y() = Trans.transform.rotation.y;
  q_ITM_last.z() = Trans.transform.rotation.z;
  q_ITM_curr = q_ITM_last;
  t_ITM_curr = t_ITM_last;
}
void process()
{
  while (1)
  {


    while (!cornerLastBuf.empty() && !surfLastBuf.empty() &&
         !fullResBuf.empty() && !odometryBuf.empty())
    {
      mBuf.lock();

      while (!odometryBuf.empty() && odometryBuf.front()->header.stamp.toSec() <
    cornerLastBuf.front()->header.stamp.toSec())
        odometryBuf.pop();
      if (odometryBuf.empty())
      {
        mBuf.unlock();
        break;
      }

      while (!surfLastBuf.empty() && surfLastBuf.front()->header.stamp.toSec() <
    cornerLastBuf.front()->header.stamp.toSec())
        surfLastBuf.pop();
      if (surfLastBuf.empty())
      {
        mBuf.unlock();
        break;
      }

      while (!fullResBuf.empty() && fullResBuf.front()->header.stamp.toSec() <
    cornerLastBuf.front()->header.stamp.toSec())
        fullResBuf.pop();
      if (fullResBuf.empty())
      {
        mBuf.unlock();
        break;
      }

      timeLaserCloudCornerLast = cornerLastBuf.front()->header.stamp.toSec();
      timeLaserCloudSurfLast = surfLastBuf.front()->header.stamp.toSec();
      timeLaserCloudFullRes = fullResBuf.front()->header.stamp.toSec();
      timeLaserOdometry = odometryBuf.front()->header.stamp.toSec();

      if (timeLaserCloudCornerLast != timeLaserOdometry ||
        timeLaserCloudSurfLast != timeLaserOdometry ||
        timeLaserCloudFullRes != timeLaserOdometry)
      {
        printf("time corner %f surf %f full %f odom %f \n", timeLaserCloudCornerLast,
    timeLaserCloudSurfLast, timeLaserCloudFullRes, timeLaserOdometry);
        printf("unsync messeage!");
        mBuf.unlock();
        break;
      }

      laserCloudCornerLast->clear();
      pcl::fromROSMsg(*cornerLastBuf.front(), *laserCloudCornerLast);
      cornerLastBuf.pop();

      laserCloudSurfLast->clear();
      pcl::fromROSMsg(*surfLastBuf.front(), *laserCloudSurfLast);
      surfLastBuf.pop();

      laserCloudFullRes->clear();
      pcl::fromROSMsg(*fullResBuf.front(), *laserCloudFullRes);
      fullResBuf.pop();

      q_wodom_curr.x() = odometryBuf.front()->pose.pose.orientation.x;
      q_wodom_curr.y() = odometryBuf.front()->pose.pose.orientation.y;
      q_wodom_curr.z() = odometryBuf.front()->pose.pose.orientation.z;
      q_wodom_curr.w() = odometryBuf.front()->pose.pose.orientation.w;
      t_wodom_curr.x() = odometryBuf.front()->pose.pose.position.x;
```

```
      t_wodom_curr.y() = odometryBuf.front()->pose.pose.position.y;
      t_wodom_curr.z() = odometryBuf.front()->pose.pose.position.z;
      odometryBuf.pop();

      while (!cornerLastBuf.empty())
      {
        cornerLastBuf.pop();
        printf("drop lidar frame in mapping for real time performance \n");
      }
      mBuf.unlock();
      TicToc t_whole;

      transformAssociateToMap();
      TicToc t_shift;


      int centerCubeI = int((t_w_curr.x() + 25.0) / 50.0) + laserCloudCenWidth;
      int centerCubeJ = int((t_w_curr.y() + 25.0) / 50.0) + laserCloudCenHeight;
      int centerCubeK = int((t_w_curr.z() + 25.0) / 50.0) + laserCloudCenDepth;

      if (t_w_curr.x() + 25.0 < 0)
        centerCubeI--;
      if (t_w_curr.y() + 25.0 < 0)
        centerCubeJ--;
      if (t_w_curr.z() + 25.0 < 0)
        centerCubeK--;


      while (centerCubeI < 3)
      {
        for (int j = 0; j < laserCloudHeight; j++)
        {
          for (int k = 0; k < laserCloudDepth; k++)
          {
            int i = laserCloudWidth - 1;

            pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k];
            pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k];
            for (; i >= 1; i--)
            {
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k] =
                laserCloudCornerArray[i - 1 + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k];
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k] =
                laserCloudSurfArray[i - 1 + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k];
            }
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k] =
              laserCloudCubeCornerPointer;
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k] =
              laserCloudCubeSurfPointer;
            laserCloudCubeCornerPointer->clear();
            laserCloudCubeSurfPointer->clear();
          }
        }
        centerCubeI++;
        laserCloudCenWidth++;
      }

      while (centerCubeI >= laserCloudWidth - 3)
      {
        for (int j = 0; j < laserCloudHeight; j++)
        {
          for (int k = 0; k < laserCloudDepth; k++)
          {
            int i = 0;
            pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth *
    laserCloudHeight * k];
            pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth *
```

```
laserCloudHeight * k];
          for (; i < laserCloudWidth - 1; i++)
          {
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
              laserCloudCornerArray[i + 1 + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
              laserCloudSurfArray[i + 1 + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
          }
          laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
            laserCloudCubeCornerPointer;
          laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
            laserCloudCubeSurfPointer;
          laserCloudCubeCornerPointer->clear();
          laserCloudCubeSurfPointer->clear();
        }
      }
      centerCubeI--;
      laserCloudCenWidth--;
    }
    while (centerCubeJ < 3)
    {
      for (int i = 0; i < laserCloudWidth; i++)
      {
        for (int k = 0; k < laserCloudDepth; k++)
        {
          int j = laserCloudHeight - 1;
          pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
          pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
          for (; j >= 1; j--)
          {
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
              laserCloudCornerArray[i + laserCloudWidth * (j - 1) + laserCloudWidth * laserCloudHeight * k];
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
              laserCloudSurfArray[i + laserCloudWidth * (j - 1) + laserCloudWidth * laserCloudHeight * k];
          }
          laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
            laserCloudCubeCornerPointer;
          laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
            laserCloudCubeSurfPointer;
          laserCloudCubeCornerPointer->clear();
          laserCloudCubeSurfPointer->clear();
        }
      }
      centerCubeJ++;
      laserCloudCenHeight++;
    }
    while (centerCubeJ >= laserCloudHeight - 3)
    {
      for (int i = 0; i < laserCloudWidth; i++)
      {
        for (int k = 0; k < laserCloudDepth; k++)
        {
          int j = 0;
          pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
          pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
          for (; j < laserCloudHeight - 1; j++)
          {
```

```
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCornerArray[i + laserCloudWidth * (j + 1) + laserCloudWidth * laserCloudHeight * k];
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudSurfArray[i + laserCloudWidth * (j + 1) + laserCloudWidth * laserCloudHeight * k];
            }
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
              laserCloudCubeCornerPointer;
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
              laserCloudCubeSurfPointer;
            laserCloudCubeCornerPointer->clear();
            laserCloudCubeSurfPointer->clear();
          }
        }
        centerCubeJ--;
        laserCloudCenHeight--;
      }
      while (centerCubeK < 3)
      {
        for (int i = 0; i < laserCloudWidth; i++)
        {
          for (int j = 0; j < laserCloudHeight; j++)
          {
            int k = laserCloudDepth - 1;
            pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            for (; k >= 1; k--)
            {
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * (k - 1)];
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * (k - 1)];
            }
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
              laserCloudCubeCornerPointer;
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
              laserCloudCubeSurfPointer;
            laserCloudCubeCornerPointer->clear();
            laserCloudCubeSurfPointer->clear();
          }
        }
        centerCubeK++;
        laserCloudCenDepth++;
      }
      while (centerCubeK >= laserCloudDepth - 3)
      {
        for (int i = 0; i < laserCloudWidth; i++)
        {
          for (int j = 0; j < laserCloudHeight; j++)
          {
            int k = 0;
            pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            for (; k < laserCloudDepth - 1; k++)
            {
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth *
```

```
laserCloudHeight * (k + 1)];
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth *
laserCloudHeight * k] =
                laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth *
laserCloudHeight * (k + 1)];
            }
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth *
laserCloudHeight * k] =
              laserCloudCubeCornerPointer;
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth *
laserCloudHeight * k] =
              laserCloudCubeSurfPointer;
            laserCloudCubeCornerPointer->clear();
            laserCloudCubeSurfPointer->clear();
          }
        }
        centerCubeK--;
        laserCloudCenDepth--;
      }
      int laserCloudValidNum = 0;
      int laserCloudSurroundNum = 0;



      for (int i = centerCubeI - 2; i <= centerCubeI + 2; i++)
      {
        for (int j = centerCubeJ - 2; j <= centerCubeJ + 2; j++)
        {
          for (int k = centerCubeK - 1; k <= centerCubeK + 1; k++)
          {
            if (i >= 0 && i < laserCloudWidth &&
              j >= 0 && j < laserCloudHeight &&
              k >= 0 && k < laserCloudDepth)
            {

              laserCloudValidInd[laserCloudValidNum] = i + laserCloudWidth * j +
laserCloudWidth * laserCloudHeight * k;
              laserCloudValidNum++;

              laserCloudSurroundInd[laserCloudSurroundNum] = i + laserCloudWidth * j +
laserCloudWidth * laserCloudHeight * k;
              laserCloudSurroundNum++;
            }
          }
        }
      }
      laserCloudCornerFromMap->clear();
      laserCloudSurfFromMap->clear();

      for (int i = 0; i < laserCloudValidNum; i++)
      {
        *laserCloudCornerFromMap += *laserCloudCornerArray[laserCloudValidInd[i]];
        *laserCloudSurfFromMap += *laserCloudSurfArray[laserCloudValidInd[i]];
      }
      int laserCloudCornerFromMapNum = laserCloudCornerFromMap->points.size();
      int laserCloudSurfFromMapNum = laserCloudSurfFromMap->points.size();


      pcl::PointCloud<PointType>::Ptr laserCloudCornerStack(new pcl::PointCloud<PointType
>());
      downSizeFilterCorner.setInputCloud(laserCloudCornerLast);
      downSizeFilterCorner.filter(*laserCloudCornerStack);
      int laserCloudCornerStackNum = laserCloudCornerStack->points.size();
      pcl::PointCloud<PointType>::Ptr laserCloudSurfStack(new pcl::PointCloud<PointType
>());
      downSizeFilterSurf.setInputCloud(laserCloudSurfLast);
      downSizeFilterSurf.filter(*laserCloudSurfStack);
      int laserCloudSurfStackNum = laserCloudSurfStack->points.size();
      printf("map prepare time %f ms\n", t_shift.toc());
      printf("map corner num %d  surf num %d \n", laserCloudCornerFromMapNum,
laserCloudSurfFromMapNum);

      if (laserCloudCornerFromMapNum > 10 && laserCloudSurfFromMapNum > 50)
      {
        TicToc t_opt;
        TicToc t_tree;
        kdtreeCornerFromMap->setInputCloud(laserCloudCornerFromMap);
        kdtreeSurfFromMap->setInputCloud(laserCloudSurfFromMap);
        printf("build tree time %f ms \n", t_tree.toc());
```

```

      for (int iterCount = 0; iterCount < 2; iterCount++)
      {

        ceres::LossFunction *loss_function = new ceres::HuberLoss(0.1);
        ceres::LocalParameterization *q_parameterization =
          new ceres::EigenQuaternionParameterization();
        ceres::Problem::Options problem_options;
        ceres::Problem problem(problem_options);
        problem.AddParameterBlock(parameters, 4, q_parameterization);
        problem.AddParameterBlock(parameters + 4, 3);
        TicToc t_data;
        int corner_num = 0;
        for (int i = 0; i < laserCloudCornerStackNum; i++)
        {
          pointOri = laserCloudCornerStack->points[i];

          pointAssociateToMap(&pointOri, &pointSel);
          kdtreeCornerFromMap->nearestKSearch(pointSel, 5, pointSearchInd,
    pointSearchSqDis);
          if (pointSearchSqDis[4] < 1.0)
          {
            std::vector<Eigen::Vector3d> nearCorners;
            Eigen::Vector3d center(0, 0, 0);
            for (int j = 0; j < 5; j++)
            {
              Eigen::Vector3d tmp(laserCloudCornerFromMap->points[pointSearchInd[j]].x,
                                  laserCloudCornerFromMap->points[pointSearchInd[j]].y,
                                  laserCloudCornerFromMap->points[pointSearchInd[j]].z);
              center = center + tmp;
              nearCorners.push_back(tmp);
            }
            center = center / 5.0;
            Eigen::Matrix3d covMat = Eigen::Matrix3d::Zero();
            for (int j = 0; j < 5; j++)
            {
              Eigen::Matrix<double, 3, 1> tmpZeroMean = nearCorners[j] - center;

              covMat = covMat + tmpZeroMean * tmpZeroMean.transpose();
            }
            Eigen::SelfAdjointEigenSolver<Eigen::Matrix3d> saes(covMat);




            Eigen::Vector3d unit_direction = saes.eigenvectors().col(2);
            Eigen::Vector3d curr_point(pointOri.x, pointOri.y, pointOri.z);
            if (saes.eigenvalues()[2] > 3 * saes.eigenvalues()[1])
            {
              Eigen::Vector3d point_on_line = center;
              Eigen::Vector3d point_a, point_b;
              point_a = 0.1 * unit_direction + point_on_line;
              point_b = -0.1 * unit_direction + point_on_line;
              ceres::CostFunction *cost_function = LidarEdgeFactor::Create(curr_point,
    point_a, point_b, 1.0);
              problem.AddResidualBlock(cost_function, loss_function, parameters,
    parameters + 4);
              corner_num++;
            }
          }

        }

        int surf_num = 0;
        for (int i = 0; i < laserCloudSurfStackNum; i++)
        {
          pointOri = laserCloudSurfStack->points[i];

          pointAssociateToMap(&pointOri, &pointSel);
          kdtreeSurfFromMap->nearestKSearch(pointSel, 5, pointSearchInd,
    pointSearchSqDis);
          Eigen::Matrix<double, 5, 3> matA0;
          Eigen::Matrix<double, 5, 1> matB0 = -1 * Eigen::Matrix<double, 5, 1>::Ones();
          if (pointSearchSqDis[4] < 1.0)
          {
            for (int j = 0; j < 5; j++)
            {
              matA0(j, 0) = laserCloudSurfFromMap->points[pointSearchInd[j]].x;
              matA0(j, 1) = laserCloudSurfFromMap->points[pointSearchInd[j]].y;
```

```
            matA0(j, 2) = laserCloudSurfFromMap->points[pointSearchInd[j]].z;

          }


          Eigen::Vector3d norm = matA0.colPivHouseholderQr().solve(matB0);
          double negative_OA_dot_norm = 1 / norm.norm();
          norm.normalize();

          bool planeValid = true;
          for (int j = 0; j < 5; j++)
          {

            if (fabs(norm(0) * laserCloudSurfFromMap->points[pointSearchInd[j]].x +
                 norm(1) * laserCloudSurfFromMap->points[pointSearchInd[j]].y +
                 norm(2) * laserCloudSurfFromMap->points[pointSearchInd[j]].z +
negative_OA_dot_norm) > 0.2)
            {
              planeValid = false;
              break;
            }
          }
          Eigen::Vector3d curr_point(pointOri.x, pointOri.y, pointOri.z);
          if (planeValid)
          {
            ceres::CostFunction *cost_function = LidarPlaneNormFactor::Create(
curr_point, norm, negative_OA_dot_norm);
            problem.AddResidualBlock(cost_function, loss_function, parameters,
parameters + 4);
            surf_num++;
          }
        }

      }


      printf("mapping data assosiation time %f ms \n", t_data.toc());
      TicToc t_solver;
      ceres::Solver::Options options;
      options.linear_solver_type = ceres::DENSE_QR;
      options.max_num_iterations = 4;
      options.minimizer_progress_to_stdout = false;
      options.check_gradients = false;
      options.gradient_check_relative_precision = 1e-4;
      ceres::Solver::Summary summary;
      ceres::Solve(options, &problem, &summary);
      printf("mapping solver time %f ms \n", t_solver.toc());




    }
    printf("mapping optimization time %f \n", t_opt.toc());
  }
  else
  {
    ROS_WARN("time Map corner and surf num are not enough");
  }
  correctionAssociateToMap();
  transformUpdate();
  TicToc t_add;

  for (int i = 0; i < laserCloudCornerStackNum; i++)
  {
    pointAssociateToMap(&laserCloudCornerStack->points[i], &pointSel);

    int cubeI = int((pointSel.x + 25.0) / 50.0) + laserCloudCenWidth;
    int cubeJ = int((pointSel.y + 25.0) / 50.0) + laserCloudCenHeight;
    int cubeK = int((pointSel.z + 25.0) / 50.0) + laserCloudCenDepth;
    if (pointSel.x + 25.0 < 0)
      cubeI--;
    if (pointSel.y + 25.0 < 0)
      cubeJ--;
    if (pointSel.z + 25.0 < 0)
      cubeK--;
    if (cubeI >= 0 && cubeI < laserCloudWidth &&
      cubeJ >= 0 && cubeJ < laserCloudHeight &&
      cubeK >= 0 && cubeK < laserCloudDepth)

    {
```

```
        int cubeInd = cubeI + laserCloudWidth * cubeJ + laserCloudWidth * laserCloudHeight * cubeK;
        laserCloudCornerArray[cubeInd]->push_back(pointSel);
      }
    }

    for (int i = 0; i < laserCloudSurfStackNum; i++)
    {
      pointAssociateToMap(&laserCloudSurfStack->points[i], &pointSel);
      int cubeI = int((pointSel.x + 25.0) / 50.0) + laserCloudCenWidth;
      int cubeJ = int((pointSel.y + 25.0) / 50.0) + laserCloudCenHeight;
      int cubeK = int((pointSel.z + 25.0) / 50.0) + laserCloudCenDepth;
      if (pointSel.x + 25.0 < 0)
        cubeI--;
      if (pointSel.y + 25.0 < 0)
        cubeJ--;
      if (pointSel.z + 25.0 < 0)
        cubeK--;
      if (cubeI >= 0 && cubeI < laserCloudWidth &&
        cubeJ >= 0 && cubeJ < laserCloudHeight &&
        cubeK >= 0 && cubeK < laserCloudDepth)
      {
        int cubeInd = cubeI + laserCloudWidth * cubeJ + laserCloudWidth * laserCloudHeight * cubeK;
        laserCloudSurfArray[cubeInd]->push_back(pointSel);
      }
    }
    printf("add points time %f ms\n", t_add.toc());
    TicToc t_filter;

    for (int i = 0; i < laserCloudValidNum; i++)
    {
      int ind = laserCloudValidInd[i];
      pcl::PointCloud<PointType>::Ptr tmpCorner(new pcl::PointCloud<PointType>());
      downSizeFilterCorner.setInputCloud(laserCloudCornerArray[ind]);
      downSizeFilterCorner.filter(*tmpCorner);
      laserCloudCornerArray[ind] = tmpCorner;
      pcl::PointCloud<PointType>::Ptr tmpSurf(new pcl::PointCloud<PointType>());
      downSizeFilterSurf.setInputCloud(laserCloudSurfArray[ind]);
      downSizeFilterSurf.filter(*tmpSurf);
      laserCloudSurfArray[ind] = tmpSurf;
    }
    printf("filter time %f ms \n", t_filter.toc());
    TicToc t_pub;


    if (frameCount % 5 == 0)
    {
      laserCloudSurround->clear();
      for (int i = 0; i < laserCloudSurroundNum; i++)
      {
        int ind = laserCloudSurroundInd[i];
        *laserCloudSurround += *laserCloudCornerArray[ind];
        *laserCloudSurround += *laserCloudSurfArray[ind];
      }
      sensor_msgs::PointCloud2 laserCloudSurround3;
      pcl::toROSMsg(*laserCloudSurround, laserCloudSurround3);
      laserCloudSurround3.header.stamp = ros::Time().fromSec(timeLaserOdometry);
      laserCloudSurround3.header.frame_id = "/camera_init";

    }


    if (frameCount % 20 == 0)
    {
      pcl::PointCloud<PointType> laserCloudMap;
      for (int i = 0; i < 4851; i++)
      {
        laserCloudMap += *laserCloudCornerArray[i];
        laserCloudMap += *laserCloudSurfArray[i];
      }
      sensor_msgs::PointCloud2 laserCloudMsg;
      pcl::toROSMsg(laserCloudMap, laserCloudMsg);
      laserCloudMsg.header.stamp = ros::Time().fromSec(timeLaserOdometry);
      laserCloudMsg.header.frame_id = "/camera_init";
      pubLaserCloudMap.publish(laserCloudMsg);
    }
    int laserCloudFullResNum = laserCloudFullRes->points.size();
```

```
      for (int i = 0; i < laserCloudFullResNum; i++)
      {
        pointAssociateToMap(&laserCloudFullRes->points[i], &laserCloudFullRes->points[i]);
      }
      sensor_msgs::PointCloud2 laserCloudFullRes3;
      pcl::toROSMsg(*laserCloudFullRes, laserCloudFullRes3);
      laserCloudFullRes3.header.stamp = ros::Time().fromSec(timeLaserOdometry);
      laserCloudFullRes3.header.frame_id = "/camera_init";
      pubLaserCloudFullRes.publish(laserCloudFullRes3);
      printf("mapping pub time %f ms \n", t_pub.toc());
      printf("whole mapping time %f ms +++++\n", t_whole.toc());
      nav_msgs::Odometry odomAftMapped;
      odomAftMapped.header.frame_id = "/camera_init";
      odomAftMapped.child_frame_id = "/aft_mapped";
      odomAftMapped.header.stamp = ros::Time().fromSec(timeLaserOdometry);
      odomAftMapped.pose.pose.orientation.x = q_w_curr.x();
      odomAftMapped.pose.pose.orientation.y = q_w_curr.y();
      odomAftMapped.pose.pose.orientation.z = q_w_curr.z();
      odomAftMapped.pose.pose.orientation.w = q_w_curr.w();
      odomAftMapped.pose.pose.position.x = t_w_curr.x();
      odomAftMapped.pose.pose.position.y = t_w_curr.y();
      odomAftMapped.pose.pose.position.z = t_w_curr.z();
      pubOdomAftMapped.publish(odomAftMapped);
      geometry_msgs::PoseStamped laserAfterMappedPose;
      laserAfterMappedPose.header = odomAftMapped.header;
      laserAfterMappedPose.pose = odomAftMapped.pose.pose;
      laserAfterMappedPath.header.stamp = odomAftMapped.header.stamp;
      laserAfterMappedPath.header.frame_id = "/camera_init";
      laserAfterMappedPath.poses.push_back(laserAfterMappedPose);
      pubLaserAfterMappedPath.publish(laserAfterMappedPath);
      static tf::TransformBroadcaster br;
      tf::Transform transform;
      tf::Quaternion q;
      transform.setOrigin(tf::Vector3(t_w_curr(0),
                                      t_w_curr(1),
                                      t_w_curr(2)));
      q.setW(q_w_curr.w());
      q.setX(q_w_curr.x());
      q.setY(q_w_curr.y());
      q.setZ(q_w_curr.z());
      transform.setRotation(q);
      br.sendTransform(tf::StampedTransform(transform, odomAftMapped.header.stamp, "/camera_init", "/aft_mapped"));
      frameCount++;
    }
    std::chrono::milliseconds dura(2);
    std::this_thread::sleep_for(dura);
  }
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "laserMapping");
  ros::NodeHandle nh;
  float lineRes = 0;
  float planeRes = 0;
  nh.param<float>("mapping_line_resolution", lineRes, 0.4);
  nh.param<float>("mapping_plane_resolution", planeRes, 0.8);
  nh.param<int>("slerp", p_slerp, 50);
  printf("line resolution %f plane resolution %f \n", lineRes, planeRes);
  downSizeFilterCorner.setLeafSize(lineRes, lineRes, lineRes);
  downSizeFilterSurf.setLeafSize(planeRes, planeRes, planeRes);

  ros::Subscriber subLaserCloudCornerLast = nh.subscribe<sensor_msgs::PointCloud2>("/laser_cloud_corner_last", 100, laserCloudCornerLastHandler);
  ros::Subscriber subLaserCloudSurfLast = nh.subscribe<sensor_msgs::PointCloud2>("/laser_cloud_surf_last", 100, laserCloudSurfLastHandler);
  ros::Subscriber subLaserOdometry = nh.subscribe<nav_msgs::Odometry>("/laser_odom_to_init", 100, laserOdometryHandler);
  ros::Subscriber subLaserCloudFullRes = nh.subscribe<sensor_msgs::PointCloud2>("/velodyne_cloud_3", 100, laserCloudFullResHandler);
  ros::Subscriber TransformSubscribe = nh.subscribe<geometry_msgs::TransformStamped>("/PCL_Trans", 2, laserTrackerCallBack);

  pubLaserCloudSurround = nh.advertise<sensor_msgs::PointCloud2>("/laser_cloud_surround", 100);
  pubLaserCloudMap = nh.advertise<sensor_msgs::PointCloud2>("/laser_cloud_map", 100);
  pubLaserCloudFullRes = nh.advertise<sensor_msgs::PointCloud2>("/
```

```
    velodyne_cloud_registered", 100);
  pubOdomAftMapped = nh.advertise<nav_msgs::Odometry>("/aft_mapped_to_init", 100);
  pubOdomAftMappedHighFrec = nh.advertise<nav_msgs::Odometry>("/
    aft_mapped_to_init_high_frec", 100);
  pubLaserAfterMappedPath = nh.advertise<nav_msgs::Path>("/aft_mapped_path", 100);
  for (int i = 0; i < laserCloudNum; i++)
  {
    laserCloudCornerArray[i].reset(new pcl::PointCloud<PointType>());
    laserCloudSurfArray[i].reset(new pcl::PointCloud<PointType>());
  }
  std::thread mapping_process{process};
  ros::spin();
  return 0;
}
```

A.3.3 Scan-to-map odometry

```
//This code is created based on A-LOAM by Weichen WEI: weichen.wei@monash.edu
#include <cmath>
#include <nav_msgs/Odometry.h>
#include <nav_msgs/Path.h>
#include <geometry_msgs/PoseStamped.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl_conversions/pcl_conversions.h>
#include <ros/ros.h>
#include <sensor_msgs/Imu.h>
#include <sensor_msgs/PointCloud2.h>
#include <tf/transform_datatypes.h>
#include <tf/transform_broadcaster.h>
#include <eigen3/Eigen/Dense>
#include <mutex>
#include <queue>
#include "loam_itm/common.h"
#include "loam_itm/tic_toc.h"
#include "lidarFactor.hpp"
#define DISTORTION 0

int corner_correspondence = 0, plane_correspondence = 0;
constexpr double SCAN_PERIOD = 0.1;
constexpr double DISTANCE_SQ_THRESHOLD = 25;
constexpr double NEARBY_SCAN = 2.5;
int skipFrameNum = 5;
bool systemInited = false;
double timeCornerPointsSharp = 0;
double timeCornerPointsLessSharp = 0;
double timeSurfPointsFlat = 0;
double timeSurfPointsLessFlat = 0;
double timeLaserCloudFullRes = 0;
pcl::KdTreeFLANN<pcl::PointXYZI>::Ptr kdtreeCornerLast(new pcl::KdTreeFLANN<pcl::
    PointXYZI>());
pcl::KdTreeFLANN<pcl::PointXYZI>::Ptr kdtreeSurfLast(new pcl::KdTreeFLANN<pcl::PointXYZI
    >());
pcl::PointCloud<PointType>::Ptr cornerPointsSharp(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr cornerPointsLessSharp(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr surfPointsFlat(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr surfPointsLessFlat(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudCornerLast(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudSurfLast(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudFullRes(new pcl::PointCloud<PointType>());
int laserCloudCornerLastNum = 0;
int laserCloudSurfLastNum = 0;

Eigen::Quaterniond q_w_curr(1, 0, 0, 0);
Eigen::Vector3d t_w_curr(0, 0, 0);

double para_q[4] = {0, 0, 0, 1};
double para_t[3] = {0, 0, 0};
Eigen::Map<Eigen::Quaterniond> q_last_curr(para_q);
Eigen::Map<Eigen::Vector3d> t_last_curr(para_t);
std::queue<sensor_msgs::PointCloud2ConstPtr> cornerSharpBuf;
std::queue<sensor_msgs::PointCloud2ConstPtr> cornerLessSharpBuf;
std::queue<sensor_msgs::PointCloud2ConstPtr> surfFlatBuf;
std::queue<sensor_msgs::PointCloud2ConstPtr> surfLessFlatBuf;
std::queue<sensor_msgs::PointCloud2ConstPtr> fullPointsBuf;
```

```
std::mutex mBuf;

void TransformToStart(PointType const *const pi, PointType *const po)
{

    double s;
    if (DISTORTION)
        s = (pi->intensity - int(pi->intensity)) / SCAN_PERIOD;
    else
        s = 1.0;

    Eigen::Quaterniond q_point_last = Eigen::Quaterniond::Identity().slerp(s, q_last_curr
    );
    Eigen::Vector3d t_point_last = s * t_last_curr;
    Eigen::Vector3d point(pi->x, pi->y, pi->z);
    Eigen::Vector3d un_point = q_point_last * point + t_point_last;
    po->x = un_point.x();
    po->y = un_point.y();
    po->z = un_point.z();
    po->intensity = pi->intensity;
}

void TransformToEnd(PointType const *const pi, PointType *const po)
{

    pcl::PointXYZI un_point_tmp;
    TransformToStart(pi, &un_point_tmp);
    Eigen::Vector3d un_point(un_point_tmp.x, un_point_tmp.y, un_point_tmp.z);
    Eigen::Vector3d point_end = q_last_curr.inverse() * (un_point - t_last_curr);
    po->x = point_end.x();
    po->y = point_end.y();
    po->z = point_end.z();

    po->intensity = int(pi->intensity);
}
void laserCloudSharpHandler(const sensor_msgs::PointCloud2ConstPtr &cornerPointsSharp2)
{
    mBuf.lock();
    cornerSharpBuf.push(cornerPointsSharp2);
    mBuf.unlock();
}
void laserCloudLessSharpHandler(const sensor_msgs::PointCloud2ConstPtr &
    cornerPointsLessSharp2)
{
    mBuf.lock();
    cornerLessSharpBuf.push(cornerPointsLessSharp2);
    mBuf.unlock();
}
void laserCloudFlatHandler(const sensor_msgs::PointCloud2ConstPtr &surfPointsFlat2)
{
    mBuf.lock();
    surfFlatBuf.push(surfPointsFlat2);
    mBuf.unlock();
}
void laserCloudLessFlatHandler(const sensor_msgs::PointCloud2ConstPtr &
    surfPointsLessFlat2)
{
    mBuf.lock();
    surfLessFlatBuf.push(surfPointsLessFlat2);
    mBuf.unlock();
}

void laserCloudFullResHandler(const sensor_msgs::PointCloud2ConstPtr &laserCloudFullRes2)
{
    mBuf.lock();
    fullPointsBuf.push(laserCloudFullRes2);
    mBuf.unlock();
}
int main(int argc, char **argv)
{
    ros::init(argc, argv, "laserOdometry");
    ros::NodeHandle nh;
    nh.param<int>("mapping_skip_frame", skipFrameNum, 2);
    printf("Mapping %d Hz \n", 10 / skipFrameNum);
    ros::Subscriber subCornerPointsSharp = nh.subscribe<sensor_msgs::PointCloud2>("/
    laser_cloud_sharp", 100, laserCloudSharpHandler);
    ros::Subscriber subCornerPointsLessSharp = nh.subscribe<sensor_msgs::PointCloud2>("/
    laser_cloud_less_sharp", 100, laserCloudLessSharpHandler);
```

```
    ros::Subscriber subSurfPointsFlat = nh.subscribe<sensor_msgs::PointCloud2>("/
    laser_cloud_flat", 100, laserCloudFlatHandler);
    ros::Subscriber subSurfPointsLessFlat = nh.subscribe<sensor_msgs::PointCloud2>("/
    laser_cloud_less_flat", 100, laserCloudLessFlatHandler);
    ros::Subscriber subLaserCloudFullRes = nh.subscribe<sensor_msgs::PointCloud2>("/
    velodyne_cloud_2", 100, laserCloudFullResHandler);
    ros::Publisher pubLaserCloudCornerLast = nh.advertise<sensor_msgs::PointCloud2>("/
    laser_cloud_corner_last", 100);
    ros::Publisher pubLaserCloudSurfLast = nh.advertise<sensor_msgs::PointCloud2>("/
    laser_cloud_surf_last", 100);
    ros::Publisher pubLaserCloudFullRes = nh.advertise<sensor_msgs::PointCloud2>("/
    velodyne_cloud_3", 100);
    ros::Publisher pubLaserOdometry = nh.advertise<nav_msgs::Odometry>("/
    laser_odom_to_init", 100);
    ros::Publisher pubLaserPath = nh.advertise<nav_msgs::Path>("/laser_odom_path", 100);
    nav_msgs::Path laserPath;
    int frameCount = 0;
    ros::Rate rate(100);
    while (ros::ok())
    {
        ros::spinOnce();
        if (!cornerSharpBuf.empty() && !cornerLessSharpBuf.empty() &&
            !surfFlatBuf.empty() && !surfLessFlatBuf.empty() &&
            !fullPointsBuf.empty())
        {
            timeCornerPointsSharp = cornerSharpBuf.front()->header.stamp.toSec();
            timeCornerPointsLessSharp = cornerLessSharpBuf.front()->header.stamp.toSec();
            timeSurfPointsFlat = surfFlatBuf.front()->header.stamp.toSec();
            timeSurfPointsLessFlat = surfLessFlatBuf.front()->header.stamp.toSec();
            timeLaserCloudFullRes = fullPointsBuf.front()->header.stamp.toSec();
            if (timeCornerPointsSharp != timeLaserCloudFullRes ||
                timeCornerPointsLessSharp != timeLaserCloudFullRes ||
                timeSurfPointsFlat != timeLaserCloudFullRes ||
                timeSurfPointsLessFlat != timeLaserCloudFullRes)
            {
                printf("unsync messeage!");
                ROS_BREAK();
            }
            mBuf.lock();
            cornerPointsSharp->clear();
            pcl::fromROSMsg(*cornerSharpBuf.front(), *cornerPointsSharp);
            cornerSharpBuf.pop();
            cornerPointsLessSharp->clear();
            pcl::fromROSMsg(*cornerLessSharpBuf.front(), *cornerPointsLessSharp);
            cornerLessSharpBuf.pop();
            surfPointsFlat->clear();
            pcl::fromROSMsg(*surfFlatBuf.front(), *surfPointsFlat);
            surfFlatBuf.pop();
            surfPointsLessFlat->clear();
            pcl::fromROSMsg(*surfLessFlatBuf.front(), *surfPointsLessFlat);
            surfLessFlatBuf.pop();
            laserCloudFullRes->clear();
            pcl::fromROSMsg(*fullPointsBuf.front(), *laserCloudFullRes);
            fullPointsBuf.pop();
            mBuf.unlock();
            TicToc t_whole;

            if (!systemInited)
            {
                systemInited = true;
                std::cout << "Initialization finished \n";
            }
            else
            {
                int cornerPointsSharpNum = cornerPointsSharp->points.size();
                int surfPointsFlatNum = surfPointsFlat->points.size();
                TicToc t_opt;
                for (size_t opti_counter = 0; opti_counter < 2; ++opti_counter)
                {
                    corner_correspondence = 0;
                    plane_correspondence = 0;

                    ceres::LossFunction *loss_function = new ceres::HuberLoss(0.1);
                    ceres::LocalParameterization *q_parameterization =
                        new ceres::EigenQuaternionParameterization();
                    ceres::Problem::Options problem_options;
                    ceres::Problem problem(problem_options);
```

```
                problem.AddParameterBlock(para_q, 4, q_parameterization);
                problem.AddParameterBlock(para_t, 3);
                pcl::PointXYZI pointSel;
                std::vector<int> pointSearchInd;
                std::vector<float> pointSearchSqDis;
                TicToc t_data;

                for (int i = 0; i < cornerPointsSharpNum; ++i)
                {
                    TransformToStart(&(cornerPointsSharp->points[i]), &pointSel);
                    kdtreeCornerLast->nearestKSearch(pointSel, 1, pointSearchInd, pointSearchSqDis);
                    int closestPointInd = -1, minPointInd2 = -1;
                    if (pointSearchSqDis[0] < DISTANCE_SQ_THRESHOLD)
                    {
                        closestPointInd = pointSearchInd[0];
                        int closestPointScanID = int(laserCloudCornerLast->points[closestPointInd].intensity);
                        double minPointSqDis2 = DISTANCE_SQ_THRESHOLD;

                        for (int j = closestPointInd + 1; j < (int)laserCloudCornerLast->points.size(); ++j)
                        {

                            if (int(laserCloudCornerLast->points[j].intensity) <= closestPointScanID)
                                continue;

                            if (int(laserCloudCornerLast->points[j].intensity) > (closestPointScanID + NEARBY_SCAN))
                                break;
                            double pointSqDis = (laserCloudCornerLast->points[j].x - pointSel.x) *
                                                    (laserCloudCornerLast->points[j].x - pointSel.x) +
                                                (laserCloudCornerLast->points[j].y - pointSel.y) *
                                                    (laserCloudCornerLast->points[j].y - pointSel.y) +
                                                (laserCloudCornerLast->points[j].z - pointSel.z) *
                                                    (laserCloudCornerLast->points[j].z - pointSel.z);
                            if (pointSqDis < minPointSqDis2)
                            {

                                minPointSqDis2 = pointSqDis;
                                minPointInd2 = j;
                            }
                        }

                        for (int j = closestPointInd - 1; j >= 0; --j)
                        {

                            if (int(laserCloudCornerLast->points[j].intensity) >= closestPointScanID)
                                continue;

                            if (int(laserCloudCornerLast->points[j].intensity) < (closestPointScanID - NEARBY_SCAN))
                                break;
                            double pointSqDis = (laserCloudCornerLast->points[j].x - pointSel.x) *
                                                    (laserCloudCornerLast->points[j].x - pointSel.x) +
                                                (laserCloudCornerLast->points[j].y - pointSel.y) *
                                                    (laserCloudCornerLast->points[j].y - pointSel.y) +
                                                (laserCloudCornerLast->points[j].z - pointSel.z) *
                                                    (laserCloudCornerLast->points[j].z - pointSel.z);
                            if (pointSqDis < minPointSqDis2)
                            {

                                minPointSqDis2 = pointSqDis;
                                minPointInd2 = j;
                            }
```

```
                        }
                    }
                    if (minPointInd2 >= 0)
                    {
                        Eigen::Vector3d curr_point(cornerPointsSharp->points[i].x,
                                                   cornerPointsSharp->points[i].y,
                                                   cornerPointsSharp->points[i].z);
                        Eigen::Vector3d last_point_a(laserCloudCornerLast->points[closestPointInd].x,
                                                     laserCloudCornerLast->points[closestPointInd].y,
                                                     laserCloudCornerLast->points[closestPointInd].z);
                        Eigen::Vector3d last_point_b(laserCloudCornerLast->points[minPointInd2].x,
                                                     laserCloudCornerLast->points[minPointInd2].y,
                                                     laserCloudCornerLast->points[minPointInd2].z);
                        double s;
                        if (DISTORTION)
                            s = (cornerPointsSharp->points[i].intensity - int(cornerPointsSharp->points[i].intensity)) / SCAN_PERIOD;
                        else
                            s = 1.0;
                        ceres::CostFunction *cost_function = LidarEdgeFactor::Create(curr_point, last_point_a, last_point_b, s);
                        problem.AddResidualBlock(cost_function, loss_function, para_q, para_t);
                        corner_correspondence++;
                    }
                }

                for (int i = 0; i < surfPointsFlatNum; ++i)
                {
                    TransformToStart(&(surfPointsFlat->points[i]), &pointSel);
                    kdtreeSurfLast->nearestKSearch(pointSel, 1, pointSearchInd, pointSearchSqDis);
                    int closestPointInd = -1, minPointInd2 = -1, minPointInd3 = -1;
                    if (pointSearchSqDis[0] < DISTANCE_SQ_THRESHOLD)
                    {
                        closestPointInd = pointSearchInd[0];

                        int closestPointScanID = int(laserCloudSurfLast->points[closestPointInd].intensity);
                        double minPointSqDis2 = DISTANCE_SQ_THRESHOLD, minPointSqDis3 = DISTANCE_SQ_THRESHOLD;

                        for (int j = closestPointInd + 1; j < (int)laserCloudSurfLast->points.size(); ++j)
                        {

                            if (int(laserCloudSurfLast->points[j].intensity) > (closestPointScanID + NEARBY_SCAN))
                                break;
                            double pointSqDis = (laserCloudSurfLast->points[j].x - pointSel.x) *
                                                    (laserCloudSurfLast->points[j].x - pointSel.x) +
                                                (laserCloudSurfLast->points[j].y - pointSel.y) *
                                                    (laserCloudSurfLast->points[j].y - pointSel.y) +
                                                (laserCloudSurfLast->points[j].z - pointSel.z) *
                                                    (laserCloudSurfLast->points[j].z - pointSel.z);

                            if (int(laserCloudSurfLast->points[j].intensity) <= closestPointScanID && pointSqDis < minPointSqDis2)
                            {
                                minPointSqDis2 = pointSqDis;
                                minPointInd2 = j;
                            }

                            else if (int(laserCloudSurfLast->points[j].intensity) > closestPointScanID && pointSqDis < minPointSqDis3)
                            {
```

```
                                minPointSqDis3 = pointSqDis;
                                minPointInd3 = j;
                            }
                        }

                        for (int j = closestPointInd - 1; j >= 0; --j)
                        {
                            if (int(laserCloudSurfLast->points[j].intensity) < (closestPointScanID - NEARBY_SCAN))
                                break;
                            double pointSqDis = (laserCloudSurfLast->points[j].x - pointSel.x) *
                                                    (laserCloudSurfLast->points[j].x - pointSel.x) +
                                                (laserCloudSurfLast->points[j].y - pointSel.y) *
                                                    (laserCloudSurfLast->points[j].y - pointSel.y) +
                                                (laserCloudSurfLast->points[j].z - pointSel.z) *
                                                    (laserCloudSurfLast->points[j].z - pointSel.z);

                            if (int(laserCloudSurfLast->points[j].intensity) >= closestPointScanID && pointSqDis < minPointSqDis2)
                            {
                                minPointSqDis2 = pointSqDis;
                                minPointInd2 = j;
                            }
                            else if (int(laserCloudSurfLast->points[j].intensity) < closestPointScanID && pointSqDis < minPointSqDis3)
                            {

                                minPointSqDis3 = pointSqDis;
                                minPointInd3 = j;
                            }
                        }
                        if (minPointInd2 >= 0 && minPointInd3 >= 0)
                        {
                            Eigen::Vector3d curr_point(surfPointsFlat->points[i].x,
                                                        surfPointsFlat->points[i].y,
                                                        surfPointsFlat->points[i].z);
                            Eigen::Vector3d last_point_a(laserCloudSurfLast->points[closestPointInd].x,
                                                            laserCloudSurfLast->points[closestPointInd].y,
                                                            laserCloudSurfLast->points[closestPointInd].z);
                            Eigen::Vector3d last_point_b(laserCloudSurfLast->points[minPointInd2].x,
                                                            laserCloudSurfLast->points[minPointInd2].y,
                                                            laserCloudSurfLast->points[minPointInd2].z);
                            Eigen::Vector3d last_point_c(laserCloudSurfLast->points[minPointInd3].x,
                                                            laserCloudSurfLast->points[minPointInd3].y,
                                                            laserCloudSurfLast->points[minPointInd3].z);
                            double s;
                            if (DISTORTION)
                                s = (surfPointsFlat->points[i].intensity - int(surfPointsFlat->points[i].intensity)) / SCAN_PERIOD;
                            else
                                s = 1.0;
                            ceres::CostFunction *cost_function = LidarPlaneFactor::Create(curr_point, last_point_a, last_point_b, last_point_c, s);
                            problem.AddResidualBlock(cost_function, loss_function, para_q, para_t);
                            plane_correspondence++;
                        }
                    }
                }

                printf("data association time %f ms \n", t_data.toc());
                if ((corner_correspondence + plane_correspondence) < 10)
                {
```

```
                        printf("less correspondence!
    *************************************************\n");
                    }
                    TicToc t_solver;
                    ceres::Solver::Options options;
                    options.linear_solver_type = ceres::DENSE_QR;
                    options.max_num_iterations = 4;
                    options.minimizer_progress_to_stdout = false;
                    ceres::Solver::Summary summary;
                    ceres::Solve(options, &problem, &summary);
                    printf("solver time %f ms \n", t_solver.toc());
                }
                printf("optimization twice time %f \n", t_opt.toc());
                t_w_curr = t_w_curr + q_w_curr * t_last_curr;
                q_w_curr = q_w_curr * q_last_curr;
            }
            TicToc t_pub;

            nav_msgs::Odometry laserOdometry;
            laserOdometry.header.frame_id = "/camera_init";
            laserOdometry.child_frame_id = "/laser_odom";
            laserOdometry.header.stamp = ros::Time().fromSec(timeSurfPointsLessFlat);
            laserOdometry.pose.pose.orientation.x = q_w_curr.x();
            laserOdometry.pose.pose.orientation.y = q_w_curr.y();
            laserOdometry.pose.pose.orientation.z = q_w_curr.z();
            laserOdometry.pose.pose.orientation.w = q_w_curr.w();
            laserOdometry.pose.pose.position.x = t_w_curr.x();
            laserOdometry.pose.pose.position.y = t_w_curr.y();
            laserOdometry.pose.pose.position.z = t_w_curr.z();
            pubLaserOdometry.publish(laserOdometry);
            geometry_msgs::PoseStamped laserPose;
            laserPose.header = laserOdometry.header;
            laserPose.pose = laserOdometry.pose.pose;
            laserPath.header.stamp = laserOdometry.header.stamp;
            laserPath.poses.push_back(laserPose);
            laserPath.header.frame_id = "/camera_init";
            pubLaserPath.publish(laserPath);

            if (0)
            {
                int cornerPointsLessSharpNum = cornerPointsLessSharp->points.size();
                for (int i = 0; i < cornerPointsLessSharpNum; i++)
                {
                    TransformToEnd(&cornerPointsLessSharp->points[i], &
    cornerPointsLessSharp->points[i]);
                }
                int surfPointsLessFlatNum = surfPointsLessFlat->points.size();
                for (int i = 0; i < surfPointsLessFlatNum; i++)
                {
                    TransformToEnd(&surfPointsLessFlat->points[i], &surfPointsLessFlat->
    points[i]);
                }
                int laserCloudFullResNum = laserCloudFullRes->points.size();
                for (int i = 0; i < laserCloudFullResNum; i++)
                {
                    TransformToEnd(&laserCloudFullRes->points[i], &laserCloudFullRes->
    points[i]);
                }
            }
            pcl::PointCloud<PointType>::Ptr laserCloudTemp = cornerPointsLessSharp;
            cornerPointsLessSharp = laserCloudCornerLast;
            laserCloudCornerLast = laserCloudTemp;
            laserCloudTemp = surfPointsLessFlat;
            surfPointsLessFlat = laserCloudSurfLast;
            laserCloudSurfLast = laserCloudTemp;
            laserCloudCornerLastNum = laserCloudCornerLast->points.size();
            laserCloudSurfLastNum = laserCloudSurfLast->points.size();

            kdtreeCornerLast->setInputCloud(laserCloudCornerLast);
            kdtreeSurfLast->setInputCloud(laserCloudSurfLast);
            if (frameCount % skipFrameNum == 0)
            {
                frameCount = 0;
                sensor_msgs::PointCloud2 laserCloudCornerLast2;
                pcl::toROSMsg(*laserCloudCornerLast, laserCloudCornerLast2);
                laserCloudCornerLast2.header.stamp = ros::Time().fromSec(
    timeSurfPointsLessFlat);
                laserCloudCornerLast2.header.frame_id = "/camera";
                pubLaserCloudCornerLast.publish(laserCloudCornerLast2);
```

```
                sensor_msgs::PointCloud2 laserCloudSurfLast2;
                pcl::toROSMsg(*laserCloudSurfLast, laserCloudSurfLast2);
                laserCloudSurfLast2.header.stamp = ros::Time().fromSec(
    timeSurfPointsLessFlat);
                laserCloudSurfLast2.header.frame_id = "/camera";
                pubLaserCloudSurfLast.publish(laserCloudSurfLast2);
                sensor_msgs::PointCloud2 laserCloudFullRes3;
                pcl::toROSMsg(*laserCloudFullRes, laserCloudFullRes3);
                laserCloudFullRes3.header.stamp = ros::Time().fromSec(
    timeSurfPointsLessFlat);
                laserCloudFullRes3.header.frame_id = "/camera";
                pubLaserCloudFullRes.publish(laserCloudFullRes3);
            }
            printf("publication time %f ms \n", t_pub.toc());
            printf("whole laserOdometry time %f ms \n \n", t_whole.toc());
            if(t_whole.toc() > 100)
                ROS_WARN("odometry process over 100ms");
            frameCount++;
        }
        rate.sleep();
    }
    return 0;
}
```

A.3.4 Feature extraction

```
//This code is created based on A-LOAM by Weichen WEI: weichen.wei@monash.edu
#include <cmath>
#include <vector>
#include <string>
#include "loam_itm/common.h"
#include "loam_itm/tic_toc.h"
#include <nav_msgs/Odometry.h>
#include <opencv/cv.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <ros/ros.h>
#include <sensor_msgs/Imu.h>
#include <sensor_msgs/PointCloud2.h>
#include <tf/transform_datatypes.h>
#include <tf/transform_broadcaster.h>
using std::atan2;
using std::cos;
using std::sin;
const double scanPeriod = 0.1;
const int systemDelay = 0;
int systemInitCount = 0;
bool systemInited = false;
int N_SCANS = 0;
float cloudCurvature[400000];
int cloudSortInd[400000];
int cloudNeighborPicked[400000];
int cloudLabel[400000];
bool comp (int i,int j) { return (cloudCurvature[i]<cloudCurvature[j]); }
ros::Publisher pubLaserCloud;
ros::Publisher pubCornerPointsSharp;
ros::Publisher pubCornerPointsLessSharp;
ros::Publisher pubSurfPointsFlat;
ros::Publisher pubSurfPointsLessFlat;
ros::Publisher pubRemovePoints;
std::vector<ros::Publisher> pubEachScan;
bool PUB_EACH_LINE = false;
double MINIMUM_RANGE = 0.1;
template <typename PointT>
void removeClosedPointCloud(const pcl::PointCloud<PointT> &cloud_in,
                              pcl::PointCloud<PointT> &cloud_out, float thres)
{
    if (&cloud_in != &cloud_out)
    {
        cloud_out.header = cloud_in.header;
        cloud_out.points.resize(cloud_in.points.size());
    }
    size_t j = 0;
    for (size_t i = 0; i < cloud_in.points.size(); ++i)
```

```
    {
        if (cloud_in.points[i].x * cloud_in.points[i].x + cloud_in.points[i].y * cloud_in
    .points[i].y + cloud_in.points[i].z * cloud_in.points[i].z < thres * thres)
            continue;
        cloud_out.points[j] = cloud_in.points[i];
        j++;
    }
    if (j != cloud_in.points.size())
    {
        cloud_out.points.resize(j);
    }
    cloud_out.height = 1;
    cloud_out.width = static_cast<uint32_t>(j);
    cloud_out.is_dense = true;
}
void laserCloudHandler(const sensor_msgs::PointCloud2ConstPtr &laserCloudMsg)
{
    if (!systemInited)
    {
        systemInitCount++;
        if (systemInitCount >= systemDelay)
        {
            systemInited = true;
        }
        else
            return;
    }
    TicToc t_whole;
    TicToc t_prepare;
    std::vector<int> scanStartInd(N_SCANS, 0);
    std::vector<int> scanEndInd(N_SCANS, 0);
    pcl::PointCloud<pcl::PointXYZ> laserCloudIn;
    pcl::fromROSMsg(*laserCloudMsg, laserCloudIn);
    std::vector<int> indices;
    pcl::removeNaNFromPointCloud(laserCloudIn, laserCloudIn, indices);
    removeClosedPointCloud(laserCloudIn, laserCloudIn, MINIMUM_RANGE);

    int cloudSize = laserCloudIn.points.size();
    float startOri = -atan2(laserCloudIn.points[0].y, laserCloudIn.points[0].x);
    float endOri = -atan2(laserCloudIn.points[cloudSize - 1].y,
                          laserCloudIn.points[cloudSize - 1].x) +
                   2 * M_PI;
    if (endOri - startOri > 3 * M_PI)
    {
        endOri -= 2 * M_PI;
    }
    else if (endOri - startOri < M_PI)
    {
        endOri += 2 * M_PI;
    }

    bool halfPassed = false;
    int count = cloudSize;
    PointType point;
    std::vector<pcl::PointCloud<PointType>> laserCloudScans(N_SCANS);
    for (int i = 0; i < cloudSize; i++)
    {
        point.x = laserCloudIn.points[i].x;
        point.y = laserCloudIn.points[i].y;
        point.z = laserCloudIn.points[i].z;
        float angle = atan(point.z / sqrt(point.x * point.x + point.y * point.y)) * 180 /
     M_PI;
        int scanID = 0;
        if (N_SCANS == 16)
        {
            scanID = int((angle + 15) / 2 + 0.5);
            if (scanID > (N_SCANS - 1) || scanID < 0)
            {
                count--;
                continue;
            }
        }
        else if (N_SCANS == 32)
        {
            scanID = int((angle + 92.0/3.0) * 3.0 / 4.0);
            if (scanID > (N_SCANS - 1) || scanID < 0)
            {
                count--;
                continue;
```

```
            }
        }
        else if (N_SCANS == 64)
        {
            if (angle >= -8.83)
                scanID = int((2 - angle) * 3.0 + 0.5);
            else
                scanID = N_SCANS / 2 + int((-8.83 - angle) * 2.0 + 0.5);

            if (angle > 2 || angle < -24.33 || scanID > 50 || scanID < 0)
            {
                count--;
                continue;
            }
        }
        else
        {
            printf("wrong scan number\n");
            ROS_BREAK();
        }

        float ori = -atan2(point.y, point.x);
        if (!halfPassed)
        {
            if (ori < startOri - M_PI / 2)
            {
                ori += 2 * M_PI;
            }
            else if (ori > startOri + M_PI * 3 / 2)
            {
                ori -= 2 * M_PI;
            }
            if (ori - startOri > M_PI)
            {
                halfPassed = true;
            }
        }
        else
        {
            ori += 2 * M_PI;
            if (ori < endOri - M_PI * 3 / 2)
            {
                ori += 2 * M_PI;
            }
            else if (ori > endOri + M_PI / 2)
            {
                ori -= 2 * M_PI;
            }
        }
        float relTime = (ori - startOri) / (endOri - startOri);
        point.intensity = scanID + scanPeriod * relTime;
        laserCloudScans[scanID].push_back(point);
    }

    cloudSize = count;
    printf("points size %d \n", cloudSize);
    pcl::PointCloud<PointType>::Ptr laserCloud(new pcl::PointCloud<PointType>());
    for (int i = 0; i < N_SCANS; i++)
    {
        scanStartInd[i] = laserCloud->size() + 5;
        *laserCloud += laserCloudScans[i];
        scanEndInd[i] = laserCloud->size() - 6;
    }
    printf("prepare time %f \n", t_prepare.toc());
    for (int i = 5; i < cloudSize - 5; i++)
    {
        float diffX = laserCloud->points[i - 5].x + laserCloud->points[i - 4].x + laserCloud->points[i - 3].x + laserCloud->points[i - 2].x + laserCloud->points[i - 1].x - 10 * laserCloud->points[i].x + laserCloud->points[i + 1].x + laserCloud->points[i + 2].x + laserCloud->points[i + 3].x + laserCloud->points[i + 4].x + laserCloud->points[i + 5].x;
        float diffY = laserCloud->points[i - 5].y + laserCloud->points[i - 4].y + laserCloud->points[i - 3].y + laserCloud->points[i - 2].y + laserCloud->points[i - 1].y - 10 * laserCloud->points[i].y + laserCloud->points[i + 1].y + laserCloud->points[i + 2].y + laserCloud->points[i + 3].y + laserCloud->points[i + 4].y + laserCloud->points[i + 5].y;
        float diffZ = laserCloud->points[i - 5].z + laserCloud->points[i - 4].z + laserCloud->points[i - 3].z + laserCloud->points[i - 2].z + laserCloud->points[i -
```

```
1].z - 10 * laserCloud->points[i].z + laserCloud->points[i + 1].z + laserCloud->
points[i + 2].z + laserCloud->points[i + 3].z + laserCloud->points[i + 4].z +
laserCloud->points[i + 5].z;
        cloudCurvature[i] = diffX * diffX + diffY * diffY + diffZ * diffZ;
        cloudSortInd[i] = i;
        cloudNeighborPicked[i] = 0;
        cloudLabel[i] = 0;
    }

    TicToc t_pts;
    pcl::PointCloud<PointType> cornerPointsSharp;
    pcl::PointCloud<PointType> cornerPointsLessSharp;
    pcl::PointCloud<PointType> surfPointsFlat;
    pcl::PointCloud<PointType> surfPointsLessFlat;
    float t_q_sort = 0;
    for (int i = 0; i < N_SCANS; i++)
    {
        if( scanEndInd[i] - scanStartInd[i] < 6)
            continue;
        pcl::PointCloud<PointType>::Ptr surfPointsLessFlatScan(new pcl::PointCloud<
PointType>);
        for (int j = 0; j < 6; j++)
        {
            int sp = scanStartInd[i] + (scanEndInd[i] - scanStartInd[i]) * j / 6;
            int ep = scanStartInd[i] + (scanEndInd[i] - scanStartInd[i]) * (j + 1) / 6 -
1;
            TicToc t_tmp;
            std::sort (cloudSortInd + sp, cloudSortInd + ep + 1, comp);
            t_q_sort += t_tmp.toc();
            int largestPickedNum = 0;
            for (int k = ep; k >= sp; k--)
            {
                int ind = cloudSortInd[k];
                if (cloudNeighborPicked[ind] == 0 &&
                    cloudCurvature[ind] > 0.1)
                {
                    largestPickedNum++;
                    if (largestPickedNum <= 2)
                    {
                        cloudLabel[ind] = 2;
                        cornerPointsSharp.push_back(laserCloud->points[ind]);
                        cornerPointsLessSharp.push_back(laserCloud->points[ind]);
                    }
                    else if (largestPickedNum <= 20)
                    {
                        cloudLabel[ind] = 1;
                        cornerPointsLessSharp.push_back(laserCloud->points[ind]);
                    }
                    else
                    {
                        break;
                    }
                    cloudNeighborPicked[ind] = 1;
                    for (int l = 1; l <= 5; l++)
                    {
                        float diffX = laserCloud->points[ind + l].x - laserCloud->points[
ind + l - 1].x;
                        float diffY = laserCloud->points[ind + l].y - laserCloud->points[
ind + l - 1].y;
                        float diffZ = laserCloud->points[ind + l].z - laserCloud->points[
ind + l - 1].z;
                        if (diffX * diffX + diffY * diffY + diffZ * diffZ > 0.05)
                        {
                            break;
                        }
                        cloudNeighborPicked[ind + l] = 1;
                    }
                    for (int l = -1; l >= -5; l--)
                    {
                        float diffX = laserCloud->points[ind + l].x - laserCloud->points[
ind + l + 1].x;
                        float diffY = laserCloud->points[ind + l].y - laserCloud->points[
ind + l + 1].y;
                        float diffZ = laserCloud->points[ind + l].z - laserCloud->points[
ind + l + 1].z;
                        if (diffX * diffX + diffY * diffY + diffZ * diffZ > 0.05)
                        {
                            break;
```

```
                        }
                        cloudNeighborPicked[ind + l] = 1;
                    }
                }
            }
            int smallestPickedNum = 0;
            for (int k = sp; k <= ep; k++)
            {
                int ind = cloudSortInd[k];
                if (cloudNeighborPicked[ind] == 0 &&
                    cloudCurvature[ind] < 0.1)
                {
                    cloudLabel[ind] = -1;
                    surfPointsFlat.push_back(laserCloud->points[ind]);
                    smallestPickedNum++;
                    if (smallestPickedNum >= 4)
                    {
                        break;
                    }
                    cloudNeighborPicked[ind] = 1;
                    for (int l = 1; l <= 5; l++)
                    {
                        float diffX = laserCloud->points[ind + l].x - laserCloud->points[
    ind + l - 1].x;
                        float diffY = laserCloud->points[ind + l].y - laserCloud->points[
    ind + l - 1].y;
                        float diffZ = laserCloud->points[ind + l].z - laserCloud->points[
    ind + l - 1].z;
                        if (diffX * diffX + diffY * diffY + diffZ * diffZ > 0.05)
                        {
                            break;
                        }
                        cloudNeighborPicked[ind + l] = 1;
                    }
                    for (int l = -1; l >= -5; l--)
                    {
                        float diffX = laserCloud->points[ind + l].x - laserCloud->points[
    ind + l + 1].x;
                        float diffY = laserCloud->points[ind + l].y - laserCloud->points[
    ind + l + 1].y;
                        float diffZ = laserCloud->points[ind + l].z - laserCloud->points[
    ind + l + 1].z;
                        if (diffX * diffX + diffY * diffY + diffZ * diffZ > 0.05)
                        {
                            break;
                        }
                        cloudNeighborPicked[ind + l] = 1;
                    }
                }
            }
            for (int k = sp; k <= ep; k++)
            {
                if (cloudLabel[k] <= 0)
                {
                    surfPointsLessFlatScan->push_back(laserCloud->points[k]);
                }
            }
        }
        pcl::PointCloud<PointType> surfPointsLessFlatScanDS;
        pcl::VoxelGrid<PointType> downSizeFilter;
        downSizeFilter.setInputCloud(surfPointsLessFlatScan);
        downSizeFilter.setLeafSize(0.2, 0.2, 0.2);
        downSizeFilter.filter(surfPointsLessFlatScanDS);
        surfPointsLessFlat += surfPointsLessFlatScanDS;
    }
    printf("sort q time %f \n", t_q_sort);
    printf("seperate points time %f \n", t_pts.toc());

    sensor_msgs::PointCloud2 laserCloudOutMsg;
    pcl::toROSMsg(*laserCloud, laserCloudOutMsg);
    laserCloudOutMsg.header.stamp = laserCloudMsg->header.stamp;
    laserCloudOutMsg.header.frame_id = "/camera_init";
    pubLaserCloud.publish(laserCloudOutMsg);
    sensor_msgs::PointCloud2 cornerPointsSharpMsg;
    pcl::toROSMsg(cornerPointsSharp, cornerPointsSharpMsg);
    cornerPointsSharpMsg.header.stamp = laserCloudMsg->header.stamp;
    cornerPointsSharpMsg.header.frame_id = "/camera_init";
```

```
    pubCornerPointsSharp.publish(cornerPointsSharpMsg);
    sensor_msgs::PointCloud2 cornerPointsLessSharpMsg;
    pcl::toROSMsg(cornerPointsLessSharp, cornerPointsLessSharpMsg);
    cornerPointsLessSharpMsg.header.stamp = laserCloudMsg->header.stamp;
    cornerPointsLessSharpMsg.header.frame_id = "/camera_init";
    pubCornerPointsLessSharp.publish(cornerPointsLessSharpMsg);
    sensor_msgs::PointCloud2 surfPointsFlat2;
    pcl::toROSMsg(surfPointsFlat, surfPointsFlat2);
    surfPointsFlat2.header.stamp = laserCloudMsg->header.stamp;
    surfPointsFlat2.header.frame_id = "/camera_init";
    pubSurfPointsFlat.publish(surfPointsFlat2);
    sensor_msgs::PointCloud2 surfPointsLessFlat2;
    pcl::toROSMsg(surfPointsLessFlat, surfPointsLessFlat2);
    surfPointsLessFlat2.header.stamp = laserCloudMsg->header.stamp;
    surfPointsLessFlat2.header.frame_id = "/camera_init";
    pubSurfPointsLessFlat.publish(surfPointsLessFlat2);

    if(PUB_EACH_LINE)
    {
        for(int i = 0; i< N_SCANS; i++)
        {
            sensor_msgs::PointCloud2 scanMsg;
            pcl::toROSMsg(laserCloudScans[i], scanMsg);
            scanMsg.header.stamp = laserCloudMsg->header.stamp;
            scanMsg.header.frame_id = "/camera_init";
            pubEachScan[i].publish(scanMsg);
        }
    }
    printf("scan registration time %f ms *************\n", t_whole.toc());
    if(t_whole.toc() > 100)
        ROS_WARN("scan registration process over 100ms");
}
int main(int argc, char **argv)
{
    ros::init(argc, argv, "scanRegistration");
    ros::NodeHandle nh;
    nh.param<int>("scan_line", N_SCANS, 16);
    nh.param<double>("minimum_range", MINIMUM_RANGE, 0.1);
    printf("scan line number %d \n", N_SCANS);
    if(N_SCANS != 16 && N_SCANS != 32 && N_SCANS != 64)
    {
        printf("only support velodyne with 16, 32 or 64 scan line!");
        return 0;
    }
    ros::Subscriber subLaserCloud = nh.subscribe<sensor_msgs::PointCloud2>("/velodyne_points", 100, laserCloudHandler);
    pubLaserCloud = nh.advertise<sensor_msgs::PointCloud2>("/velodyne_cloud_2", 100);
    pubCornerPointsSharp = nh.advertise<sensor_msgs::PointCloud2>("/laser_cloud_sharp", 100);
    pubCornerPointsLessSharp = nh.advertise<sensor_msgs::PointCloud2>("/laser_cloud_less_sharp", 100);
    pubSurfPointsFlat = nh.advertise<sensor_msgs::PointCloud2>("/laser_cloud_flat", 100);
    pubSurfPointsLessFlat = nh.advertise<sensor_msgs::PointCloud2>("/laser_cloud_less_flat", 100);
    pubRemovePoints = nh.advertise<sensor_msgs::PointCloud2>("/laser_remove_points", 100);
    if(PUB_EACH_LINE)
    {
        for(int i = 0; i < N_SCANS; i++)
        {
            ros::Publisher tmp = nh.advertise<sensor_msgs::PointCloud2>("/laser_scanid_" + std::to_string(i), 100);
            pubEachScan.push_back(tmp);
        }
    }
    ros::spin();
    return 0;
}
```

A.4 Dual LiDAR LOAM Codes

This section documented the source code of modified LOAM SLAM described in Chapter 5. The functions listed mainly include the 2D point cloud feature extraction functions and 3D point cloud segmentation functions. The modifications of the original LOAM is also documented.

A.4.1 2D LiDAR point cloud feature extraction

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <cmath>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/LaserScan.h>
#include <string>
#include <vector>
#include <mloam/Section.h>
#include <mloam/SectionMsg.h>
class Hokuyo_Proc
{
    private:
        struct section
            {
                int section_id;
                int start_point;
                int end_point;
                int feature_flag;
                int icon_point;
            };

        ros::NodeHandle nh;
        ros::Subscriber sub_hokuyo_msg;
        ros::Publisher pub_hokuyo_proc;
        ros::Publisher pub_hokuyo_plane;
        ros::Publisher pub_hokuyo_corner;
        pcl::PointCloud<pcl::PointXYZINormal> hokuyo_points;
        std::vector<std::pair<int, float>> curvature;
        std::vector<section> sections;
        const int corner_num = 10;
        const int plane_num = 5;
        const int section_num = corner_num + plane_num;//num of sections the scan will be devided

        static bool sortbydiff(const std::pair<int,float> &a, const std::pair<int,float> &b) {
            return (a.second < b.second);
        }
    public:
        Hokuyo_Proc()
        {
            sub_hokuyo_msg = nh.subscribe("/scan", 100, &Hokuyo_Proc::hokuyo_msg_callback, this);
            pub_hokuyo_plane = nh.advertise<sensor_msgs::PointCloud2>("hokuyo_plane", 1, this);
            pub_hokuyo_corner = nh.advertise<sensor_msgs::PointCloud2>("hokuyo_corner", 1, this);
            pub_hokuyo_proc = nh.advertise<mloam::SectionMsg>("hokuyo_sections", 1, this);
        }


        void hokuyo_msg_callback(const sensor_msgs::LaserScan& scan){
            double newPointAngle;
            pcl::PointXYZINormal newPoint;
            // ROS_INFO("Num: %i", scan.ranges.size());
            for(size_t i=0; i<scan.ranges.size(); i++){
                newPointAngle = scan.angle_min + scan.angle_increment * i;
                newPoint.x = scan.ranges[i] * cos(newPointAngle);
                newPoint.y = scan.ranges[i] * sin(newPointAngle);
                newPoint.z = 0.0;
```

```
                // newPoint.intensity = scan.intensities[i];//TODO update urg_node
                hokuyo_points.push_back(newPoint);
            }
            // ROS_INFO("Num: %i", hokuyo_points.points.size());
            hokuyo_line_filter(hokuyo_points, scan.header.stamp);
            hokuyo_points.clear();
        }
        void hokuyo_line_filter(const pcl::PointCloud<pcl::PointXYZINormal>& hokuyo_points, const ros::Time timestamp){
            //matching FOV with Livox
            float fov_ratio = 38.4/180;
            int view_range = hokuyo_points.size() * fov_ratio;// num of points in hokuyo scan which are in Livox FOV
            int starting  = (hokuyo_points.size()/2)-(view_range/2);//starting point in hokuyo matching livox FOV
            int ending  = (hokuyo_points.size()/2)+(view_range/2);
            int section_size = (ending-starting)/section_num;
            for(size_t i=starting; i<ending; i++){
                float diffX = hokuyo_points.points[ i - 5 ].x + hokuyo_points.points[ i - 4 ].x + hokuyo_points.points[ i - 3 ].x + hokuyo_points.points[ i - 2 ].x + hokuyo_points.points[ i - 1 ].x - 10 * hokuyo_points.points[ i ].x + hokuyo_points.points[ i + 1 ].x + hokuyo_points.points[ i + 2 ].x + hokuyo_points.points[ i + 3 ].x + hokuyo_points.points[ i + 4 ].x + hokuyo_points.points[ i + 5 ].x;
                float diffY = hokuyo_points.points[ i - 5 ].y + hokuyo_points.points[ i - 4 ].y + hokuyo_points.points[ i - 3 ].y + hokuyo_points.points[ i - 2 ].y + hokuyo_points.points[ i - 1 ].y - 10 * hokuyo_points.points[ i ].y + hokuyo_points.points[ i + 1 ].y + hokuyo_points.points[ i + 2 ].y + hokuyo_points.points[ i + 3 ].y + hokuyo_points.points[ i + 4 ].y + hokuyo_points.points[ i + 5 ].y;
                // float diffZ = hokuyo_points.points[ i - 5 ].z + hokuyo_points.points[ i - 4 ].z + hokuyo_points.points[ i - 3 ].z + hokuyo_points.points[ i - 2 ].z + hokuyo_points.points[ i - 1 ].z - 10 * hokuyo_points.points[ i ].z + hokuyo_points.points[ i + 1 ].z + hokuyo_points.points[ i + 2 ].z + hokuyo_points.points[ i + 3 ].z + hokuyo_points.points[ i + 4 ].z + hokuyo_points.points[ i + 5 ].z;
                float diff = diffX * diffX + diffY * diffY;
                curvature.push_back(std::make_pair(i,diff));
            }
            sort(curvature.begin(), curvature.end(), Hokuyo_Proc::sortbydiff);
            // ROS_INFO("Num: %i", curvature.size());
            for (int s = 0; s < section_num; s++){
                    // ROS_INFO("Point ID: %i, Curvature: %f", curvature[i].first, curvature[i].second);
                    section current_sec;
                    current_sec.section_id = s;
                    current_sec.start_point = s * section_size;
                    current_sec.end_point = (s + 1) * section_size;
                    current_sec.feature_flag = 0; //0 not asigned, 1 plane, 2 corner.
                    sections.push_back(current_sec);
            }
            int plane_num_copy = plane_num;
            for (int i = 0; i < plane_num_copy; i++){
                int sec_index = (curvature[i].first-starting)/section_size;
                // ROS_INFO("SecID: %i, SecSize: %i, PointID: %i, Starting: %i, plane_num_copy %i", sec_index, section_size, curvature[i].first, starting, plane_num_copy);
                if (sections[sec_index].feature_flag == 0){
                    sections[sec_index].feature_flag = 1;
                    sections[sec_index].icon_point = curvature[i].first;
                }else{
                    plane_num_copy ++;
                }
            }
            int corner_num_copy = corner_num;
            for (int i = 0; i < corner_num_copy; i++){
                int sec_index = (curvature[curvature.size()-i-1].first-starting)/section_size;
                // ROS_INFO("SecID: %i, SecSize: %i, PointID: %i, Starting: %i, plane_num_copy %i", sec_index, section_size, curvature[i].first, starting, corner_num_copy);
                if (sections[sec_index].feature_flag == 0){
                    sections[sec_index].feature_flag = 2;
                    sections[sec_index].icon_point = curvature[curvature.size()-i-1].first;
                }else{
                    corner_num_copy ++;
                }
            }
            pcl::PointCloud<pcl::PointXYZINormal> hokuyo_planes;
```

```
            pcl::PointCloud<pcl::PointXYZINormal> hokuyo_corners;
            pcl::PointCloud<pcl::PointXYZINormal> hokuyo_all;
            for (int i = 0; i < section_num;i++){
                if(sections[i].feature_flag==1){
                    for(int j = (sections[i].icon_point-5);j<= sections[i].icon_point+5;j++){
                        pcl::PointXYZINormal temp_point;
                        temp_point = hokuyo_points.points[j];
                        temp_point.intensity = i;
                        hokuyo_planes.push_back(temp_point);
                        // hokuyo_planes.push_back(hokuyo_points.points[j]);
                    }
                }
                if(sections[i].feature_flag==2){
                    pcl::PointXYZINormal temp_point;
                    temp_point = hokuyo_points.points[sections[i].icon_point];
                    temp_point.intensity = i;
                    hokuyo_corners.push_back(temp_point);
                    // hokuyo_corners.push_back(hokuyo_points.points[sections[i].icon_point]);
                }
                hokuyo_all.push_back(hokuyo_points.points[sections[i].icon_point]);
            }
            // ROS_INFO("Number of Sections: %i", sections.size());
            sensor_msgs::PointCloud2 pcl_ros_msg1;
            pcl::toROSMsg(hokuyo_planes, pcl_ros_msg1);
            pcl_ros_msg1.header.stamp = timestamp;
            pcl_ros_msg1.header.frame_id = "laser";
            pub_hokuyo_plane.publish(pcl_ros_msg1);
            sensor_msgs::PointCloud2 pcl_ros_msg2;
            pcl::toROSMsg(hokuyo_corners, pcl_ros_msg2);
            pcl_ros_msg2.header.stamp = timestamp;
            pcl_ros_msg2.header.frame_id = "laser";
            pub_hokuyo_corner.publish(pcl_ros_msg2);
            mloam::SectionMsg sections_msg;
            for(int i = 0; i < sections.size();i++){
                mloam::Section new_section;
                new_section.sx = hokuyo_points.points[sections[i].start_point].x;
                new_section.sy = hokuyo_points.points[sections[i].start_point].y;
                new_section.sz = hokuyo_points.points[sections[i].start_point].z;
                new_section.ex = hokuyo_points.points[sections[i].end_point].x;
                new_section.ey = hokuyo_points.points[sections[i].end_point].y;
                new_section.ez = hokuyo_points.points[sections[i].end_point].z;
                new_section.type = sections[i].feature_flag;
                new_section.section_id = sections[i].section_id;
                sections_msg.sections.push_back(new_section);
            }
            sections_msg.header.stamp = timestamp;
            sections_msg.header.frame_id = "laser";
            sections_msg.section_size = sections.size();
            pub_hokuyo_proc.publish(sections_msg);
            sections.clear();
            curvature.clear();
        }
};
int main(int argc, char **argv)
{
    ros::init(argc, argv, "hokuyo_reciver");

    Hokuyo_Proc HP;
    ros::spin();
    return 0;
}
```

A.4.2 3D LiDAR point cloud feature extraction

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <std_msgs/Int32.h>
#include <mloam/Section.h>
#include <mloam/SectionMsg.h>
#include <cmath>
#include <nav_msgs/Odometry.h>
#include <pcl/point_types.h>
#include <pcl_conversions/pcl_conversions.h>
#include <string>
```

```cpp
#include <vector>
#include <tf2_ros/static_transform_broadcaster.h>
#include <tf2/LinearMath/Quaternion.h>

class Livox_Proc
{

    private:
        struct section
            {
                int section_id;
                pcl::PointXYZ start_point;
                pcl::PointXYZ end_point;
                int feature_flag;
            };
        ros::NodeHandle nh;
        ros::Subscriber sub_livox_msg;
        ros::Subscriber sub_hokuyo_section_info;
        ros::Subscriber sub_livox_joint;
        ros::Publisher pub_livox_proc;
        ros::Publisher pub_livox_corner_proc;
        ros::Publisher pub_livox_surface_proc;
        // std::mutex mp, mc, ma;
        pcl::PointCloud<pcl::PointXYZI> livox_data;
        // std::vector<pcl::PointCloud<pcl::PointXYZI>> livox_clouds_in_sections;
        std::vector<std::vector<pcl::PointCloud<pcl::PointXYZI>>> livox_clouds_in_sections;
        std::vector<pcl::PointCloud<pcl::PointXYZI>> livox_plane_in_sections;
        std::vector<pcl::PointCloud<pcl::PointXYZI>> livox_corner_in_sections;
        std::vector<section> sections;
        int section_num=0;
        float rot;
        bool section_flag =  false;
    public:
        Livox_Proc()
        {
            sub_livox_msg = nh.subscribe<sensor_msgs::PointCloud2>("/livox/lidar", 100, &Livox_Proc::livox_msg_callback, this);
            sub_hokuyo_section_info = nh.subscribe<mloam::SectionMsg>("hokuyo_sections", 100, &Livox_Proc::hokuyo_section_callback, this);
            sub_livox_joint = nh.subscribe("livox_joint", 100, &Livox_Proc::joint_callback, this);
            pub_livox_proc = nh.advertise<sensor_msgs::PointCloud2>("pc2_full", 1, this);
            pub_livox_corner_proc = nh.advertise<sensor_msgs::PointCloud2>("pc2_corners", 1, this);
            pub_livox_surface_proc = nh.advertise<sensor_msgs::PointCloud2>("pc2_surface", 1, this);
        }
        void hokuyo_section_callback(const mloam::SectionMsg hokuyo_sections)
        {
            sections.clear();
            livox_plane_in_sections.clear();
            livox_clouds_in_sections.clear();
            livox_corner_in_sections.clear();
            section_num = hokuyo_sections.section_size;
            for(int i = 0; i < hokuyo_sections.section_size; i ++){
                section newSection;
                newSection.section_id = i;
                newSection.start_point.x = hokuyo_sections.sections[i].sx;
                newSection.start_point.y = hokuyo_sections.sections[i].sy;
                newSection.end_point.x = hokuyo_sections.sections[i].ex;
                newSection.end_point.y = hokuyo_sections.sections[i].ey;
                newSection.feature_flag = hokuyo_sections.sections[i].type;
                sections.push_back(newSection);
            }
            // ROS_INFO("Section Size %i", sections.size());
            livox_clouds_in_sections.resize(section_num);//resize to section number
            livox_plane_in_sections.resize(section_num);
            livox_corner_in_sections.resize(section_num);
            pcl::PointCloud<pcl::PointXYZI>::Ptr livox_section(new pcl::PointCloud<pcl::PointXYZI>()),
                                                     livox_plane(new pcl::PointCloud<pcl::PointXYZI>()),
                                                     livox_corner(new pcl::PointCloud<pcl::PointXYZI>());
            for (int i = 0; i < section_num; i++){
                // livox_clouds_in_sections[i] = *livox_section;
                livox_plane_in_sections[i] = *livox_plane;
                livox_corner_in_sections[i] = *livox_corner;
            }
```

```
        section_flag = true;
    }

    void joint_callback(const std_msgs::Int32& joint_pos)
    {
        int joint = joint_pos.data;
        // rot = (joint-2048)*((float)360/(float)4096);
        // ROS_INFO("Degree: %f", rot);
        // double q_curr[4];

        // // AngleAxisToQuaternion(cere_r_t, q_curr);
        float alg_rad_z = (joint-2048)*((2*M_PI)/(float)4096);
        static tf2_ros::StaticTransformBroadcaster livox_head;
        geometry_msgs::TransformStamped livox_transformStamped;
        livox_transformStamped.header.stamp = ros::Time::now();
        livox_transformStamped.header.frame_id = "camera_init";
        livox_transformStamped.child_frame_id = "livox_frame";
        livox_transformStamped.transform.translation.x = 0;
        livox_transformStamped.transform.translation.y = -0.02;
        livox_transformStamped.transform.translation.z = 0.18;
        tf2::Quaternion q_curr;
        // ROS_INFO("ROT: %f", alg_rad_z);
        q_curr.setRPY(0, 0, alg_rad_z);

        livox_transformStamped.transform.rotation.x = q_curr.x();
        livox_transformStamped.transform.rotation.y = q_curr.y();
        livox_transformStamped.transform.rotation.z = q_curr.z();
        livox_transformStamped.transform.rotation.w = q_curr.w();
        livox_head.sendTransform(livox_transformStamped);
    }
    void livox_msg_callback(const sensor_msgs::PointCloud2ConstPtr& livox_msg_in)
    {
        if (section_flag == false){
            return;
        }
        livox_data.clear();
        pcl::fromROSMsg(*livox_msg_in, livox_data);

        int cloudSize = livox_data.points.size();
        // ROS_INFO("Num of Points: %i", cloudSize);
        if(cloudSize == 0){
            return;
        }
        if(cloudSize > 32000) cloudSize = 32000;
        int count = cloudSize;
        // pcl::PointXYZI point;

        for (int i = 0; i < cloudSize-1; i++) {
            int scan_id;
            if (!pcl_isfinite(livox_data.points[i].x) || !pcl_isfinite(livox_data.points[i].y) || !pcl_isfinite(livox_data.points[i].z)) {
                continue;
            }
            // point.x = livox_data.points[i].x;
            // point.y = livox_data.points[i].y;
            // point.z = livox_data.points[i].z;
            double theta = std::atan2(livox_data.points[i].y,livox_data.points[i].x) / M_PI * 180;
            // float dis = livox_data.points[i].x * livox_data.points[i].x + livox_data.points[i].y * livox_data.points[i].y + livox_data.points[i].z * livox_data.points[i].z;
            // double dis2 = livox_data.points[i].x * livox_data.points[i].x + livox_data.points[i].y * livox_data.points[i].y;//XZY
            // double theta2 = std::asin(sqrt(dis2/dis)) / M_PI * 180;//X
            // ROS_INFO("theta 1 2: %f, %f", theta, theta2);
            // if (theta == 0.000000){
            //     continue;
            // }
            scan_id = theta/(38.4/section_num);//ID
            livox_data.points[i].intensity = scan_id;
            int per_scan_id = scan_id;
            pcl::PointCloud<pcl::PointXYZI> temp_sect;
            while(scan_id == per_scan_id && i<cloudSize-1){
                temp_sect.push_back(livox_data.points[i]);
                i++;
                theta = std::atan2(livox_data.points[i].y,livox_data.points[i].x) / M_PI * 180;
                scan_id = theta/(38.4/section_num);
                livox_data.points[i].intensity = (scan_id==per_scan_id) ? per_scan_id
```

```
 : scan_id;
                }
                livox_clouds_in_sections[per_scan_id+std::floor(section_num/2)].push_back(temp_sect);
                // ROS_INFO("scan_id: %i, %f", scan_id, theta);
                // livox_data.points[i].intensity = scan_id+(livox_data.points[i].intensity/10000);//ID,
                // livox_data.points[i].intensity = scan_id+(double(i)/cloudSize);

                // livox_clouds_in_sections[scan_id+(std::floor(section_num/2))].push_back(livox_data.points[i]);
            }
            // ROS_INFO("Index: %d, %d, %d", livox_clouds_in_sections.size(), livox_clouds_in_sections[0].size(), livox_clouds_in_sections[0][0].size());
            for(int i = 0;i < sections.size();i++){
                // ROS_INFO("Index i: %d", i);
                if(sections[i].feature_flag==1){
                    for(int j = 0; j< livox_clouds_in_sections[i].size();j++){
                        // ROS_INFO("Index j: %d, %d, %d", j, livox_clouds_in_sections[i].size(), livox_clouds_in_sections[i][j].size());
                        if(livox_clouds_in_sections[i][j].size()<6){
                            continue;
                        }
                        for(int m = 5; m<livox_clouds_in_sections[i][j].size()-5;m++ ){
                            // ROS_INFO("Index m: %d", m);
                            float diffX = livox_clouds_in_sections[i][j][m - 5 ].x + livox_clouds_in_sections[i][j][m - 4 ].x + livox_clouds_in_sections[i][j][m - 3 ].x + livox_clouds_in_sections[i][j][m - 2 ].x + livox_clouds_in_sections[i][j][m - 1 ].x - 10 * livox_clouds_in_sections[i][j][m].x + livox_clouds_in_sections[i][j][m + 1 ].x + livox_clouds_in_sections[i][j][m + 2 ].x + livox_clouds_in_sections[i][j][m + 3 ].x + livox_clouds_in_sections[i][j][m + 4 ].x + livox_clouds_in_sections[i][j][m + 5 ].x;
                            float diffY = livox_clouds_in_sections[i][j][m - 5 ].y + livox_clouds_in_sections[i][j][m - 4 ].y + livox_clouds_in_sections[i][j][m - 3 ].y + livox_clouds_in_sections[i][j][m - 2 ].y + livox_clouds_in_sections[i][j][m - 1 ].y - 10 * livox_clouds_in_sections[i][j][m].y + livox_clouds_in_sections[i][j][m + 1 ].y + livox_clouds_in_sections[i][j][m + 2 ].y + livox_clouds_in_sections[i][j][m + 3 ].y + livox_clouds_in_sections[i][j][m + 4 ].y + livox_clouds_in_sections[i][j][m + 5 ].y;
                            float diffZ = livox_clouds_in_sections[i][j][m - 5 ].z + livox_clouds_in_sections[i][j][m - 4 ].z + livox_clouds_in_sections[i][j][m - 3 ].z + livox_clouds_in_sections[i][j][m - 2 ].z + livox_clouds_in_sections[i][j][m - 1 ].z - 10 * livox_clouds_in_sections[i][j][m].z + livox_clouds_in_sections[i][j][m + 1 ].z + livox_clouds_in_sections[i][j][m + 2 ].z + livox_clouds_in_sections[i][j][m + 3 ].z + livox_clouds_in_sections[i][j][m + 4 ].z + livox_clouds_in_sections[i][j][m + 5 ].z;
                            float diff = diffX * diffX + diffY * diffY;
                            // ROS_INFO("Index: %d, %d, %d", i, j, m);
                            if (diff<0.1){
                                livox_plane_in_sections[i].points.push_back(livox_clouds_in_sections[i][j][m]);
                            }
                        }
                    }
                }else if(sections[i].feature_flag==2){
                    for(int j =0; j<livox_clouds_in_sections[i].size();j++ ){
                        // ROS_INFO("Index j: %d, %d, %d", j, livox_clouds_in_sections[i].size(), livox_clouds_in_sections[i][j].size());
                        if(livox_clouds_in_sections[i][j].size()<6){
                            continue;
                        }
                        for(int m = 5; m<livox_clouds_in_sections[i][j].size()-5;m++ ){
                            // ROS_INFO("Index m: %d, %d", m, livox_clouds_in_sections[i][j].size());
                            // ROS_INFO("Corner Num: %d,", livox_clouds_in_sections[i].size());
                            float ldiffX =livox_clouds_in_sections[i][j][m - 5].x + livox_clouds_in_sections[i][j][m - 4].x + livox_clouds_in_sections[i][j][m - 3].x + livox_clouds_in_sections[i][j][m - 2].x + livox_clouds_in_sections[i][j][m - 1].x - 5 * livox_clouds_in_sections[i][j][m].x;
                            float ldiffY =livox_clouds_in_sections[i][j][m - 5].y + livox_clouds_in_sections[i][j][m - 4].y + livox_clouds_in_sections[i][j][m - 3].y + livox_clouds_in_sections[i][j][m - 2].y + livox_clouds_in_sections[i][j][m - 1].y - 5 * livox_clouds_in_sections[i][j][m].y;
                            float ldiffZ =livox_clouds_in_sections[i][j][m - 5].z + livox_clouds_in_sections[i][j][m - 4].z + livox_clouds_in_sections[i][j][m - 3].z +
```

```
livox_clouds_in_sections[i][j][m - 2].z + livox_clouds_in_sections[i][j][m - 1].z - 5 * livox_clouds_in_sections[i][j][m].z;
                        float ldiff = ldiffX * ldiffX + ldiffY * ldiffY + ldiffZ * ldiffZ;
                        float rdiffX = livox_clouds_in_sections[i][j][m + 1 ].x + livox_clouds_in_sections[i][j][m + 2].x + livox_clouds_in_sections[i][j][m + 3].x  + livox_clouds_in_sections[i][j][m + 4].x +livox_clouds_in_sections[i][j][m + 5].x - 5 * livox_clouds_in_sections[i][j][m].x;
                        float rdiffY = livox_clouds_in_sections[i][j][m + 1 ].y + livox_clouds_in_sections[i][j][m + 2].y + livox_clouds_in_sections[i][j][m + 3].y  + livox_clouds_in_sections[i][j][m + 4].y +livox_clouds_in_sections[i][j][m + 5].y - 5 * livox_clouds_in_sections[i][j][m].y;
                        float rdiffZ = livox_clouds_in_sections[i][j][m + 1 ].z + livox_clouds_in_sections[i][j][m + 2].z + livox_clouds_in_sections[i][j][m + 3].z  + livox_clouds_in_sections[i][j][m + 4].z +livox_clouds_in_sections[i][j][m + 5].z - 5 * livox_clouds_in_sections[i][j][m].z;
                        float rdiff = rdiffX * rdiffX + rdiffY * rdiffY + rdiffZ * rdiffZ;
                        float cdiff = (ldiffX + rdiffX) * (ldiffX + rdiffX) + (ldiffY + rdiffY) * (ldiffY + rdiffY) + (ldiffZ + rdiffZ) * (ldiffZ + rdiffZ);

                        // if (ldiff+rdiff<cdiff){
                        //     // ROS_INFO("Corner Spec: %f, %f, %f", ldiff, rdiff, cdiff);
                        //     Eigen::Vector3d norm_left(0,0,0);
                        //     Eigen::Vector3d norm_right(0,0,0);
                        //     for(int k = 1;k<5;k++){
                        //         Eigen::Vector3d tmp = Eigen::Vector3d(livox_clouds_in_sections[i][j][m-k].x-livox_clouds_in_sections[i][j][m].x,
                        //  livox_clouds_in_sections[i][j][m-k].y-livox_clouds_in_sections[i][j][m].y,
                        //  livox_clouds_in_sections[i][j][m-k].z-livox_clouds_in_sections[i][j][m].z);
                        //         tmp.normalize();//Normalizes a compile time known vector (as in a vector that is known to be a vector at compile time) in place, returns nothing.
                        //         norm_left += (k/10.0)* tmp;
                        //     }
                        //     for(int k = 1;k<5;k++){
                        //         Eigen::Vector3d tmp = Eigen::Vector3d(livox_clouds_in_sections[i][j][m+k].x-livox_clouds_in_sections[i][j][m].x,
                        //  livox_clouds_in_sections[i][j][m+k].y-livox_clouds_in_sections[i][j][m].y,
                        //  livox_clouds_in_sections[i][j][m+k].z-livox_clouds_in_sections[i][j][m].z);
                        //         tmp.normalize();//
                        //         norm_right += (k/10.0)* tmp;
                        //     }
                        //     double cc = fabs( norm_left.dot(norm_right) / (norm_left.norm()*norm_right.norm()) );


                        //     double dis = livox_clouds_in_sections[i][j][m].x * livox_clouds_in_sections[i][j][m].x + livox_clouds_in_sections[i][j][m].y * livox_clouds_in_sections[i][j][m].y + livox_clouds_in_sections[i][j][m].z * livox_clouds_in_sections[i][j][m].z;
                        //     double dis2 = livox_clouds_in_sections[i][j][m].z * livox_clouds_in_sections[i][j][m].z + livox_clouds_in_sections[i][j][m].y * livox_clouds_in_sections[i][j][m].y;
                        //     double theta2 = std::asin(sqrt(dis2/dis)) / M_PI * 180;
                        //     int section_gap = 5;

                        //     if(0.6<cc<1
                        //         &&fabs(0-m)>section_gap&&fabs(livox_clouds_in_sections[i][j].size()-m)>section_gap
                        //         // &&fabs(livox_clouds_in_sections[i][j][m].x-sections[i].start_point.x)>section_gap
                        //         // &&fabs(livox_clouds_in_sections[i][j][m].x-sections[i].end_point.x)>section_gap
                        //         &&fabs(theta2<18))
                        //     {
                        //         // ROS_INFO("Gap Spec: %f: ", livox_clouds_in_sections[i][j].x-sections[i].start_point.x);
                        //         // ROS_INFO("Gap Spec: %f: ", livox_clouds_in_sections[i][j].y-sections[i].start_point.y);
                        //         // ROS_INFO("Gap Spec: %f: ",
```

```
    livox_clouds_in_sections[i][j].x-sections[i].end_point.x);
                                //          // ROS_INFO("Gap Spec: %f: ",
    livox_clouds_in_sections[i][j].y-sections[i].end_point.y);
                                //          // ROS_INFO("Point Spec: %f, %f:",
    livox_clouds_in_sections[i][j].x, livox_clouds_in_sections[i][j].y);
                                //          livox_corner_in_sections[i].points.push_back(
    livox_clouds_in_sections[i][j][m]);
                                //      }
                                // }
                                float section_gap = 5;
                                if (ldiff+rdiff<cdiff){
                                    double dis = livox_clouds_in_sections[i][j][m].x *
    livox_clouds_in_sections[i][j][m].x + livox_clouds_in_sections[i][j][m].y *
    livox_clouds_in_sections[i][j][m].y + livox_clouds_in_sections[i][j][m].z *
    livox_clouds_in_sections[i][j][m].z;
                                    double dis2 = livox_clouds_in_sections[i][j][m].z *
    livox_clouds_in_sections[i][j][m].z + livox_clouds_in_sections[i][j][m].y *
    livox_clouds_in_sections[i][j][m].y;
                                    double theta2 = std::asin(sqrt(dis2/dis)) / M_PI * 180;
                                    // ROS_INFO("Angle: %f: ", theta2);
                                    if(fabs(0-m)>section_gap&&fabs(livox_clouds_in_sections[i
    ][j].size()-m)>section_gap
                                        &&fabs(theta2<18))
                                        {
                                            livox_corner_in_sections[i].points.push_back(
    livox_clouds_in_sections[i][j][m]);
                                        }
                                }
                            }
                        }
                    }
                }
                pcl::PointCloud<pcl::PointXYZI> allcloud, cornercloud, planecloud;
                for (int i =0; i < livox_corner_in_sections.size();i++){
                    // allcloud += livox_corner_in_sections[i];
                    cornercloud += livox_corner_in_sections[i];
                }
                for (int i =0; i < livox_plane_in_sections.size();i++){
                    // allcloud += livox_plane_in_sections[i];
                    planecloud += livox_plane_in_sections[i];
                }
                livox_data.clear();
                for (int i = 0; i < livox_clouds_in_sections.size();i++){
                    for (int j = 0; j < livox_clouds_in_sections[i].size();j++){
                        livox_data += livox_clouds_in_sections[i][j];
                    }
                }
                ROS_INFO("Total LiDAR Points: %i, Surface Points: %i, Corner Points: %i",
    livox_data.size(), planecloud.size(), cornercloud.size());
                sensor_msgs::PointCloud2 laserCloudOutMsg, laserCloudCornerOutMsg,
    laserCloudSurfaceOutMsg;
                pcl::toROSMsg(livox_data, laserCloudOutMsg);
                laserCloudOutMsg.header.stamp = livox_msg_in->header.stamp;
                laserCloudOutMsg.header.frame_id = "livox_frame";
                pub_livox_proc.publish(laserCloudOutMsg);
                pcl::toROSMsg(cornercloud, laserCloudCornerOutMsg);
                laserCloudCornerOutMsg.header.stamp = livox_msg_in->header.stamp;
                laserCloudCornerOutMsg.header.frame_id = "livox_frame";
                pub_livox_corner_proc.publish(laserCloudCornerOutMsg);
                pcl::toROSMsg(planecloud, laserCloudSurfaceOutMsg);
                laserCloudSurfaceOutMsg.header.stamp = livox_msg_in->header.stamp;
                laserCloudSurfaceOutMsg.header.frame_id = "livox_frame";
                pub_livox_surface_proc.publish(laserCloudSurfaceOutMsg);
            }
            // void livox_feature_proc
};

int main(int argc, char **argv)
{
    ros::init(argc, argv, "livox_reciver");
    Livox_Proc LP;
    ros::spin();
    return 0;
}
```

A.4.3 6 DOF odometry

```
#include <math.h>
#include <nav_msgs/Odometry.h>
#include <opencv2/opencv.hpp>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/io/pcd_io.h>
#include <mutex>
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <tf/transform_datatypes.h>
#include <tf/transform_broadcaster.h>
#include <mloam/HighFreqOdom.h>
typedef pcl::PointXYZI PointType;
int kfNum = 0;
float timeLaserCloudCornerLast = 0;
float timeLaserCloudSurfLast = 0;
float timeLaserCloudFullRes = 0;
bool newLaserCloudCornerLast = false;
bool newLaserCloudSurfLast = false;
bool newLaserCloudFullRes = false;
bool newtwoDCloudFullRes = false;
bool init2d = false;
int laserCloudCenWidth = 10;
int laserCloudCenHeight = 5;
int laserCloudCenDepth = 10;
const int laserCloudWidth = 21;
const int laserCloudHeight = 11;
const int laserCloudDepth = 21;
const int laserCloudNum = laserCloudWidth * laserCloudHeight * laserCloudDepth;//4851
int laserCloudValidInd[125];
int laserCloudSurroundInd[125];
std::mutex mutex_trans_update;
//corner feature
pcl::PointCloud<PointType>::Ptr laserCloudCornerLast(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudCornerLast_down(new pcl::PointCloud<PointType
    >());
//surf feature
pcl::PointCloud<PointType>::Ptr laserCloudSurfLast(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudSurfLast_down(new pcl::PointCloud<PointType>())
    ;
pcl::PointCloud<PointType>::Ptr laserCloudOri(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr coeffSel(new pcl::PointCloud<PointType>());
// pcl::PointCloud<PointType>::Ptr laserCloudSurround(new pcl::PointCloud<PointType>());
// pcl::PointCloud<PointType>::Ptr laserCloudSurround_corner(new pcl::PointCloud<
    PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudSurround2(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudSurround2_corner(new pcl::PointCloud<PointType
    >());
//corner feature in map
pcl::PointCloud<PointType>::Ptr laserCloudCornerFromMap(new pcl::PointCloud<PointType>())
    ;
//surf feature in map
pcl::PointCloud<PointType>::Ptr laserCloudSurfFromMap(new pcl::PointCloud<PointType>());
std::vector< Eigen::Matrix<float,7,1> > keyframe_pose;
std::vector< Eigen::Matrix4f > pose_map;
//all points
pcl::PointCloud<PointType>::Ptr laserCloudFullRes(new pcl::PointCloud<PointType>());
pcl::PointCloud<PointType>::Ptr laserCloudFullRes2(new pcl::PointCloud<PointType>());
pcl::PointCloud<pcl::PointXYZRGB>::Ptr laserCloudFullResColor(new pcl::PointCloud<pcl::
    PointXYZRGB>());
pcl::PointCloud<pcl::PointXYZRGB>::Ptr laserCloudFullResColor_pcd(new pcl::PointCloud<pcl
    ::PointXYZRGB>());

pcl::PointCloud<PointType>::Ptr laserCloudCornerArray[laserCloudNum];
pcl::PointCloud<PointType>::Ptr laserCloudSurfArray[laserCloudNum];
pcl::PointCloud<PointType>::Ptr laserCloudCornerArray2[laserCloudNum];
pcl::PointCloud<PointType>::Ptr laserCloudSurfArray2[laserCloudNum];
pcl::KdTreeFLANN<PointType>::Ptr kdtreeCornerFromMap(new pcl::KdTreeFLANN<PointType>());
pcl::KdTreeFLANN<PointType>::Ptr kdtreeSurfFromMap(new pcl::KdTreeFLANN<PointType>());
//optimization states
float transformTobeMapped[6] = {0};
//optimization states after mapping
float transformAftMapped[6] = {0};
```

```
//last optimization states
float transformLastMapped[6] = {0};
double rad2deg(double radians)
{
  return radians * 180.0 / M_PI;
}
double deg2rad(double degrees)
{
  return degrees * M_PI / 180.0;
}
Eigen::Matrix4f trans_euler_to_matrix(const float *trans)
{
    Eigen::Matrix4f T = Eigen::Matrix4f::Identity();
    Eigen::Matrix3f R;
    Eigen::AngleAxisf rollAngle(Eigen::AngleAxisf(trans[0],Eigen::Vector3f::UnitX()));
    Eigen::AngleAxisf pitchAngle(Eigen::AngleAxisf(trans[1],Eigen::Vector3f::UnitY()));
    Eigen::AngleAxisf yawAngle(Eigen::AngleAxisf(trans[2],Eigen::Vector3f::UnitZ()));
    R = pitchAngle * rollAngle * yawAngle; //zxy
    T.block<3,3>(0,0) = R;
    T.block<3,1>(0,3) = Eigen::Vector3f(trans[3],trans[4],trans[5]);
    return T;
}
void transformAssociateToMap()
{
    Eigen::Matrix4f T_aft,T_last,T_predict;
    Eigen::Matrix3f R_predict;
    Eigen::Vector3f euler_predict,t_predict;
    T_aft = trans_euler_to_matrix(transformAftMapped);
    T_last = trans_euler_to_matrix(transformLastMapped);

    T_predict = T_aft * T_last.inverse() * T_aft;
    R_predict = T_predict.block<3,3>(0,0);
    euler_predict = R_predict.eulerAngles(1,0,2);
    t_predict = T_predict.block<3,1>(0,3);
    transformTobeMapped[0] = euler_predict[0];//X
    transformTobeMapped[1] = euler_predict[1];//Y
    transformTobeMapped[2] = euler_predict[2];//Z
    transformTobeMapped[3] = t_predict[0];
    transformTobeMapped[4] = t_predict[1];
    transformTobeMapped[5] = t_predict[2];
    // std::cout<<"DEBUG transformAftMapped : "<<transformAftMapped[0]<<" "<<transformAftMapped[1]<<" "<<transformAftMapped[2]<<" "
    // <<transformAftMapped[3]<<" "<<transformAftMapped[4]<<" "<<transformAftMapped[5]<<std::endl;
    // std::cout<<"DEBUG transformTobeMapped : "<<transformTobeMapped[0]<<" "<<transformTobeMapped[1]<<" "<<transformTobeMapped[2]<<" "
    // <<transformTobeMapped[3]<<" "<<transformTobeMapped[4]<<" "<<transformTobeMapped[5]<<std::endl;
}
void transformUpdate()
{
    for (int i = 0; i < 6; i++) {
        transformLastMapped[i] = transformAftMapped[i];
        transformAftMapped[i] = transformTobeMapped[i];
    }
}
//lidar coordinate sys to world coordinate sys
void pointAssociateToMap(PointType const * const pi, PointType * const po)
{
    //rot ztransformTobeMapped[2]
    float x1 = cos(transformTobeMapped[2]) * pi->x
            - sin(transformTobeMapped[2]) * pi->y;
    float y1 = sin(transformTobeMapped[2]) * pi->x
            + cos(transformTobeMapped[2]) * pi->y;
    float z1 = pi->z;
    //rot xtransformTobeMapped[0]
    float x2 = x1;
    float y2 = cos(transformTobeMapped[0]) * y1 - sin(transformTobeMapped[0]) * z1;
    float z2 = sin(transformTobeMapped[0]) * y1 + cos(transformTobeMapped[0]) * z1;
    //rot ytransformTobeMapped[1]then add trans
    po->x = cos(transformTobeMapped[1]) * x2 + sin(transformTobeMapped[1]) * z2
            + transformTobeMapped[3];
    po->y = y2 + transformTobeMapped[4];
    po->z = -sin(transformTobeMapped[1]) * x2 + cos(transformTobeMapped[1]) * z2
            + transformTobeMapped[5];
    po->intensity = pi->intensity;
}
```

```
//lidar coordinate sys to world coordinate sys USE S
void RGBpointAssociateToMap(PointType const * const pi, pcl::PointXYZRGB * const po)
{
    double s;
    s = pi->intensity - int(pi->intensity);
    // float rx = (1-s)*transformLastMapped[0] + s * transformAftMapped[0];
    // float ry = (1-s)*transformLastMapped[1] + s * transformAftMapped[1];
    // float rz = (1-s)*transformLastMapped[2] + s * transformAftMapped[2];
    // float tx = (1-s)*transformLastMapped[3] + s * transformAftMapped[3];
    // float ty = (1-s)*transformLastMapped[4] + s * transformAftMapped[4];
    // float tz = (1-s)*transformLastMapped[5] + s * transformAftMapped[5];
    float rx = transformAftMapped[0];
    float ry = transformAftMapped[1];
    float rz = transformAftMapped[2];
    float tx = transformAftMapped[3];
    float ty = transformAftMapped[4];
    float tz = transformAftMapped[5];
    //rot ztransformTobeMapped[2]
    float x1 = cos(rz) * pi->x
            - sin(rz) * pi->y;
    float y1 = sin(rz) * pi->x
            + cos(rz) * pi->y;
    float z1 = pi->z;
    //rot xtransformTobeMapped[0]
    float x2 = x1;
    float y2 = cos(rx) * y1 - sin(rx) * z1;
    float z2 = sin(rx) * y1 + cos(rx) * z1;
    //rot ytransformTobeMapped[1]then add trans
    po->x = cos(ry) * x2 + sin(ry) * z2 + tx;
    po->y = y2 + ty;
    po->z = -sin(ry) * x2 + cos(ry) * z2 + tz;
    //po->intensity = pi->intensity;
    float intensity = pi->intensity;
    intensity = intensity - std::floor(intensity);
    int reflection_map = intensity*10000;
    //std::cout<<"DEBUG reflection_map "<<reflection_map<<std::endl;
    if (reflection_map < 30)
    {
        int green = (reflection_map * 255 / 30);
        po->r = 0;
        po->g = green & 0xff;
        po->b = 0xff;
    }
    else if (reflection_map < 90)
    {
        int blue = (((90 - reflection_map) * 255) / 60);
        po->r = 0x0;
        po->g = 0xff;
        po->b = blue & 0xff;
    }
    else if (reflection_map < 150)
    {
        int red = ((reflection_map-90) * 255 / 60);
        po->r = red & 0xff;
        po->g = 0xff;
        po->b = 0x0;
    }
    else
    {
        int green = (((255-reflection_map) * 255) / (255-150));
        po->r = 0xff;
        po->g = green & 0xff;
        po->b = 0;
    }
}
void pointAssociateTobeMapped(PointType const * const pi, PointType * const po)
{
    //add trans then rot y
    float x1 = cos(transformTobeMapped[1]) * (pi->x - transformTobeMapped[3])
            - sin(transformTobeMapped[1]) * (pi->z - transformTobeMapped[5]);
    float y1 = pi->y - transformTobeMapped[4];
    float z1 = sin(transformTobeMapped[1]) * (pi->x - transformTobeMapped[3])
            + cos(transformTobeMapped[1]) * (pi->z - transformTobeMapped[5]);
    //rot x
    float x2 = x1;
    float y2 = cos(transformTobeMapped[0]) * y1 + sin(transformTobeMapped[0]) * z1;
    float z2 = -sin(transformTobeMapped[0]) * y1 + cos(transformTobeMapped[0]) * z1;
    //rot z
```

```
    po->x = cos(transformTobeMapped[2]) * x2
            + sin(transformTobeMapped[2]) * y2;
    po->y = -sin(transformTobeMapped[2]) * x2
            + cos(transformTobeMapped[2]) * y2;
    po->z = z2;
    po->intensity = pi->intensity;
}
void laserCloudCornerLastHandler(const sensor_msgs::PointCloud2ConstPtr&
    laserCloudCornerLast2)
{
    timeLaserCloudCornerLast = laserCloudCornerLast2->header.stamp.toSec();
    laserCloudCornerLast->clear();
    pcl::fromROSMsg(*laserCloudCornerLast2, *laserCloudCornerLast);
    newLaserCloudCornerLast = true;
}
void laserCloudSurfLastHandler(const sensor_msgs::PointCloud2ConstPtr&
    laserCloudSurfLast2)
{
    timeLaserCloudSurfLast = laserCloudSurfLast2->header.stamp.toSec();
    laserCloudSurfLast->clear();
    pcl::fromROSMsg(*laserCloudSurfLast2, *laserCloudSurfLast);
    newLaserCloudSurfLast = true;
}
void laserCloudFullResHandler(const sensor_msgs::PointCloud2ConstPtr& laserCloudFullRes2)
{
    timeLaserCloudFullRes = laserCloudFullRes2->header.stamp.toSec();
    laserCloudFullRes->clear();
    laserCloudFullResColor->clear();
    pcl::fromROSMsg(*laserCloudFullRes2, *laserCloudFullRes);
    newLaserCloudFullRes = true;
}
void HighFreqOdomHandler(const mloam::HighFreqOdom& high_freq_odom)
{
    float highfreqtransformTobeMapped[6] = {0};
    highfreqtransformTobeMapped[0] = high_freq_odom.rx;
    highfreqtransformTobeMapped[1] = high_freq_odom.ry;
    highfreqtransformTobeMapped[2] = high_freq_odom.rz;
    highfreqtransformTobeMapped[3] = high_freq_odom.tx;
    highfreqtransformTobeMapped[4] = high_freq_odom.ty;
    highfreqtransformTobeMapped[5] = high_freq_odom.tz;
    // ROS_INFO("Ceres Rot x: %f, y: %f, z:%f:",  highfreqtransformTobeMapped[0],
    highfreqtransformTobeMapped[1], highfreqtransformTobeMapped[2]);
    // ROS_INFO("Ceres Trans x: %f, y: %f, z:%f:",  highfreqtransformTobeMapped[3],
    highfreqtransformTobeMapped[4], highfreqtransformTobeMapped[5]);
    mutex_trans_update.lock();
    for (int i = 0; i < 6; i++)
    {
        transformTobeMapped[i] += highfreqtransformTobeMapped[i];
    }
    mutex_trans_update.unlock();

}
int main(int argc, char** argv)
{
    ros::init(argc, argv, "laserMapping");
    ros::NodeHandle nh;
    ros::Subscriber subLaserCloudCornerLast = nh.subscribe<sensor_msgs::PointCloud2>
            ("/pc2_corners", 100, laserCloudCornerLastHandler);
    ros::Subscriber subLaserCloudSurfLast = nh.subscribe<sensor_msgs::PointCloud2>
            ("/pc2_surface", 100, laserCloudSurfLastHandler);
    ros::Subscriber subLaserCloudFullRes = nh.subscribe<sensor_msgs::PointCloud2>
            ("/pc2_full", 100, laserCloudFullResHandler);

    ros::Publisher pubLaserCloudSurround = nh.advertise<sensor_msgs::PointCloud2>
            ("/laser_cloud_surround", 100);
    ros::Publisher pubLaserCloudSurround_corner = nh.advertise<sensor_msgs::PointCloud2>
            ("/laser_cloud_surround_corner", 100);
    ros::Publisher pubLaserCloudFullRes = nh.advertise<sensor_msgs::PointCloud2>
            ("/velodyne_cloud_registered", 100);
    ros::Subscriber subHighFreqOdom = nh.subscribe
            ("/high_freq_odom", 100, HighFreqOdomHandler);
    ros::Publisher pubOdomAftMapped = nh.advertise<nav_msgs::Odometry> ("/
    aft_mapped_to_init", 1);
    nav_msgs::Odometry odomAftMapped;
    odomAftMapped.header.frame_id = "camera_init";
    odomAftMapped.child_frame_id = "aft_mapped";
    std::string map_file_path;
    ros::param::get("~map_file_path",map_file_path);
```

```
    double filter_parameter_corner;
    ros::param::get("~filter_parameter_corner",filter_parameter_corner);
    double filter_parameter_surf;
    ros::param::get("~filter_parameter_surf",filter_parameter_surf);
    std::vector<int> pointSearchInd;
    std::vector<float> pointSearchSqDis;
    PointType pointOri, pointSel, coeff;
    cv::Mat matA0(10, 3, CV_32F, cv::Scalar::all(0));
    cv::Mat matB0(10, 1, CV_32F, cv::Scalar::all(-1));
    cv::Mat matX0(10, 1, CV_32F, cv::Scalar::all(0));
    cv::Mat matA1(3, 3, CV_32F, cv::Scalar::all(0));
    cv::Mat matD1(1, 3, CV_32F, cv::Scalar::all(0));
    cv::Mat matV1(3, 3, CV_32F, cv::Scalar::all(0));
    bool isDegenerate = false;
    cv::Mat matP(6, 6, CV_32F, cv::Scalar::all(0));
    //VoxelGrid
    pcl::VoxelGrid<PointType> downSizeFilterCorner;
    downSizeFilterCorner.setLeafSize(filter_parameter_corner, filter_parameter_corner, filter_parameter_corner);
    pcl::VoxelGrid<PointType> downSizeFilterSurf;
    downSizeFilterSurf.setLeafSize(filter_parameter_surf, filter_parameter_surf, filter_parameter_surf);
    // pcl::VoxelGrid<PointType> downSizeFilterFull;
    // downSizeFilterFull.setLeafSize(0.15, 0.15, 0.15);
    for (int i = 0; i < laserCloudNum; i++) {//LOAMCube 21*11*21
        laserCloudCornerArray[i].reset(new pcl::PointCloud<PointType>());
        laserCloudSurfArray[i].reset(new pcl::PointCloud<PointType>());
        laserCloudCornerArray2[i].reset(new pcl::PointCloud<PointType>());
        laserCloudSurfArray2[i].reset(new pcl::PointCloud<PointType>());
    }
//    ----------------------------------------------------------------------------------------------------------
    ros::Rate rate(100);
    bool status = ros::ok();
    while (status) {
        ros::spinOnce();
        if (newLaserCloudCornerLast && newLaserCloudSurfLast && newLaserCloudFullRes &&
                fabs(timeLaserCloudSurfLast - timeLaserCloudCornerLast) < 0.005 &&
                fabs(timeLaserCloudFullRes - timeLaserCloudCornerLast) < 0.005) {//
            clock_t t1,t2,t3,t4;
            t1 = clock();
            newLaserCloudCornerLast = false;
            newLaserCloudSurfLast = false;
            newLaserCloudFullRes = false;
            //transformAssociateToMap();
            // std::cout<<"DEBUG mapping start "<<std::endl;
            PointType pointOnYAxis;
            pointOnYAxis.x = 0.0;
            pointOnYAxis.y = 10.0;
            pointOnYAxis.z = 0.0;
            pointAssociateToMap(&pointOnYAxis, &pointOnYAxis);//Y10m
            //cube transformTobeMapped 345
            int centerCubeI = int((transformTobeMapped[3] + 25.0) / 50.0) + laserCloudCenWidth;//cube
            int centerCubeJ = int((transformTobeMapped[4] + 25.0) / 50.0) + laserCloudCenHeight;
            int centerCubeK = int((transformTobeMapped[5] + 25.0) / 50.0) + laserCloudCenDepth;
            if (transformTobeMapped[3] + 25.0 < 0) centerCubeI--;
            if (transformTobeMapped[4] + 25.0 < 0) centerCubeJ--;
            if (transformTobeMapped[5] + 25.0 < 0) centerCubeK--;
            while (centerCubeI < 3) {
                for (int j = 0; j < laserCloudHeight; j++) {
                    for (int k = 0; k < laserCloudDepth; k++) {
                        int i = laserCloudWidth - 1;
                        pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
                                laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
                        pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
                                laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
                        for (; i >= 1; i--) {
                            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                    laserCloudCornerArray[i - 1 + laserCloudWidth*j + laserCloudWidth * laserCloudHeight * k];
                            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
```

```
                  laserCloudSurfArray[i - 1 + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            }
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCubeCornerPointer;
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCubeSurfPointer;
            laserCloudCubeCornerPointer->clear();
            laserCloudCubeSurfPointer->clear();
          }
        }
        centerCubeI++;
        laserCloudCenWidth++;
      }
      while (centerCubeI >= laserCloudWidth - 3) {
        for (int j = 0; j < laserCloudHeight; j++) {
          for (int k = 0; k < laserCloudDepth; k++) {
            int i = 0;
            pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
                laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
                laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            for (; i < laserCloudWidth - 1; i++) {
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                  laserCloudCornerArray[i + 1 + laserCloudWidth*j + laserCloudWidth * laserCloudHeight * k];
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                  laserCloudSurfArray[i + 1 + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            }
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCubeCornerPointer;
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCubeSurfPointer;
            laserCloudCubeCornerPointer->clear();
            laserCloudCubeSurfPointer->clear();
          }
        }
        centerCubeI--;
        laserCloudCenWidth--;
      }
      while (centerCubeJ < 3) {
        for (int i = 0; i < laserCloudWidth; i++) {
          for (int k = 0; k < laserCloudDepth; k++) {
            int j = laserCloudHeight - 1;
            pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
                laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
                laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
            for (; j >= 1; j--) {
              laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                  laserCloudCornerArray[i + laserCloudWidth*(j - 1) + laserCloudWidth * laserCloudHeight*k];
              laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                  laserCloudSurfArray[i + laserCloudWidth * (j - 1) + laserCloudWidth * laserCloudHeight*k];
            }
            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCubeCornerPointer;
            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                laserCloudCubeSurfPointer;
            laserCloudCubeCornerPointer->clear();
            laserCloudCubeSurfPointer->clear();
          }
```

```
                }
                centerCubeJ++;
                laserCloudCenHeight++;
            }
            while (centerCubeJ >= laserCloudHeight - 3) {
                for (int i = 0; i < laserCloudWidth; i++) {
                    for (int k = 0; k < laserCloudDepth; k++) {
                        int j = 0;
                        pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
                                laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
                        pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
                                laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
                        for (; j < laserCloudHeight - 1; j++) {
                            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                    laserCloudCornerArray[i + laserCloudWidth*(j + 1) + laserCloudWidth * laserCloudHeight*k];
                            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                    laserCloudSurfArray[i + laserCloudWidth * (j + 1) + laserCloudWidth * laserCloudHeight*k];
                        }
                        laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                laserCloudCubeCornerPointer;
                        laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                laserCloudCubeSurfPointer;
                        laserCloudCubeCornerPointer->clear();
                        laserCloudCubeSurfPointer->clear();
                    }
                }
                centerCubeJ--;
                laserCloudCenHeight--;
            }
            while (centerCubeK < 3) {
                for (int i = 0; i < laserCloudWidth; i++) {
                    for (int j = 0; j < laserCloudHeight; j++) {
                        int k = laserCloudDepth - 1;
                        pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
                                laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
                        pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
                                laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
                        for (; k >= 1; k--) {
                            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                    laserCloudCornerArray[i + laserCloudWidth*j + laserCloudWidth * laserCloudHeight*(k - 1)];
                            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                    laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight*(k - 1)];
                        }
                        laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                laserCloudCubeCornerPointer;
                        laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                laserCloudCubeSurfPointer;
                        laserCloudCubeCornerPointer->clear();
                        laserCloudCubeSurfPointer->clear();
                    }
                }
                centerCubeK++;
                laserCloudCenDepth++;
            }
            while (centerCubeK >= laserCloudDepth - 3) {
                for (int i = 0; i < laserCloudWidth; i++) {
                    for (int j = 0; j < laserCloudHeight; j++) {
                        int k = 0;
                        pcl::PointCloud<PointType>::Ptr laserCloudCubeCornerPointer =
                                laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
                        pcl::PointCloud<PointType>::Ptr laserCloudCubeSurfPointer =
```

```
                                laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k];
                        for (; k < laserCloudDepth - 1; k++) {
                            laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                    laserCloudCornerArray[i + laserCloudWidth*j + laserCloudWidth * laserCloudHeight*(k + 1)];
                            laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                    laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight*(k + 1)];
                        }
                        laserCloudCornerArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                laserCloudCubeCornerPointer;
                        laserCloudSurfArray[i + laserCloudWidth * j + laserCloudWidth * laserCloudHeight * k] =
                                laserCloudCubeSurfPointer;
                        laserCloudCubeCornerPointer->clear();
                        laserCloudCubeSurfPointer->clear();
                    }
                }
                centerCubeK--;
                laserCloudCenDepth--;
            }
            int laserCloudValidNum = 0;
            int laserCloudSurroundNum = 0;
            for (int i = centerCubeI - 2; i <= centerCubeI + 2; i++) {//NOTE livox
                for (int j = centerCubeJ - 2; j <= centerCubeJ + 2; j++) {
                    for (int k = centerCubeK - 2; k <= centerCubeK + 2; k++) {
                        if (i >= 0 && i < laserCloudWidth &&
                                j >= 0 && j < laserCloudHeight &&
                                k >= 0 && k < laserCloudDepth) {
                            float centerX = 50.0 * (i - laserCloudCenWidth);
                            float centerY = 50.0 * (j - laserCloudCenHeight);
                            float centerZ = 50.0 * (k - laserCloudCenDepth);
                            bool isInLaserFOV = false;
                            for (int ii = -1; ii <= 1; ii += 2) {
                                for (int jj = -1; jj <= 1; jj += 2) {
                                    for (int kk = -1; kk <= 1; kk += 2) {
                                        float cornerX = centerX + 25.0 * ii;
                                        float cornerY = centerY + 25.0 * jj;
                                        float cornerZ = centerZ + 25.0 * kk;
                                        float squaredSide1 = (transformTobeMapped[3] - cornerX)
                                                * (transformTobeMapped[3] - cornerX)
                                                + (transformTobeMapped[4] - cornerY)
                                                * (transformTobeMapped[4] - cornerY)
                                                + (transformTobeMapped[5] - cornerZ)
                                                * (transformTobeMapped[5] - cornerZ);
                                        float squaredSide2 = (pointOnYAxis.x - cornerX) * (pointOnYAxis.x - cornerX)
                                                + (pointOnYAxis.y - cornerY) * (pointOnYAxis.y - cornerY)
                                                + (pointOnYAxis.z - cornerZ) * (pointOnYAxis.z - cornerZ);
                                        float check1 = 100.0 + squaredSide1 - squaredSide2
                                                - 10.0 * sqrt(3.0) * sqrt(squaredSide1);
                                        float check2 = 100.0 + squaredSide1 - squaredSide2
                                                + 10.0 * sqrt(3.0) * sqrt(squaredSide1);
                                        if (check1 < 0 && check2 > 0) {
                                            isInLaserFOV = true;
                                        }
                                    }
                                }
                            }
                            if (isInLaserFOV) {//NOTE cube
                                laserCloudValidInd[laserCloudValidNum] = i + laserCloudWidth * j
                                        + laserCloudWidth * laserCloudHeight * k;
                                laserCloudValidNum++;
                            }
                            laserCloudSurroundInd[laserCloudSurroundNum] = i + laserCloudWidth * j
                                    + laserCloudWidth * laserCloudHeight * k;
                            laserCloudSurroundNum++;
                        }
```

```
                }
            }
        }
        laserCloudCornerFromMap->clear();
        laserCloudSurfFromMap->clear();

        for (int i = 0; i < laserCloudValidNum; i++) {
            *laserCloudCornerFromMap += *laserCloudCornerArray[laserCloudValidInd[i]];
            *laserCloudSurfFromMap += *laserCloudSurfArray[laserCloudValidInd[i]];
        }
        int laserCloudCornerFromMapNum = laserCloudCornerFromMap->points.size();
        int laserCloudSurfFromMapNum = laserCloudSurfFromMap->points.size();
        laserCloudCornerLast_down->clear();
        downSizeFilterCorner.setInputCloud(laserCloudCornerLast);
        downSizeFilterCorner.filter(*laserCloudCornerLast_down);
        int laserCloudCornerLast_downNum = laserCloudCornerLast_down->points.size();
        laserCloudSurfLast_down->clear();
        downSizeFilterSurf.setInputCloud(laserCloudSurfLast);
        downSizeFilterSurf.filter(*laserCloudSurfLast_down);
        int laserCloudSurfLast_downNum = laserCloudSurfLast_down->points.size();
        // std::cout<<"DEBUG MAPPING laserCloudCornerLast_down : "<<laserCloudCornerLast_down->points.size()<<" laserCloudSurfLast_down : "
        // <<laserCloudSurfLast_down->points.size()<<std::endl;
        // std::cout<<"DEBUG MAPPING laserCloudCornerLast : "<<laserCloudCornerLast->points.size()<<" laserCloudSurfLast : "
        // <<laserCloudSurfLast->points.size()<<std::endl;
        // std::cout<<"DEBUG MAPPING laserCloudCornerFromMapNum : "<<laserCloudCornerFromMapNum<<" laserCloudSurfFromMapNum : "
        // <<laserCloudSurfFromMapNum<<std::endl;
        t2 = clock();
        if (laserCloudCornerFromMapNum > 10 && laserCloudSurfFromMapNum > 100) {//NOTE
        //if (laserCloudSurfFromMapNum > 100) {
            kdtreeCornerFromMap->setInputCloud(laserCloudCornerFromMap);
            kdtreeSurfFromMap->setInputCloud(laserCloudSurfFromMap);
            int num_temp = 0;
            mutex_trans_update.lock();
            for (int iterCount = 0; iterCount < 20; iterCount++) {//TODO 20 20
                num_temp++;
                laserCloudOri->clear();
                coeffSel->clear();
                for (int i = 0; i < laserCloudCornerLast->points.size(); i++) {
                    pointOri = laserCloudCornerLast->points[i];
                    pointAssociateToMap(&pointOri, &pointSel);
                    //find the closest 5 points
                    kdtreeCornerFromMap->nearestKSearch(pointSel, 5, pointSearchInd, pointSearchSqDis);//NOTE 5 KNNLOAM
                    if (pointSearchSqDis[4] < 1.5) {//NOTE 1.5
                        float cx = 0;
                        float cy = 0;
                        float cz = 0;
                        for (int j = 0; j < 5; j++) {//
                            cx += laserCloudCornerFromMap->points[pointSearchInd[j]].x;
                            cy += laserCloudCornerFromMap->points[pointSearchInd[j]].y;
                            cz += laserCloudCornerFromMap->points[pointSearchInd[j]].z;
                        }
                        cx /= 5;
                        cy /= 5;
                        cz /= 5;
                        //mean square error
                        float a11 = 0;
                        float a12 = 0;
                        float a13 = 0;
                        float a22 = 0;
                        float a23 = 0;
                        float a33 = 0;
                        for (int j = 0; j < 5; j++) {
                            float ax = laserCloudCornerFromMap->points[pointSearchInd[j]].x - cx;
                            float ay = laserCloudCornerFromMap->points[pointSearchInd[j]].y - cy;
                            float az = laserCloudCornerFromMap->points[pointSearchInd[j]].z - cz;
                            a11 += ax * ax;
```

```
                a12 += ax * ay;
                a13 += ax * az;
                a22 += ay * ay;
                a23 += ay * az;
                a33 += az * az;
            }
            a11 /= 5;
            a12 /= 5;
            a13 /= 5;
            a22 /= 5;
            a23 /= 5;
            a33 /= 5;
            matA1.at<float>(0, 0) = a11;
            matA1.at<float>(0, 1) = a12;
            matA1.at<float>(0, 2) = a13;
            matA1.at<float>(1, 0) = a12;
            matA1.at<float>(1, 1) = a22;
            matA1.at<float>(1, 2) = a23;
            matA1.at<float>(2, 0) = a13;
            matA1.at<float>(2, 1) = a23;
            matA1.at<float>(2, 2) = a33;
            cv::eigen(matA1, matD1, matV1);//
            /*
            cv::eigen()
            (Eigenvectors)(Eigenvalues)


            lowindexhighindex
            01
            bool cv::eigen(cv::InputArray src, cv::OutputArray eigenvalues, cv::OutputArray eigenvectors, int lowindex = -1,int highindex = -1);
            */
            if (matD1.at<float>(0, 0) > 3 * matD1.at<float>(0, 1)) {//
                float x0 = pointSel.x;
                float y0 = pointSel.y;
                float z0 = pointSel.z;
                float x1 = cx + 0.1 * matV1.at<float>(0, 0);
                float y1 = cy + 0.1 * matV1.at<float>(0, 1);
                float z1 = cz + 0.1 * matV1.at<float>(0, 2);
                float x2 = cx - 0.1 * matV1.at<float>(0, 0);
                float y2 = cy - 0.1 * matV1.at<float>(0, 1);
                float z2 = cz - 0.1 * matV1.at<float>(0, 2);
                //OA = (x0 - x1, y0 - y1, z0 - z1),OB = (x0 - x2, y0 - y2, z0 - z2)AB = x1 - x2, y1 - y2, z1 - z2
                //cross:
                //|  i       j       k  |
                //|x0-x1  y0-y1  z0-z1|
                //|x0-x2  y0-y2  z0-z2|
                float a012 = sqrt(((x0 - x1)*(y0 - y2) - (x0 - x2)*(y0 - y1))
                                  * ((x0 - x1)*(y0 - y2) - (x0 - x2)*(y0 - y1))
                                  + ((x0 - x1)*(z0 - z2) - (x0 - x2)*(z0 - z1))
                                  * ((x0 - x1)*(z0 - z2) - (x0 - x2)*(z0 - z1))
                                  + ((y0 - y1)*(z0 - z2) - (y0 - y2)*(z0 - z1))
                                  * ((y0 - y1)*(z0 - z2) - (y0 - y2)*(z0 - z1)));
                float l12 = sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2) + (z1 - z2)*(z1 - z2));
                float la = ((y1 - y2)*((x0 - x1)*(y0 - y2) - (x0 - x2)*(y0 - y1))
                            + (z1 - z2)*((x0 - x1)*(z0 - z2) - (x0 - x2)*(z0 - z1))) / a012 / l12;
                float lb = -((x1 - x2)*((x0 - x1)*(y0 - y2) - (x0 - x2)*(y0 - y1))
                                - (z1 - z2)*((y0 - y1)*(z0 - z2) - (y0 - y2)*(z0 - z1))) / a012 / l12;
                float lc = -((x1 - x2)*((x0 - x1)*(z0 - z2) - (x0 - x2)*(z0 - z1))
                                + (y1 - y2)*((y0 - y1)*(z0 - z2) - (y0 - y2)*(z0 - z1))) / a012 / l12;
                float ld2 = a012 / l12;
                //if(fabs(ld2) > 1) continue;
                float s = 1 - 0.9 * fabs(ld2);
```

```
                            coeff.x = s * la;
                            coeff.y = s * lb;
                            coeff.z = s * lc;
                            coeff.intensity = s * ld2;
                            if (s > 0.1) {
                                laserCloudOri->push_back(pointOri);
                                coeffSel->push_back(coeff);
                            }
                        }
                    }
                }
                //std::cout <<"DEBUG mapping select corner points : " << coeffSel->size() << std::endl;
                for (int i = 0; i < laserCloudSurfLast_down->points.size(); i++) {
                    pointOri = laserCloudSurfLast_down->points[i];
                    pointAssociateToMap(&pointOri, &pointSel);
                    kdtreeSurfFromMap->nearestKSearch(pointSel, 8, pointSearchInd, pointSearchSqDis);
                    if (pointSearchSqDis[7] < 5.0) {//85
                        for (int j = 0; j < 8; j++) {//8XYZmatA0
                            matA0.at<float>(j, 0) = laserCloudSurfFromMap->points[pointSearchInd[j]].x;
                            matA0.at<float>(j, 1) = laserCloudSurfFromMap->points[pointSearchInd[j]].y;
                            matA0.at<float>(j, 2) = laserCloudSurfFromMap->points[pointSearchInd[j]].z;
                        }
                        //matA0*matX0=matB0
                        //AX+BY+CZ+D = 0 <=> AX+BY+CZ=-D <=> (A/D)X+(B/D)Y+(C/D)Z = -1
                        //(X,Y,Z)<=>mat_a0
                        //A/D, B/D, C/D <=> mat_x0

                        cv::solve(matA0, matB0, matX0, cv::DECOMP_QR);  //TODO
                        /*
                        cv::invert()
                            // lhsn??n
                            // rhsn??1
                            // dstn??1
                            // method
                            int cv::solve(cv::InputArray lhs, cv::InputArray rhs,
                                        cv::OutputArray dst, int method = cv::DECOMP_LU);
                            cv::DECOMP_QR QR
                        AlhsBrhsCmethod
                        XXdst
                        0
                        */
                        float pa = matX0.at<float>(0, 0);
                        float pb = matX0.at<float>(1, 0);
                        float pc = matX0.at<float>(2, 0);
                        float pd = 1;
                        //ps is the norm of the plane normal vector
                        //pd is the distance from point to plane
                        float ps = sqrt(pa * pa + pb * pb + pc * pc);
                        pa /= ps;
                        pb /= ps;
                        pc /= ps;
                        pd /= ps;
                        bool planeValid = true;
                        for (int j = 0; j < 8; j++) {//
                            if (fabs(pa * laserCloudSurfFromMap->points[pointSearchInd[j]].x +
                                        pb * laserCloudSurfFromMap->points[pointSearchInd[j]].y +
                                        pc * laserCloudSurfFromMap->points[pointSearchInd[j]].z + pd) > 0.2) {
                                planeValid = false;
                                break;
                            }
                        }
                        if (planeValid) {
                            //loss fuction
                            float pd2 = pa * pointSel.x + pb * pointSel.y + pc * pointSel.z + pd;
                            //if(fabs(pd2) > 0.1) continue;
                            float s = 1 - 0.9 * fabs(pd2) / sqrt(sqrt(pointSel.x * pointSel.x + pointSel.y * pointSel.y + pointSel.z * pointSel.z));
```

```
                            coeff.x = s * pa;
                            coeff.y = s * pb;
                            coeff.z = s * pc;
                            coeff.intensity = s * pd2;
                            if (s > 0.1) {
                                laserCloudOri->push_back(pointOri);
                                coeffSel->push_back(coeff);
                            }
                        }
                    }
                }
                //std::cout <<"DEBUG mapping select all points : " << coeffSel->size() << std::endl;
                float srx = sin(transformTobeMapped[0]);
                float crx = cos(transformTobeMapped[0]);
                float sry = sin(transformTobeMapped[1]);
                float cry = cos(transformTobeMapped[1]);
                float srz = sin(transformTobeMapped[2]);
                float crz = cos(transformTobeMapped[2]);
                int laserCloudSelNum = laserCloudOri->points.size();
                if (laserCloudSelNum < 50) {//50
                    continue;
                }

                //|c1c3+s1s2s3 c3s1s2-c1s3 c2s1|
                //|   c2s3        c2c3       -s2|
                //|c1s2s3-c3s1 c1c3s2+s1s3 c1c2|
                //AT*A*x = AT*b
                cv::Mat matA(laserCloudSelNum, 6, CV_32F, cv::Scalar::all(0));
                cv::Mat matAt(6, laserCloudSelNum, CV_32F, cv::Scalar::all(0));
                cv::Mat matAtA(6, 6, CV_32F, cv::Scalar::all(0));
                cv::Mat matB(laserCloudSelNum, 1, CV_32F, cv::Scalar::all(0));
                cv::Mat matAtB(6, 1, CV_32F, cv::Scalar::all(0));
                cv::Mat matX(6, 1, CV_32F, cv::Scalar::all(0));
                float debug_distance = 0;
                for (int i = 0; i < laserCloudSelNum; i++) {
                    pointOri = laserCloudOri->points[i];
                    coeff = coeffSel->points[i];
                    float arx = (crx*sry*srz*pointOri.x + crx*crz*sry*pointOri.y - srx*sry*pointOri.z) * coeff.x
                            + (-srx*srz*pointOri.x - crz*srx*pointOri.y - crx*pointOri.z) * coeff.y
                            + (crx*cry*srz*pointOri.x + crx*cry*crz*pointOri.y - cry*srx*pointOri.z) * coeff.z;
                    float ary = ((cry*srx*srz - crz*sry)*pointOri.x
                                    + (sry*srz + cry*crz*srx)*pointOri.y + crx*cry*pointOri.z) * coeff.x
                            + ((-cry*crz - srx*sry*srz)*pointOri.x
                                + (cry*srz - crz*srx*sry)*pointOri.y - crx*sry*pointOri.z) * coeff.z;
                    float arz = ((crz*srx*sry - cry*srz)*pointOri.x + (-cry*crz-srx*sry*srz)*pointOri.y)*coeff.x
                            + (crx*crz*pointOri.x - crx*srz*pointOri.y) * coeff.y
                            + ((sry*srz + cry*crz*srx)*pointOri.x + (crz*sry-cry*srx*srz)*pointOri.y)*coeff.z;
                    matA.at<float>(i, 0) = arx;
                    matA.at<float>(i, 1) = ary;
                    matA.at<float>(i, 2) = arz;
                    //TODO: the partial derivative
                    matA.at<float>(i, 3) = coeff.x;
                    matA.at<float>(i, 4) = coeff.y;
                    matA.at<float>(i, 5) = coeff.z;
                    matB.at<float>(i, 0) = -coeff.intensity;
                    debug_distance += fabs(coeff.intensity);
                }
                cv::transpose(matA, matAt);//
                matAtA = matAt * matA;
                matAtB = matAt * matB;
                cv::solve(matAtA, matAtB, matX, cv::DECOMP_QR);
                //Deterioration judgment
                if (iterCount == 0) {
                    cv::Mat matE(1, 6, CV_32F, cv::Scalar::all(0));
                    cv::Mat matV(6, 6, CV_32F, cv::Scalar::all(0));
                    cv::Mat matV2(6, 6, CV_32F, cv::Scalar::all(0));
                    cv::eigen(matAtA, matE, matV);
                    matV.copyTo(matV2);
                    isDegenerate = false;
```

```
                    float eignThre[6] = {1, 1, 1, 1, 1, 1};
                    for (int i = 5; i >= 0; i--) {
                        if (matE.at<float>(0, i) < eignThre[i]) {
                            for (int j = 0; j < 6; j++) {
                                matV2.at<float>(i, j) = 0;
                            }
                            isDegenerate = true;
                        } else {
                            break;
                        }
                    }
                    matP = matV.inv() * matV2;
                }
                if (isDegenerate) {
                    cv::Mat matX2(6, 1, CV_32F, cv::Scalar::all(0));
                    matX.copyTo(matX2);
                    matX = matP * matX2;
                }
                transformTobeMapped[0] += matX.at<float>(0, 0);//NOTE
                transformTobeMapped[1] += matX.at<float>(1, 0);
                transformTobeMapped[2] += matX.at<float>(2, 0);
                transformTobeMapped[3] += matX.at<float>(3, 0);
                transformTobeMapped[4] += matX.at<float>(4, 0);
                transformTobeMapped[5] += matX.at<float>(5, 0);
                float deltaR = sqrt(
                            pow(rad2deg(matX.at<float>(0, 0)), 2) +
                            pow(rad2deg(matX.at<float>(1, 0)), 2) +
                            pow(rad2deg(matX.at<float>(2, 0)), 2));
                float deltaT = sqrt(
                            pow(matX.at<float>(3, 0) * 100, 2) +
                            pow(matX.at<float>(4, 0) * 100, 2) +
                            pow(matX.at<float>(5, 0) * 100, 2));
                if (deltaR < 0.05 && deltaT < 0.05) {
                    break;
                }
            }
            std::cout<<"DEBUG num_temp: "<<num_temp << std::endl;
            transformUpdate();
        }
        mutex_trans_update.unlock();
        t3 = clock();
        for (int i = 0; i < laserCloudCornerLast->points.size(); i++) {
            pointAssociateToMap(&laserCloudCornerLast->points[i], &pointSel);
            int cubeI = int((pointSel.x + 25.0) / 50.0) + laserCloudCenWidth;
            int cubeJ = int((pointSel.y + 25.0) / 50.0) + laserCloudCenHeight;
            int cubeK = int((pointSel.z + 25.0) / 50.0) + laserCloudCenDepth;
            if (pointSel.x + 25.0 < 0) cubeI--;
            if (pointSel.y + 25.0 < 0) cubeJ--;
            if (pointSel.z + 25.0 < 0) cubeK--;
            if (cubeI >= 0 && cubeI < laserCloudWidth &&
                    cubeJ >= 0 && cubeJ < laserCloudHeight &&
                    cubeK >= 0 && cubeK < laserCloudDepth) {
                int cubeInd = cubeI + laserCloudWidth * cubeJ + laserCloudWidth *
    laserCloudHeight * cubeK;
                laserCloudCornerArray[cubeInd]->push_back(pointSel);
            }
        }
        for (int i = 0; i < laserCloudSurfLast_down->points.size(); i++) {
            pointAssociateToMap(&laserCloudSurfLast_down->points[i], &pointSel);
            int cubeI = int((pointSel.x + 25.0) / 50.0) + laserCloudCenWidth;
            int cubeJ = int((pointSel.y + 25.0) / 50.0) + laserCloudCenHeight;
            int cubeK = int((pointSel.z + 25.0) / 50.0) + laserCloudCenDepth;
            if (pointSel.x + 25.0 < 0) cubeI--;
            if (pointSel.y + 25.0 < 0) cubeJ--;
            if (pointSel.z + 25.0 < 0) cubeK--;
            if (cubeI >= 0 && cubeI < laserCloudWidth &&
                    cubeJ >= 0 && cubeJ < laserCloudHeight &&
                    cubeK >= 0 && cubeK < laserCloudDepth) {
                int cubeInd = cubeI + laserCloudWidth * cubeJ + laserCloudWidth *
    laserCloudHeight * cubeK;
                laserCloudSurfArray[cubeInd]->push_back(pointSel);
            }
        }
        for (int i = 0; i < laserCloudValidNum; i++) {
            int ind = laserCloudValidInd[i];
            laserCloudCornerArray2[ind]->clear();
            downSizeFilterCorner.setInputCloud(laserCloudCornerArray[ind]);
```

```
                downSizeFilterCorner.filter(*laserCloudCornerArray2[ind]);
                laserCloudSurfArray2[ind]->clear();
                downSizeFilterSurf.setInputCloud(laserCloudSurfArray[ind]);
                downSizeFilterSurf.filter(*laserCloudSurfArray2[ind]);
                pcl::PointCloud<PointType>::Ptr laserCloudTemp = laserCloudCornerArray[ind];
                laserCloudCornerArray[ind] = laserCloudCornerArray2[ind];
                laserCloudCornerArray2[ind] = laserCloudTemp;
                laserCloudTemp = laserCloudSurfArray[ind];
                laserCloudSurfArray[ind] = laserCloudSurfArray2[ind];
                laserCloudSurfArray2[ind] = laserCloudTemp;
            }
            laserCloudSurround2->clear();
            laserCloudSurround2_corner->clear();
            for (int i = 0; i < laserCloudSurroundNum; i++) {
                int ind = laserCloudSurroundInd[i];
                *laserCloudSurround2_corner += *laserCloudCornerArray[ind];
                *laserCloudSurround2 += *laserCloudSurfArray[ind];
            }
            // laserCloudSurround->clear();
            // downSizeFilterSurf.setInputCloud(laserCloudSurround2);
            // downSizeFilterSurf.filter(*laserCloudSurround);
            // laserCloudSurround_corner->clear();
            // downSizeFilterCorner.setInputCloud(laserCloudSurround2_corner);
            // downSizeFilterCorner.filter(*laserCloudSurround_corner);
            sensor_msgs::PointCloud2 laserCloudSurround3;
            pcl::toROSMsg(*laserCloudSurround2, laserCloudSurround3);
            laserCloudSurround3.header.stamp = ros::Time().fromSec(timeLaserCloudCornerLast);
            laserCloudSurround3.header.frame_id = "camera_init";
            pubLaserCloudSurround.publish(laserCloudSurround3);
            sensor_msgs::PointCloud2 laserCloudSurround3_corner;
            pcl::toROSMsg(*laserCloudSurround2_corner, laserCloudSurround3_corner);
            laserCloudSurround3_corner.header.stamp = ros::Time().fromSec(timeLaserCloudCornerLast);
            laserCloudSurround3_corner.header.frame_id = "camera_init";
            pubLaserCloudSurround_corner.publish(laserCloudSurround3_corner);

            laserCloudFullRes2->clear();
            *laserCloudFullRes2 = *laserCloudFullRes;
            int laserCloudFullResNum = laserCloudFullRes2->points.size();
            for (int i = 0; i < laserCloudFullResNum; i++) {
                pcl::PointXYZRGB temp_point;
                RGBpointAssociateToMap(&laserCloudFullRes2->points[i], &temp_point);
                laserCloudFullResColor->push_back(temp_point);
            }
            sensor_msgs::PointCloud2 laserCloudFullRes3;
            pcl::toROSMsg(*laserCloudFullResColor, laserCloudFullRes3);
            laserCloudFullRes3.header.stamp = ros::Time().fromSec(timeLaserCloudCornerLast);
            laserCloudFullRes3.header.frame_id = "camera_init";
            pubLaserCloudFullRes.publish(laserCloudFullRes3);
            *laserCloudFullResColor_pcd += *laserCloudFullResColor;
            geometry_msgs::Quaternion geoQuat = tf::createQuaternionMsgFromRollPitchYaw
                    (transformAftMapped[2], - transformAftMapped[0], - transformAftMapped[1]);
            odomAftMapped.header.stamp = ros::Time().fromSec(timeLaserCloudCornerLast);
            odomAftMapped.pose.pose.orientation.x = -geoQuat.y;
            odomAftMapped.pose.pose.orientation.y = -geoQuat.z;
            odomAftMapped.pose.pose.orientation.z = geoQuat.x;
            odomAftMapped.pose.pose.orientation.w = geoQuat.w;
            odomAftMapped.pose.pose.position.x = transformAftMapped[3];
            odomAftMapped.pose.pose.position.y = transformAftMapped[4];
            odomAftMapped.pose.pose.position.z = transformAftMapped[5];
            pubOdomAftMapped.publish(odomAftMapped);
            static tf::TransformBroadcaster br;
            tf::Transform                   transform;
            tf::Quaternion                  q;
            transform.setOrigin( tf::Vector3( odomAftMapped.pose.pose.position.x,
                                                odomAftMapped.pose.pose.position.y,
                                                odomAftMapped.pose.pose.position.z ) );
            q.setW( odomAftMapped.pose.pose.orientation.w );
            q.setX( odomAftMapped.pose.pose.orientation.x );
            q.setY( odomAftMapped.pose.pose.orientation.y );
            q.setZ( odomAftMapped.pose.pose.orientation.z );
            transform.setRotation( q );
            br.sendTransform( tf::StampedTransform( transform, odomAftMapped.header.stamp, "camera_init", "aft_mapped" ) );
```

```
            kfNum++;
            if(kfNum >= 20){
            Eigen::Matrix<float,7,1> kf_pose;
            kf_pose << -geoQuat.y,-geoQuat.z,geoQuat.x,geoQuat.w,transformAftMapped[3],
    transformAftMapped[4],transformAftMapped[5];
            keyframe_pose.push_back(kf_pose);
            kfNum = 0;
            }
            t4 = clock();
            std::cout<<"mapping time : "<<t2-t1<<" "<<t3-t2<<" "<<t4-t3<<std::endl;

        }
        status = ros::ok();
        rate.sleep();
    }
    //--------------------------save map---------------
    std::string surf_filename(map_file_path + "/surf.pcd");
    std::string corner_filename(map_file_path + "/corner.pcd");
    std::string all_points_filename(map_file_path + "/all_points.pcd");
    std::ofstream keyframe_file(map_file_path + "/key_frame.txt");
    for(auto kf : keyframe_pose){
        keyframe_file << kf[0] << " "<< kf[1] << " "<< kf[2] << " "<< kf[3] << " "
                              << kf[4] << " "<< kf[5] << " "<< kf[6] << " "<< std::endl;
    }
    keyframe_file.close();
    pcl::PointCloud<pcl::PointXYZI> surf_points, corner_points;
    surf_points = *laserCloudSurfFromMap;
    corner_points = *laserCloudCornerFromMap;
      if (surf_points.size() > 0 && corner_points.size() > 0) {
    pcl::PCDWriter pcd_writer;
    std::cout << "saving...";
    pcd_writer.writeBinary(surf_filename, surf_points);
    pcd_writer.writeBinary(corner_filename, corner_points);
    pcd_writer.writeBinary(all_points_filename, *laserCloudFullResColor_pcd);
  } else {
    std::cout << "no points saved";
  }
    //--------------------------
    //  loss_output.close();

    return 0;
}
```