THE UNIVERSITY OF HONG KONG

DOCTORAL THESIS

Towards Efficient LiDAR Mapping for Robotics

Author:

Yixi CAI

Supervisor:

Dr. Fu ZHANG

Co-Supervisor:

Prof. James LAM

In Partial Fulfillment of the Requirements for the

Degree of

Doctor of Philosophy

at

The University of Hong Kong

September 12, 2024

Abstract of thesis entitled

Towards Efficient LiDAR Mapping for Robotics

by

Yixi Cai

Submitted to the Department of Mechanical Engineering
on September 12, 2024, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Mobile robots have been increasingly popular as a replacement for human labor, especially in hazardous or challenging environments. Recent advancements in LiDAR technologies have greatly enhanced the sensing ability of mobile robots with longer range, denser measurements, and higher accuracy. This presents great potential for mapping systems to achieve a more comprehensive understanding of the environment. However, improved sensing ability of LiDAR sensors (e.g., dense measurements) also poses significant efficiency challenges in real-time scenarios to create a general and consistent representation of the environments. This thesis addresses critical challenges related to computational efficiency in LiDAR mapping, with a focus on two typical modules in robotics: simultaneous localization and mapping (SLAM) and occupancy mapping.

The contributions of this thesis are as follows:

Firstly, this thesis proposes a new data structure, the incremental k-d tree (ikd-Tree), to efficiently manage LiDAR point clouds. The ikd-Tree supports incremental updates including point-wise and box-wise operations of insertion, delete, and re-insertion, providing a high level of flexibility for mapping and navigation in robotic applications. To ensure the efficiency of incremental updates and nearest neighbor search, the ikd-Tree employs a twin-threaded re-balancing mechanism that partially rebuilds unbalanced (sub-)trees after each update. The efficiency of the proposed ikd-Tree is validated with a theoretical time complexity analysis as well as benchmark experiments, which demonstrate its superior performance compared to static k-d trees.

Secondly, this thesis presents FAST-LIO2, a fast, accurate, and versatile LiDAR-inertial framework. FAST-LIO2 takes advantage of the high efficiency of ikd-Tree to incrementally register raw points into a point cloud map, eliminating the need for extracting geometric features (e.g., planes and edges) and fully exploiting subtle environmental features. This approach results in significantly improved accuracy and robustness in cluttered environments. Moreover, the removal of the feature extraction module enables FAST-LIO2 to be adaptive to a wide range of emerging LiDAR sensors with different scanning patterns. This thesis also presents exhaustive experiments for validation, including benchmark comparison of accuracy and efficiency against state-of-the-art methods, as well as real-world applications on both handheld and aerial platforms. The results demonstrate the superior performance of FAST-LIO2 compared to other methods and highlight its generality in various challenging scenarios.

Finally, this thesis introduces D-Map, an efficient occupancy mapping framework for high-resolution LiDAR sensors with three key novelties. Firstly, D-Map utilizes a depth image for occupancy state determination as an alternative to ray casting. Secondly, an on-tree update strategy is proposed on a tree-based map structure that reduces the amount of redundant updates. Thirdly, D-Map takes advantage of the low false-alarm rate of LiDAR sensors to directly remove known cells from the map, resulting in a decreasing map size that improves computational and memory efficiency due to the decreasing map size. This thesis presents extensive benchmark experiments to validate the superior efficiency of D-Map compared to existing methods and demonstrates its effectiveness in real-world applications for real-time occupancy mapping using high-resolution LiDAR sensors.

(480 words)

Declaration

I, Yixi Cai, declare that this thesis titled, "Towards Efficient LiDAR Mapping for Robotics", which is submitted in fulfillment of the requirements for the Degree of Doctor of Philosophy, represents my own work except where due acknowledgement have been made. I further declared that it has not been previously included in a thesis, dissertation, or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Signed: _____

Date: September 12, 2024

Delication

To mom.

To my girl.

To my younger self.

In memory of my dear grandma.

You are always with me.

R.I.P.

Acknowledgements

For countless days and sleepless nights over the past few years, I have imagined this very moment: the end of a four-year doctoral journey that has taken me through countless challenges, joys, and discoveries. I have rehearsed the words I would say, the emotions I would express, and the gratitude I would feel for all those who have supported me along the way. But now, as I stand here, I find myself at a loss for words. The weight of this achievement, the depth of my gratitude, and the sheer magnitude of the memories and experiences I have gained seem almost too much to express. Yet, in this moment, I know that I must try.

To begin with, I would like to express my heartfelt gratitude to Hui Pun Hing Endowment Fund for its generous support for my research studies in HKU.

I am compelled to express my sincere gratitude and highest respect to my esteemed supervisor, Prof. Fu Zhang. I recall the summer of 2019 when I had my research internship under his supervision. It was during this period that he instilled in me a profound education that continues to shape my journey until now. Throughout my entire journal towards a Ph.D., Prof. Zhang was always patient in his supervision, providing meticulous instructions that spanned from coding, analysis, hardware, to hands-on experiments. He impelled me to maintain a clear and logical framework in expressing my thoughts, to uphold rigorous standards in every outcome, and to approach the reasons behind each abnormal phenomenon with the utmost patience. Without explicitly dictating what constitutes a distinguished researcher, his every action has left an indelible impact on my academic ethos. Prof. Zhang never simply told me what good research should be; rather, his dedication to excellence instilled within me commitments to maintaining the highest standards in my own research. As I reflect upon our journey together, I am filled with immense gratitude for the privilege of having Prof. Zhang as my supervisor. It is his kindness, boundless patience, profound knowledge, and countless other outstanding qualities that have shaped me from a novice into a successful Ph.D. candidate.

I would like to extend my sincere gratitude to the members of my Ph.D. defense committee. I am grateful to Prof. Maurice Fallon from Oxford University for serving as the external committee member. I also want to thank Prof. Zhiyi Huang for chairing the

committee, and Prof. Xiaojuan Qi and Prof. Yanchao Yang for joining as the internal committee members. I appreciate the time and effort that each of them dedicated to reviewing my thesis and providing insightful advice during my defense.

Next, I would like to express my appreciation to everyone in the Mechatronics and Robotic System (MaRS) Laboratory. I am especially grateful to Dr. Wei Xu for recommending me to MaRS Lab and for his valuable guidance during my early research. I also want to thank Dr. Jiarong Lin for his insightful suggestions and support in shaping my career plan, and Mr. Fanze Kong, my best collaborator, for his invaluable discussions and support during our hardcore experiments. I am also deeply grateful to Dr. Guozheng Lu, Dr. Youming Qin, Dr. Yihang Li, Dr. Dongjiao He, and Dr. Xiyuan Liu for their guidance and suggestions during my Ph.D. journey. Furthermore, I would like to thank Mr. Nan Chen, Mr. Yufan Ren, Mr. Fangcheng Zhu, Mr. Rundong Li, Mr. Haotian Li, Mr. Longji Yin, and Mr. Ziliang Miao for their discussions with me over the years, which inspired me from different perspectives. Joining MaRS Lab at its early stages has been the greatest fortune of my Ph.D. journey. I am honored to have shared countless challenges, joys, and triumphs with all of you. We have supported and encouraged one another through difficult times, contributed to each other's research, and witnessed the growth of our lab from obscurity to fame. I am truly grateful for the opportunity to be a part of this incredible community.

Finally, I want to express my deepest appreciation to my mother for her unwavering support in every aspect of my life - physical, psychological, mental, and spiritual. There are no words that can fully capture the gratitude I have for her, but I wish her all the best for a happy and fulfilling life. I also want to extend a special thank you to my girlfriend, Miss Yuhan Xie. Throughout this challenging journey, she has brought me an abundance of joy and happiness every day and night. She has been my pillar of support, encouraging me whenever I felt lost. Now, as I am approaching the end of my doctoral journey, it is my turn to support her in her own endeavors.

Yixi Cai

The University of Hong Kong

September 12, 2024

Contents

List of Publications

Relevant to Thesis

1. Wei Xu*, **Yixi Cai***, Dongjiao He, Jiarong Lin, and Fu Zhang. Fast-lio2: Fast direct lidar-inertial odometry. *IEEE Transactions on Robotics*, volume 38, pages 2053–2073. IEEE, 2022 (* is equally first author).

2. **Yixi Cai**, Fanze Kong, Yunfan Ren, Fangcheng Zhu, Jiarong Lin, and Fu Zhang. Occupancy grid mapping without ray-casting for high-resolution lidar sensors. *IEEE Transactions on Robotics*, volume 40, pages 172–192. IEEE, 2024.

Collaborative works

1. Fanze Kong, Wei Xu, **Yixi Cai**, and Fu Zhang. Avoiding dynamic small obstacles with onboard sensing and computation on aerial robots. *IEEE Robotics and Automation Letters*, volume 6, pages 7869–7876. IEEE, 2021.

2. Youming Qin, Nan Chen, **Yixi Cai**, Wei Xu, and Fu Zhang. Gemini ii: Design, modeling, and control of a compact yet efficient servoless bi-copter. *IEEE/ASME Transactions on Mechatronics*, volume 27, pages 4304–4315. IEEE, 2022. (**Best Paper Award Finalist 2022**)

3. Wei Xu, Dongjiao He, **Yixi Cai**, and Fu Zhang. Robots' state estimation and observability analysis based on statistical motion models. *IEEE Transactions on Control Systems Technology*, volume 30, pages 2030–2045. IEEE, 2022.

4. Nan Chen, Fanze Kong, Wei Xu, **Yixi Cai**, Haotian Li, Dongjiao He, Youming Qin, and Fu Zhang. A self-rotating, single-actuated uav with extended sensor field of view for autonomous navigation. *Science Robotics*, volume 8, page eade4538. American Association for the Advancement of Science, 2023.

5. Fanze Kong, Xiyuan Liu, Benxu Tang, Jiarong Lin, Yunfan Ren, **Yixi Cai**, Fangcheng Zhu, Nan Chen, and Fu Zhang. Marsim: A light-weight point-realistic simulator for lidar-based uavs. *IEEE Robotics and Automation Letters*, volume 8, pages 2954–2961. IEEE, 2023.

6. Jiarong Lin, Chongjian Yuan, **Yixi Cai**, Haotian Li, Yunfan Ren, Yuying Zou, Xiaoping Hong, and Fu Zhang. Immesh: An immediate lidar localization and meshing framework. *IEEE Transactions on Robotics*, volume 39, pages 4312–4331. IEEE 2023.

7. Yuying Zou, Haotian Li, Yunfan Ren, Wei Xu, Yihang Li, **Yixi Cai**, Shenji Zhou, and Fu Zhang. Perch a quadrotor on planes by the ceiling effect. *In 2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, pages 1–7. IEEE, 2023.

8. Guozheng Lu, **Yixi Cai**, Nan Chen, Fanze Kong, Yunfan Ren, and Fu Zhang, Trajectory Generation and Tracking Control for Aggressive Tail-Sitter Flights. *The International Journal of Robotics Research*, 2024.

9. Haotian Li, Yuying Zou, Nan Chen, Jiarong Lin, Wei Xu, Chunran Zheng, Xiyuan Liu, Dongjiao He, Fanze Kong, **Yixi Cai**, Zheng Liu, Shunbo Zhou, Kaiwen Xue, and Fu Zhang. A multi-sensor aerial robots slam dataset for lidar-visual-inertial-gnss fusion. *The International Journal of Robotics Research*, 2024.

10. Yunfan Ren, **Yixi Cai**, Fangcheng Zhu, Siqi Liang, Fu Zhang. ROG-map: An efficient robocentric occupancy grid map for large-scene and high-resolution LiDAR-based motion planning. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024.

11. Yunfan Ren, Fangcheng Zhu, Guozheng Lu, **Yixi Cai**, Longji Yin, Fanze Kong, Jiarong Lin, Nan Chen, Fu Zhang. Safety-assured High-speed Navigation for MAVs. *Science Robotics*, 2024 (In Revision).

12. Guozheng Lu, Yunfan Ren, Fangcheng Zhu, Haotian Li, Ruize Xue, **Yixi Cai**, Ximin Lyu, and Fu Zhang. Autonomous Tail-Sitter Flights in Unknown Environments. *IEEE Transactions on Robotics*, 2024 (In Revision).

13. Fangcheng Zhu, Yunfan Ren, Longji Yin, Fanze Kong, Qingbo Liu, Ruize Xue, Wenyi Liu, **Yixi Cai**, Guozheng Lu, Haotian Li, Fu Zhang. Swarm-LIO2: Decentralized, Efficient LiDAR-inertial Odometry for UAV Swarms. *Transactions on Robotics*, 2024 (In Revision).

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Mobile robots are becoming increasingly important in various scenarios, as they could potentially replace humans in performing challenging and laborious tasks. Researchers have been investigating the usage of robotics in the field of agriculture [1, 2]. Mobile robots are also deployed in hazardous or extreme environments, such as caves [3], subarctic glaciers [4], and nuclear radioactive facilities [5], as shown in Figure 1.1. Furthermore, mobile robots like the Curiosity rover and Ingenuity helicopter have been deployed for space exploration beyond Earth [6]. Scientists at ETH Zurich are also researching technologies for legged robots to conduct planetary exploration [7]. One of the key capabilities needed to achieve success of these challenging field tasks is the ability for robots to perceive and comprehend the surrounding environment, which relies on two core components: onboard sensors and mapping systems.

Onboard sensors serve as the robot's eyes to perceive the surrounding environment. One primary function of onboard sensors is to accurately determine the self-location of robots and measure distances to objects, thereby facilitating the fundamental task of navigation from one point to another. In addition, sensors are expected to capture implicit information from the environment, enabling a comprehensive understanding of the surroundings, including semantic details.

Since the beginning of modern robotics, cameras have played a pivotal role in various applications, including self-localization [8–11], occupancy mapping [12–14], metric-semantic mapping [15, 16], and scene understanding [17, 18]. These applications have

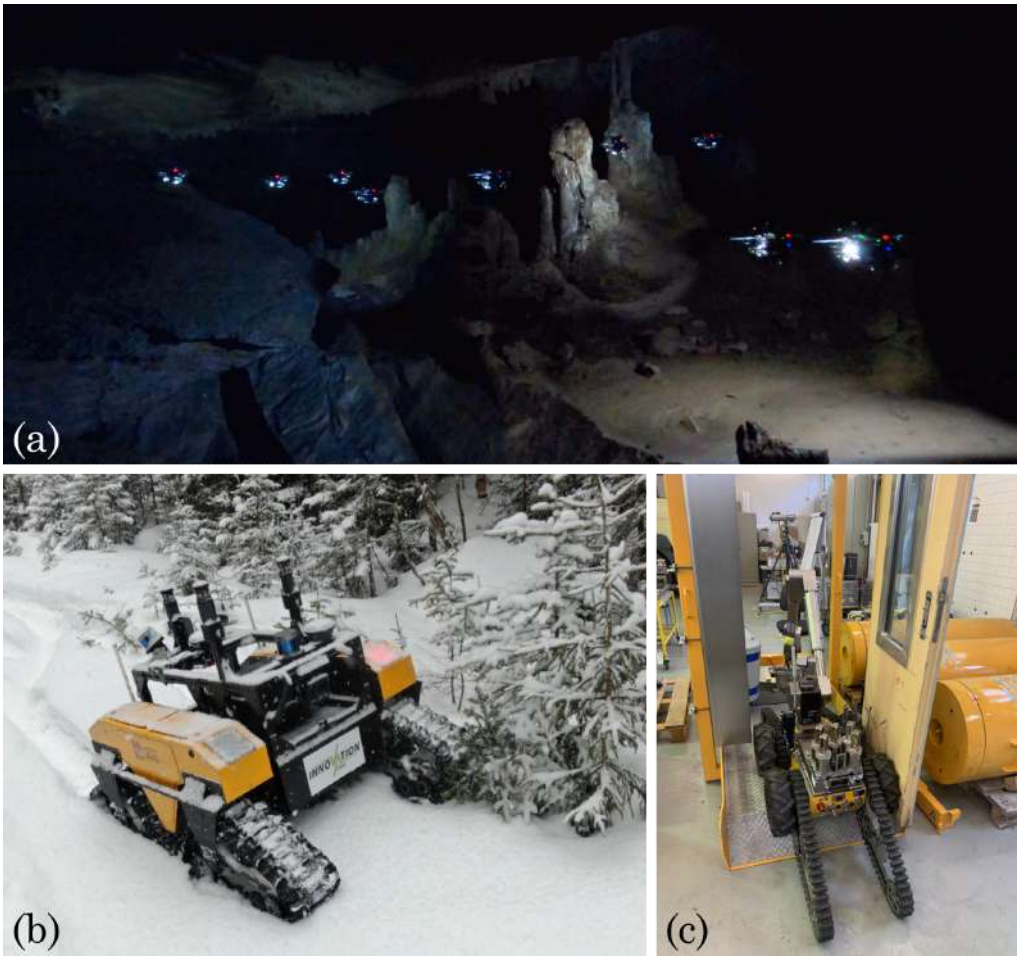


Figure 1.1: (a) A composite image demonstrates an unmanned aerial vehicle (UAV) conducting autonomous exploration in a cave, taken from [3]. (b) A mobile robot is navigating autonomously on a 0.7 m compacted snow cover in subzero temperatures, taken from [4]. (c) A mobile robot carrying a manipulator is navigating through a hazardous area to remove a contaminated object, taken from [5].

demonstrated the remarkable efficacy of cameras in providing crucial environmental information to robots, even in complex tasks. However, it is important to note that the performance of vision-based algorithms is affected by the illumination conditions of the environment. The robots in the dark need to carry their own light sources. Consequently, their practical utility is limited in scenarios with low light or inadequate illumination, such as caves or environments with limited light. Moreover, the presence of motion blur in captured images further restricts their applicability, particularly for high-dynamic robots such as unmanned aerial vehicles (UAVs).

In the past decade, advancements in 3D Light Detection and Ranging (LiDAR) sensors have led to the development of products that are cost-effective, lightweight, and energy-efficient. As a result, LiDARs have become a crucial sensor for robots which provides direct, active, dense, and accurate depth measurements [19, 20]. Unlike visual sensors that rely on passive mechanisms and incoming light to generate images, LiDAR sensors actively emit laser pulses and precisely measure the time it takes for these pulses to return after interacting with objects in the environment. This active approach enables LiDAR sensors to provide highly accurate depth measurements (e.g., centimeter accuracy at hundreds of meters measuring range [21]). Moreover, significant advancements in LiDAR technology have resulted in the generation of point clouds comprising several million points per second. This dense and accurate sampling of the environment enhances a robot's perception capabilities. Additionally, LiDAR sensors boast an extended detection range, surpassing that of depth cameras, often exceeding 100 meters [22]. Consequently, robots equipped with LiDAR sensors can achieve a broader field of view in their perception tasks. It is worth noting that early LiDARs were bulky due to mechanical rotating mechanisms. However, recent technological advancements have facilitated the commercialization and mass production of lightweight LiDARs, weighing as little as 260g, with affordable price points (e.g., hundreds of dollars) and exceptional performance [23]. These developments have positioned LiDAR sensors as suitable and promising sensors for integration into robots, offering immense potential for various applications.

With onboard sensors serving as the eyes on the robot, a mapping system acts as the brain for the robot to comprehend the surrounding information. Typically, a mapping system on a robot aims to build a general and persistent scene representation to include

multiple levels of information such as geometry, appearance, and semantics[24]. Given the strong ability of LiDARs to obtain accurate depth measurements from surroundings, this thesis focuses on a LiDAR mapping system aiming to create a scene representation that is close to real 3D geometry and understandable for robots. Specifically, this thesis addresses the efficiency issues in LiDAR mapping within two typical modules in robotics: Simultaneous Localization and Mapping (SLAM) and Occupancy Mapping. The former plays a crucial role in providing accurate self-localization to the robot, while the latter supports autonomous navigation by helping the robot to reason about the unknown and known regions of the environment.

1.2 LiDAR SLAM

Simultaneous Localization and Mapping (SLAM) is a crucial task in robotics that aims to create a real-time estimation of the shape and structure of the surrounding environment while simultaneously determining the robot's self-location [25]. The acronym "SLAM" was first introduced at the 1995 International Symposium on Robotics Research, where the formulation of the SLAM problem and its convergence result were presented [26]. Early research in SLAM primarily focused on utilizing visual sensors [27, 28], laser range finders [29, 30], and sonar sensors [31, 32] to address the problem. A significant breakthrough in LiDAR SLAM, LOAM[33], was presented at the 2014 Robotics: Science and Systems conference, where a systematic solution for real-time odometry and mapping was proposed. In LOAM[33], the complex problem of SLAM was divided into two sub-problems: odometry and mapping, which were solved in parallel at different rates. This section follows a similar division by discussing the state estimation and mapping of LiDAR SLAM separately, with a primary focus on the mapping part.

1.2.1 State Estimation

State estimation in LiDAR SLAM involves estimating a set of physical attributes of a robot, such as position, velocity, and orientation. Typically, there are two steps involved in estimating robot states using a LiDAR sensor. The first step involves finding correspondences between current LiDAR measurements and either the recent

scan (scan-to-scan matching) or the historical map (scan-to-map matching). Correspondences found through scan-to-scan matching provide an approximate estimation of the relative position to the recent scan, while scan-to-map matching provides a more accurate and reliable estimation with lower drift. However, the accuracy and efficiency of mapping heavily influence scan-to-map matching, which will be discussed in Chapter 3. The second step aims to optimize the robot state such that the residuals between the correspondence and the current measurement are minimized. Filter-based and graph optimization-based approaches have been extensively investigated in this regard. Filter-based methods, particularly Kalman filters and their variants, estimate the current state by assuming optimal estimation of previous states. These methods are favored for providing a high-frequency estimation of the robot's state due to their low computational complexity [34–38]. In contrast, graph optimization-based approaches optimize states over an interval, typically serving as the backend to achieve higher estimation accuracy at a lower rate [33, 39, 40].

1.2.2 Mapping

Mapping in LiDAR SLAM aims to incrementally build a geometric map of the unknown environment with high fidelity. LiDAR mapping involves two core components: the map representations and map structures. Specifically, the former involves abstracting information to represent the geometric structure, while the latter serves as the container to manage the various types of information that support access, update, and association.

1.2.2.1 Map Representations

The nature of LiDAR measurements is essentially point clouds which are sampled points of object surfaces in the surrounding environments. Consequently, the primary objective of map representations is to maximize the utilization of the sampled information from LiDAR measurements to reconstruct the geometric surfaces in high fidelity.

The most straightforward and commonly used representation of an environment for LiDAR mapping is to use the point cloud itself. In early research [33, 41, 42], a feature extraction module was employed to extract geometric points from each LiDAR scan, as shown in Figure 1.2. These features were then used to create a point cloud map for correspondence matching. This type of map representation has been widely adopted in

subsequent works [38, 40, 43–48]. However, there are three critical drawbacks associated with feature extraction. Firstly, the feature extraction module is limited in its generality, as it requires handcrafted designs to accurately extract geometric features from a single LiDAR scan. This poses challenges when dealing with LiDARs that have different scanning patterns or sparse measurements, making it difficult to extract geometric features effectively [49, 50]. Secondly, extracting features from each LiDAR scan results in non-feature points being discarded, leading to unexpected information loss. This loss of information reduces the efficacy of creating a high-fidelity geometric representation of the environments. Lastly, the feature extraction process incurs additional computational load, which is at least linearly proportional to the number of LiDAR points. This limits its ability to keep pace with the increasing number of measurements from LiDAR sensors.

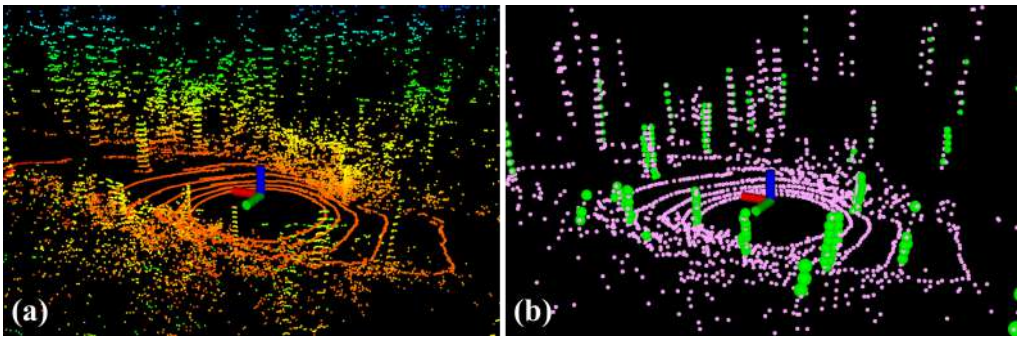


Figure 1.2: An illustration of feature extraction from point cloud, taken from [43]. (a) The original point cloud. (b) The green and pink points represent edge and planar features, respectively.

Another commonly used geometric representation for LiDAR mapping is planes with parameterized representations, as any curved surface in the environment can be approximated by a combination of small planes. An illustration of planar representation is shown in Figure 1.3. Surfel, an abbreviation acronym for a "surface element", was first introduced in [51] and widely applied in visual SLAM [52–57]. It is firstly introduced into the LiDAR SLAM community in [58], followed by several research studies aiming to improve its efficiency and accuracy. In [59] and [60], surfels were maintained in a multi-resolution map by merging valid ones from the finest solution to coarser solutions. Probabilistic representations of surfels were introduced in [61] and [62] to consider the noise in LiDAR measurements. Although surfels are a good representation of planar geometry, the extraction of surfels relies on projection to a range image, which is sensitive

to the sparsity and irregularity of LiDAR measurements. Instead of extracting planar features on projected range images, Yuan *et al.* proposed constructing a hierarchical voxel map in a coarse-to-fine manner, in which each voxel stores a plane with explicit parameterization[63]. The explicit parameters describe not only the position and normal of a plane but also the covariance, allowing for a probabilistic representation that considers the uncertainty from both pose estimation and LiDAR measurements. Wu *et al.* further enhance [63] by introducing a merging strategy of small planes that leads to increased accuracy and efficiency [64]. Liu *et al.* presented a general mathematical representation of point clouds, named point clusters, which can generalize to represent plane, edge, and point features without information loss [65]. Although planes with parameterized representations provide efficiency in both computation and memory with satisfactory accuracy, the drawback of plane representations in [63–65] is the voxelization process to decide the clustering of points to form a plane. This process has the possibility of incorrectly dividing points belonging to different planes into the same voxel due to the discretization error. Additionally, correspondence matching of planar features is also limited within a voxel since no nearest neighbor search is applied in these methods. Therefore, the scan matching requires a good initial guess of the robot's state to ensure that planar features of the current scan are placed into the correct voxel for matching.

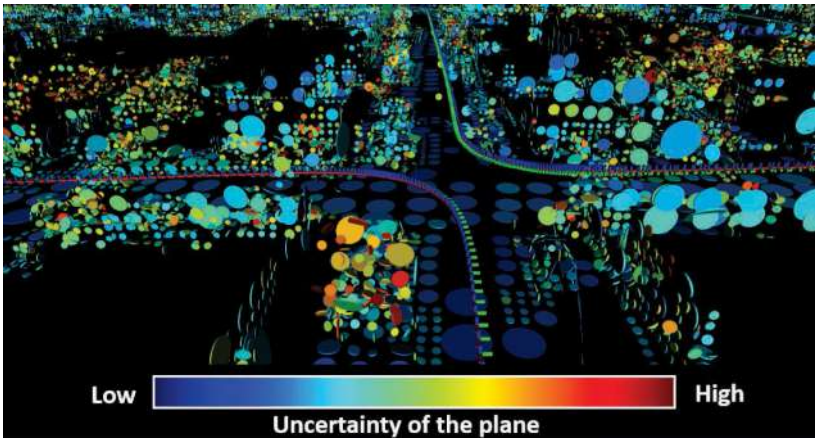


Figure 1.3: An illustration of the plane representation in [63].

Meshes are a widely-used representation of geometric structure in the computer graphics community comprised of faces, edges, and vertices, as shown in Figure 1.4. In the field of 3D geometry, triangle meshes have gained popularity due to the simplicity and ability to approximate complex environments effectively. Compared to point clouds

and planes, meshes offer several advantages. They avoid the voxelization problem and provide dense, smooth, and complete surface reconstructions. Moreover, meshes are memory-efficient and retain topological information of the environments [66, 67]. However, the application of meshes in LiDAR SLAM faces challenges in real-time construction and updates of a mesh map. KinectFusion [68] was a breakthrough to reconstruct a mesh online, which updates a truncated signed distance field (TSDF) and utilizes the Marching Cubes algorithm [69] to construct a triangle mesh. Subsequent research has improved this approach, enabling its usage in large-scale scenarios [54, 70], adaptive resolution [71, 72], and achieving better efficiency [73–75]. However TSDF-based methods were not computationally efficient, necessitating GPU acceleration for real-time mesh construction and updates. Instead of following the two-step pipeline, Vizzo *et al.* proposed leveraging Poisson surface reconstruction [76] directly on point clouds in LiDAR SLAM [77]. The mesh map in [77] is not incrementally updated at each scan but reconstructed from point clouds aggregating N historical scans. Another recent development by Lin *et al.* proposed ImMesh, which can directly and incrementally construct triangle mesh from LiDAR measurements in real-time on a CPU platform [78]. Although the localization in ImMesh leverages the technologies in [63], its real-time ability to construct mesh online lays a solid foundation for subsequent research utilizing mesh structures in LiDAR SLAM.

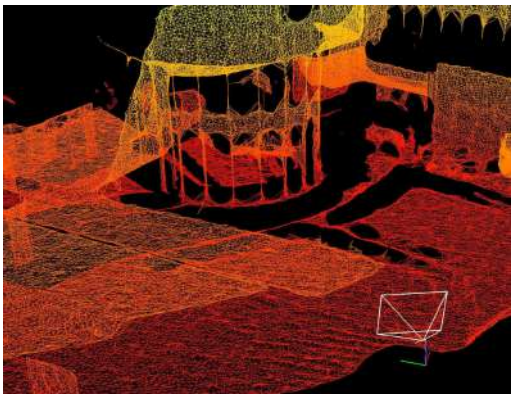


Figure 1.4: An illustration of the mesh representation in [78].

In addition to the map representations mentioned above, there are other representations in LiDAR SLAM that have been proposed. One such representation is the normal distribution transform (NDT) which fits a Gaussian distribution from the point cloud within a voxel [79–81], as shown in Figure 1.5(b). There were several studies exploring

the usage of NDT in LiDAR SLAM [82, 83]. Although NDT shows higher robustness in initial poses, its convergence is less predictable when compared with ICP [84], limiting its wider application. The Gaussian mixture model (GMM) was first introduced into LiDAR SLAM by Wolcott and Eustice, which characterized the distribution of z-height in each cell of a 2D grid structure [85, 86]. Later, [87, 88] introduced 3D mapping approaches that fit GMMs over 3D point cloud data, as shown in Figure 1.5(d). However, these methods suffered from the expensive computational cost of Expectation Maximization (EM) and sensitivity to initial parameters. Semantic information has also been incorporated to enhance LiDAR SLAM in approaches such as SuMa++ [89]. An illustration of the semantic LiDAR map is shown in Figure 1.5(e). SuMa++ used RangeNet++ [90] to generate semantic labels for each LiDAR point and integrates the refined labels into the surfel map. Furthermore, a semantic Iterative Closest Point (ICP) algorithm is proposed to improve pose estimation accuracy. The potential of semantic information in SLAM has recently been demonstrated in studies such as [15, 16]. However, a significant gap still exists in the widespread utilization of semantic information in LiDAR SLAM, primarily due to the limited generalizability of LiDAR segmentation models.

1.2.2.2 Map Structures

Map structures are crucial in LiDAR SLAM as they serve as containers for map representations, facilitating the access, update, and data association of geometric (and semantic) information. The functions of access and data association are both applicable in supporting scan-to-map matching in LiDAR SLAM. The data association function returns data with specific metrics (e.g., nearest neighbors), while the access function directly returns the data by index. The update operation incrementally integrates incoming information from LiDAR sensors to construct the map.

In the field of LiDAR SLAM, there are four commonly used data structures: arrays, hash tables, trees, and graphs. To gain a comprehensive understanding of these data structures, we analyze each based on computational efficiency for access, update, and data association, as well as their memory efficiency for data storage.

Arrays and hash tables are among the simplest data structures employed in LiDAR SLAM. Typically, the space is divided into dense axis-aligned voxels at a predetermined resolution, and map representations can be stored into an element of the array or hash

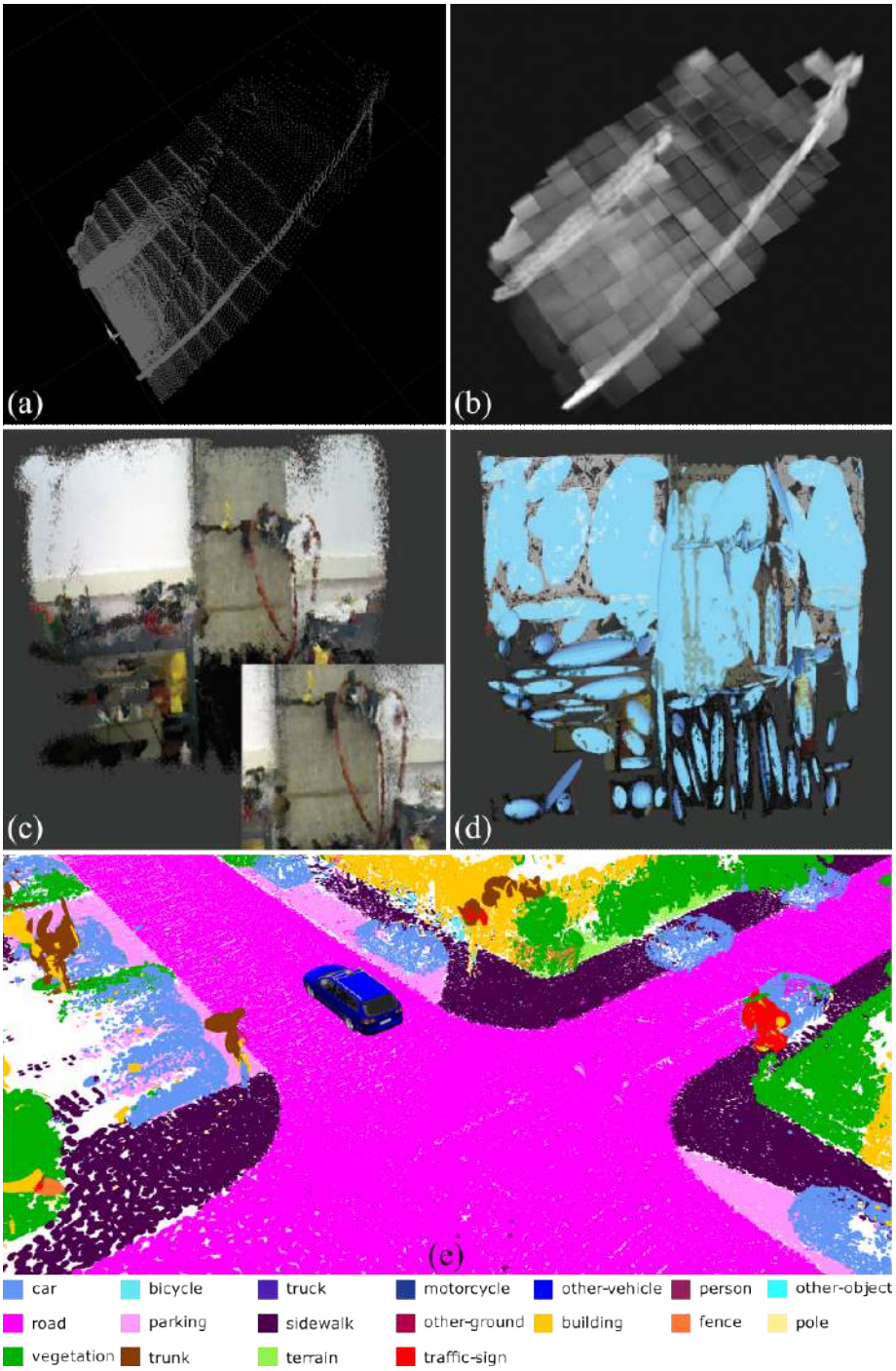


Figure 1.5: Illustrations of Various Map Representations. (a) depicts the original point cloud, while (b) showcases the NDT representation, taken from [81]. (c) exhibits the original colored point cloud, and (d) represents it using GMM, sourced from [88]. (e) presents the semantic map created in SuMa++ [89].

table by mapping the spatial location to the corresponding voxel index. The arrays and hash tables are suitable for managing information in real-time because of their high efficiency in data access and updates. However, since there is no explicit relationship between elements inside an array or a hash table, data association can be time-consuming, as it requires searching the entire storage to have a complete and accurate check of any possible association. Consequently, LiDAR SLAM algorithms that use arrays or hash tables as map structures often assume that the related data is stored either inside the same element [41, 63–65] or neighboring elements [91] to avoid expensive traversal over the entire storage. However, this assumption relies heavily on a good initial guess for the robot's state and may fail under aggressive motion or large drift. When comparing arrays against hash tables, although both require memory pre-allocation, the latter is more commonly used due to its more flexible memory allocation ability and higher memory efficiency. Hash conflicts are the main concern for hash tables, which, without a well-designed hashing function and algorithm, can lead to a significant decrease in computational efficiency.

Trees represent a form of data structure that stores information based on a given rule (e.g., an octree for hierarchical structuring or an ordered tree for sequential arrangement). Compared to arrays and hash tables, tree structures are more efficient in data association due to their ordered organization of information, but require a higher cost in data access and update to manage the data order. K-Dimensional trees (k-d trees) [92] and octrees [93] are two typical and popular tree structures that have been widely employed in LiDAR SLAM algorithms. A k-d tree is a binary tree that splits space by hyperplanes, and has been proven to be the most efficient data structure for nearest neighbor search in low dimensions [94], making it favored in many LiDAR SLAM algorithms for correspondence matching. An octree is a prefix-tree (also known as a trie in computer science) in which child nodes share the same prefix. When applied to space partitioning, an octree is particularly suitable for representing space in a hierarchical structure, in which the space represented by child nodes is encapsulated inside its ancestry nodes. When comparing k-d trees and octrees, the latter exhibits higher performance in access and update capabilities, while the former is more suitable for data association. The memory efficiency of k-d trees, which is $\mathcal{O}(n)$, is also better than that of octrees, which is $\mathcal{O}(n \log(n))$, where n is the number of spatial data points.

A graph is a structure composed of vertices and edges, which naturally provides a strong representation for the topological connection of different objects in an environment. This strong data association capability comes at the cost of time-consuming access and update operations, as traversing the entire graph is often required to find the corresponding vertices. Recent research in the visual SLAM community has utilized graphs to represent topological connections with the aid of semantic understanding [15, 16]. However, in LiDAR SLAM, graphs are typically used for pose graph optimization rather than representing the surrounding environment. When considering the incorporation of semantic information with geometric structures in LiDAR SLAM, a graph structure serves as a promising alternative for high-level organization.

In conclusion, Table 1.1 summarizes a qualitative comparison of the capabilities of different data structures in this section.

Table 1.1: Qualitative Comparison of Capabilities among Different Map Structures

	Arrays	Hash Tables	Trees		Graph
			octree	k-d tree	
Access and Update	Strongest	Strong	Moderate	Weak	Weakest
Data Association	Weakest	Weak	Moderate	Strong	Strongest
Memory Efficiency	Weakest	Weak	Moderate	Strong	Strong

1.2.3 Challenges

The high accuracy of depth measurements in LiDAR sensors inherently offers significant potential for enhancing the precision of LiDAR SLAM. However, the sparsity of measurements in LiDAR, as compared to visual sensors, introduces new challenges in LiDAR mapping. In visual SLAM, a sparse map is sufficient to enable the scan matching due to the dense measurements of the visual sensors, as shown in Figure 1.6(a). In contrast, LiDAR SLAM necessitates a dense map for effective scan matching due to its sparse measurements, as shown in Figure 1.6(b). As a result, map representations for LiDAR sensors must capture the geometry of environments as densely as possible.

Furthermore, maintaining a dense map significantly increases the computational load on the map structure, which has become a bottleneck that restricts the advancement of LiDAR SLAM. Considering the limited computational resources in mobile robots, the

development of a more efficient map structure is essential. This would allow for more computational power to manage denser and more complex maps, ultimately enhancing the accuracy and robustness of LiDAR SLAM.

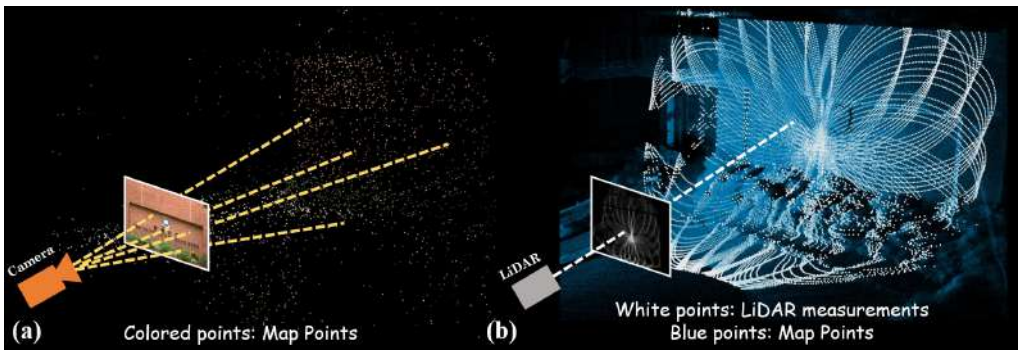


Figure 1.6: An illustration of maps with different sparsity. (a) showcases a feature point map using visual SLAM. (b) visualizes the point cloud map (colored by intensity) and the sparse measurements (white points) created using a LiDAR sensor.

1.3 Occupancy Mapping

Occupancy mapping addresses the problem of creating consistent maps from noisy and uncertain measurement data, under the assumption that the robot pose is known from a odometry algorithm [95]. Different from maps in LiDAR SLAM that only represent geometric structures, the maps in occupancy mapping are volumetric maps that represent occupied, free, and unknown spaces of the environment. In this section, we separately discuss the discrete and continuous occupancy representations, followed by popular map structures to manage the occupancy information.

1.3.1 Occupancy Representations

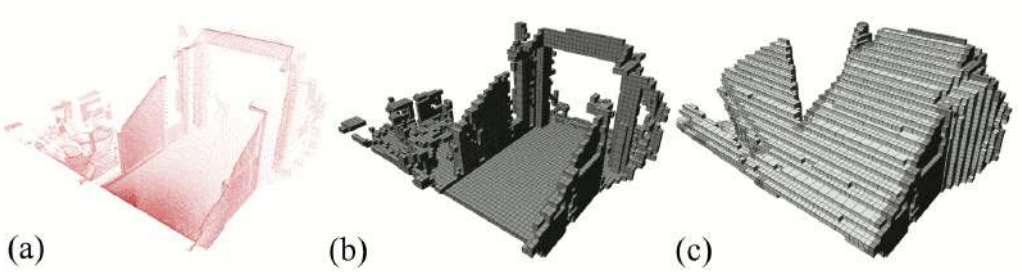


Figure 1.7: An illustration of discrete representations sourced from [96]. (a) depicts the original point cloud. (b) visualizes only the occupied voxels in the octree. In (c), white and black voxels represent free and occupied space, respectively.

Discrete representations are widely used for occupancy mapping due to their simplicity and efficiency. A common discrete representation partitions the space into evenly distributed 2D grids or 3D voxels, where each grid or voxel is represented by a Bernoulli random variable indicating its occupancy probability. By assuming spatial independence between neighboring voxels, this representation allows for efficient updates of occupancy probabilities on each voxel. This approach is known as occupancy grid mapping. Early research was conducted in [97] and [98], while the concept was systematically introduced in [95]. The occupancy grid mapping framework was further extended to 3D using octree in Octomap [96], which leveraged a log-odd function to reduce computation complexity. Figure 1.7 depicts an illustration of the discrete representation using Octomap. Although subsequent works mainly focus on addressing issues related to memory consumption and computational efficiency from the map structure side, to the best of the author's knowledge, there have been very few significant modifications or improvements made to the probability representation itself in occupancy grid mapping. Another discrete representation is the Euclidean signed distance field (ESDF) that computes the Euclidean distance to the closest surface at each evenly distributed voxel. While most of the research obtains ESDF representations from the occupancy grid map, Voxblox [99] directly builds a ESDF map from TSDF data. However, Voxblox suffers from inaccuracy in the estimation of Euclidean distance because of the approximated conversion from TSDF TO ESDF data [100].

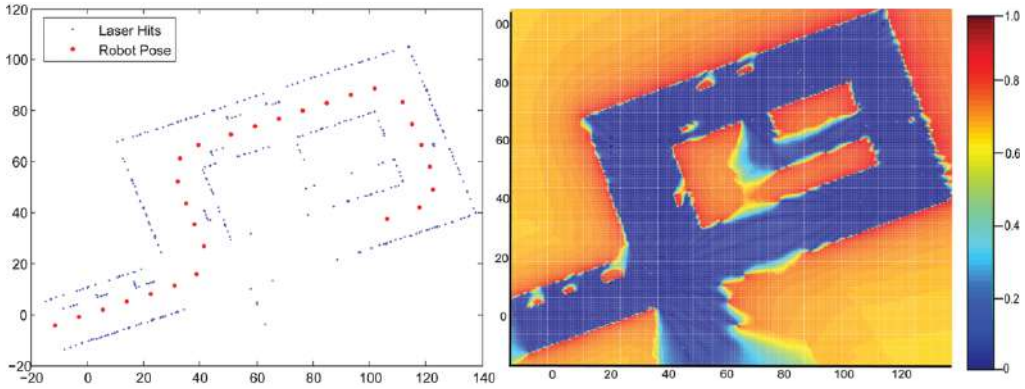


Figure 1.8: An 2D illustration of continuous map representations taken from [101]. (a) depicts the laser measurements (blue dots) and the robot pose (red dots). (b) visualizes the occupancy probability respect to location in Gaussian process occupancy mapping.

Continuous map representations assume spatial correlation in neighboring spaces

considering the physical structure of environments. A 2D illustration of the continuous representation is shown in Figure 1.8. The earlier attempts by [102, 103] trained curves and polygon random fields from range samplings to represent the boundaries and occupancy probabilities of the environments. O'Callaghan and Ramos proposed a non-parametric approach using Gaussian processes to estimate continuous occupancy distributions in 2D space [101]. However, Gaussian processes require storing all N measurements from an entire mapping dataset to estimate the covariance function and an invert operation with time complexity of $\mathcal{O}(N^3)$ to predict the occupancy of a test position. Although the authors mentioned that clustering the measurements in proximity to create a local model could avoid using all measurements, a new covariance matrix must be built at each query of a new position. Subsequent works such as GPmap [104], GPOctomap [105] and BGKOctomap-L [106] extended the Gaussian process occupancy maps into 3D and utilized octrees [93] for space partitioning, but still relied on storing raw measurements for occupancy prediction.

Apart from non-parametric approaches, semi-parametric methods cluster raw measurements into parameterized functions (e.g., Gaussians and kernels) instead of storing the raw measurements themselves. NDT-OM [107] partitioned space into voxels, with each voxel containing a Gaussian to represent the measurements within it, but faced severe discretization errors in unstructured environments. A similar approach, confidence-rich mapping, considered the correlation within a measurement cone of a sensor to provide uncertainty estimation for planning and exploration [108]. The Hilbert map [109] and its variants [110–112] utilized kernel feature approximations to reduce the time complexity of inference down to $\mathcal{O}(N)$. Occupancy maps based on Gaussian mixture models were also investigated, variants using hierarchical structures [113] and various resolution [114].

Although continuous occupancy mapping approaches effectively consider spatial correlation and provide uncertainty estimation, there remains a significant gap in computational efficiency compared to discrete representations, which hinders their widespread deployment in real robotic systems.

1.3.2 Occupancy Map Structures

As discrete representations divide the space into evenly distributed 2D cells or 3D voxels, the early research utilized arrays to store grids, typically known as uniform grid maps [115]. However, a critical drawback of the uniform grid maps is their tremendous memory consumption, which prevents their application in high-resolution and large-scale mapping. Nonetheless, uniform grid maps offer the strongest efficiency in updating and querying compared to other map structures to be discussed in this section, due to their continuous memory allocation. Thus, uniform grid maps are well-suited for occupancy mapping in a local space [116].

To reduce the memory consumption in uniform grid maps, hashing techniques were introduced to organize the cells (or voxels) [117]. Rather than pre-allocating memory for each cell (or voxel), the hashing-based grid map allocates a smaller array. The hashing technique uses a hash function to generate a hash value for each voxel to indicate its index in the array. The design of the hashing function is critical to minimizing the conflict rate, and a smart strategy is also required to deal with possible conflicts to avoid information loss. The hashing grid map allows dynamic map resizing through rehashing and reallocation, thus not requiring knowledge of the mapping area beforehand [99]. Compared to uniform grid maps, hashing grid maps possess higher memory efficiency and better dynamic ability. However, the computational efficiency of hashing grid maps is lower than that of uniform grid maps due to hash conflicts and a lower cache hit rate [118].

Quadtrees and octrees are useful data structures for organizing voxels at various resolutions and have become popularly used in occupancy grid mapping [96, 119]. These tree-based map structures exhibit superior memory efficiency compared to uniform grid maps and hashing grid maps, making them favorable for high-resolution maps and large-scale environments. However, the time complexity for updating occupancy probabilities in a tree structure is logarithmic, while that in grid maps is constant. Consequently, subsequent research has focused on enhancing update efficiency without compromising original accuracy and memory efficiency [72, 120–124]

The occupancy map structures for continuous representation are similar to those

for discrete representations. Grid maps and tree-based maps are used to store raw measurements [104–106] or parameterized clusters of raw measurements [107, 108, 113, 114]. Another data structure worth mentioning is the R-tree [125] which is used to organize GMMs without evenly partitioning the space, thus avoiding discretization errors [126].

1.3.3 Challenges

Although previous research in the field of occupancy mapping has been deployed on LiDAR sensors, there remains significant challenges in the computation efficiency due to the long detection range and dense measurements of LiDAR sensors. Furthermore, to fully exploit the high accuracy offered by LiDAR sensors, there is a preference for high-resolution maps to accurately represent the environment, either in the form of a high-resolution occupancy grid map or a continuous map. However, these requirements introduce a substantial computational burden. This challenge becomes even more pronounced when attempting to implement existing occupancy mapping approaches on robotic systems, as the computational resources are often strictly limited due to payload constraints, such as on unmanned aerial vehicles (UAVs). Consequently, there remains an unsolved problem of how to further enhance computational and memory efficiency to enable LiDAR-based occupancy mapping on robots.

1.4 Thesis Outline

This thesis presents novel designs in map representations and structures to tackle the efficiency challenges encountered in LiDAR SLAM and occupancy mapping. The first contribution is the introduction of an incremental k-d tree (ikd-Tree), a new data structure that enables efficient and dynamic map management. Subsequently, the high efficiency of the ikd-Tree is fully leveraged in the development of a LiDAR-inertial framework called FAST-LIO2, leading to significant improvements in efficiency, accuracy, and robustness compared to state-of-the-art methods. Additionally, this thesis proposes a novel occupancy mapping structure called D-Map to address efficiency issues in occupancy mapping for high-resolution LiDAR sensors. The remaining chapters of this thesis are organized as follows:

Chapter 2 addresses the limited ability for static k-d trees to manage a stream of sequentially LiDAR point clouds coming from the sensor. This chapter introduces the ikd-Tree, which includes incremental functions such as point-wise updates and box-wise updates that are useful in robotic applications. As these incremental updates can degrade the tree's efficiency for the destroyed balance property, it is crucial to preserve its balance following updates to maintain high computational efficiency in a k-d tree. The ikd-Tree monitors the balance property online and utilizes a twin-threaded re-balancing strategy to reconstruct unbalanced (sub-)trees when they exceed a given balance threshold. Additionally, the ikd-Tree is equipped with an on-tree down-sampling mechanism that limits the tree size in bounded environments. The experimental results show that the proposed ikd-Tree achieves superior performance in terms of computational efficiency compared to traditional k-d trees. The effectiveness of the ikd-Tree in managing sequentially incoming LiDAR point clouds is demonstrated in real-world experiments.

Chapter 3 introduces a fast, robust, and versatile LiDAR-inertial odometry framework named FAST-LIO2. The proposed framework leverages the ikd-Tree in the mapping module to effectively manage a point cloud map as the geometric representation of the environment. Specifically, the use of ikd-Tree allows for incremental registration of incoming point clouds and facilitates the management of a local map at a bounded size via the box-wise delete function with a logarithmic time complexity. The dynamic balancing ability of the ikd-Tree ensures consistent computational efficiency in incremental updates and nearest neighbor search. As a result, the efficient mapping module enables the usage of raw points in FAST-LIO2 without the need for feature extraction. This approach significantly enhances the overall efficiency, accuracy, and robustness of the proposed framework. FAST-LIO2 demonstrates superiority in extensive benchmark experiments when compared to state-of-the-art LIO frameworks. The effectiveness of FAST-LIO2 in accurately estimating a robot's pose and mapping the environment in real-time is also validated through real-world experiments.

Chapter 4 proposes a novel occupancy mapping framework, termed D-Map, to exploit the full potential of LiDAR sensors. The recent advancements in LiDAR technology have led to longer detection ranges, denser measurements, and higher accuracy. However, significant challenges remain in the computational and memory efficiency required to process large amounts of LiDAR measurements in a high-resolution and large-scale

occupancy map. D-Map proposes a depth image projection-based approach to determine the occupancy state, followed by an on-tree update strategy in a hybrid map structure to achieve high efficiency in occupancy updates. The hybrid map structure contains a hashing grid map for efficient updates of occupied space and an octree for managing dense known space with high memory efficiency. The high accuracy and low false alarm rate of LiDAR sensors are also leveraged to decrease the map size of D-Map which further enhances both computational and memory efficiency. The results of benchmark and real-world experiments demonstrate the effectiveness of D-Map in improving the computational and memory efficiency while maintaining comparable mapping accuracy.

Chapter 5 concludes this thesis by summarizing the contents and highlighting the contributions of the proposed mapping approaches. Limitations of these approaches are also discussed to provide a comprehensive understanding of their scope and potential for future research. Based on previous research experiences and insights, three potential directions for future work are discussed. The first direction involves creating a consistent map that is robust to inaccurate pose estimation, dynamic scenarios, and temporal changes. The second direction involves pursuing a more comprehensive understanding of surrounding environments through multi-modal sensor fusion and collaborative mapping. The last direction involves exploring heterogeneous computational structures for mapping, discussing possible approaches to utilizing a combination of parallel and serial computation architectures along with their advantages.

Chapter 2

ikd-Tree: An Incremental K-D Tree for Robotic Applications

In this chapter, we will present an efficient data structure, ikd-Tree, for dynamic space partition. The ikd-Tree incrementally updates a k-d tree with new coming points only, leading to much lower computation time than existing static k-d trees. Besides point-wise operations, the ikd-Tree supports several features such as box-wise operations and down-sampling that are practically useful in robotic applications. In parallel to the incremental operations (i.e., insert, re-insert, and delete), ikd-Tree actively monitors the tree structure and partially re-balances the tree, which enables efficient nearest point search in later stages. The ikd-Tree is carefully engineered and supports multi-thread parallel computing to maximize the overall efficiency.

2.1 Introduction

The K-Dimensional Tree (K-D Tree) is an efficient data structure that organizes multi-dimensional point data [92] which enables fast search of nearest neighbors, an essential operation that is widely required in various robotic applications [127]. For example, in LiDAR odometry and mapping, k-d tree-based nearest point search is crucial to match a point in a new LiDAR scan to its correspondences in the map (or the previous scan) [33, 43, 50, 128–130]. Nearest point search is also important in motion planning for fast obstacle collision check on point-cloud, such as in [131–136].

Commonly-used k-d tree structure in robotic applications [137] is "static", where the tree is built from scratch using all points. This contradicts with the fact that the data is usually acquired sequentially in actual robotic applications. In this case, incorporating a frame of new data to existing ones by re-building the entire tree from scratch is typically very inefficient and time-consuming. As a result, k-d trees are usually updated at a low frequency [33, 43, 50] or simply re-built only on the new points [133, 134].

To fit the sequential data acquisition nature, a more natural k-d tree design would be updating (i.e., insert and delete) the existing tree locally with the newly acquired data. The local update would effectively eliminate redundant operations in re-building the entire tree, and save much computation. Such a dynamic k-d tree is particularly promising when the new data is much smaller than existing ones in the tree.

However, a dynamic k-d tree suitable for robotic applications brings several challenges: 1) It should support not merely efficient point operations such as insertion and delete but also space operations such as point-cloud down-sampling; 2) It easily grows unbalanced after massive point or space operations which deteriorates efficiency of queries. Hence re-building is required to re-balance the tree. 3) The re-building should be sufficiently efficient to enable real-time robotic applications.

Even though some existing dynamic data structure can satisfy small amount of specific requirements in robotics, an efficient and dynamic space partition data structure is still in need, especially for applications on small scale mobile robots such as UAVs.

In this chapter, we propose a dynamic k-d tree structure called ikd-Tree, which builds and incrementally updates a k-d tree with new points only while simultaneously down-samples them into the desired resolution. It supports incremental operations including insertion, re-insertion, and delete of a single point (i.e., point-wise) or a box of points (i.e., box-wise). The tree is automatically re-balanced by partial re-building. To preserve efficient real-time tree update, ikd-Tree separates the tree re-building into two parallel threads when necessary. This paper also provides a complete time complexity analysis for all tree updates, including both incremental operations and re-balancing. The time complexity of ikd-Tree is reduced substantially as verified on both random data and real-world point-cloud in LiDAR-inertial mapping applications. The ikd-Tree

is open sourced at Github[1]. Fig. 2.1 illustrates the incremental updates and re-balancing on the tree from a 3-D view.

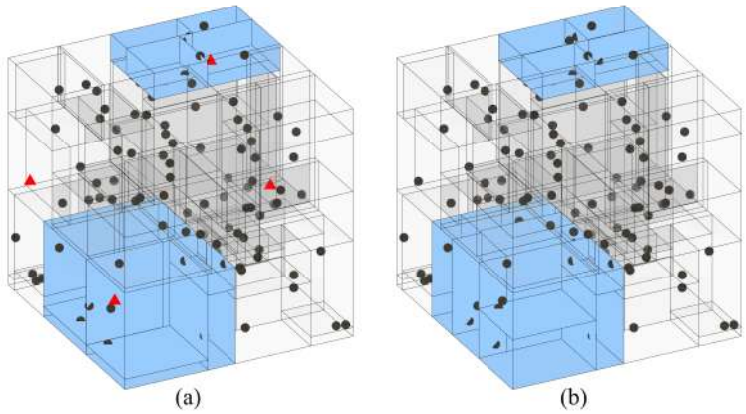


Figure 2.1: Illustration of incremental k-d tree update and re-balancing. (a): an existing k-d tree (black dots) and new points (red triangles) to insert, blue cubes denote the space (i.e., branches) need to be re-balanced. (b): the k-d tree after points insertion and tree re-balancing, blue cubes denote the space after re-balancing while rest majority tree does not change. Video available at https://youtu.be/ue0unk03zxA

This chapter is organized as follows: Section 2.2 introduces related work. The design of ikd-Tree is described in Section 2.3. Theoretical analysis of time and space complexity are presented in Section 2.4. Experiments are shown in Section 2.5, followed by conclusions in Section 2.6.

2.2 Related work

A k-d tree can be viewed as a binary search tree and inherits the same incremental operations (i.e., insert, re-insert, and delete), such as those in AVL trees [138], treaps [139] and splay trees [140]. In these techniques, re-balancing of a binary search tree after many point operations can be easily achieved by tree node rotations. However, this straightforward tree rotation only works for one-dimensional data. For k-d trees with higher data dimension, it requires much more complicated tree reorganization.

Strategies specifically designed for fast re-balancing k-d trees fall into two categories: hardware-based acceleration and specially designed structure enabling dynamic re-balancing. Hardware-based algorithms exploits the computing hardware to (re-) balance a kd-tree by building a new one. It has been thoroughly investigated to solve

[1]Git: https://github.com/hku-mars/ikd-Tree.git

the ray tracing problem in dynamic scenes. In 3D graphic applications, algorithms on single-core CPU [141, 142] and multi-core CPU [143] are firstly proposed to speed up the k-d tree construction. Zhou *et al.* proposed a real-time construction algorithm on GPU [144]. These algorithms rely heavily on the computing resource which is usually limited on onboard computers.

For the second category, Bentley *et al.* proposed a general binary transformation method for converting a static k-d tree to a dynamic one [145]. The dynamic k-d tree supports only insertion but not delete, which leads to a growing tree size hence increased time for nearest search. Galperin *et al.* [146] proposes a scapegoat k-d tree that can dynamically re-balance the tree by re-building unbalanced sub-trees, which is much more efficient than a complete re-building of the entire tree. Bkd tree [147] is a dynamic data structure extended from a K-D-B tree [148] which focus on external memory adaptions. A set of static k-d trees are built in the Bkd tree where the trees are re-balanced by re-building partial set of the trees at regular intervals. The well-known point cloud library (PCL) [137] uses the fast library for approximate nearest neighbors (FLANN) search [149]. Point insertion and delete are supported in FLANN but the trees are re-balanced via inefficient complete tree re-building after a predetermined amount of points insertion or delete [150].

Our ikd-Tree is an efficient and complete data structure enabling incremental operations (i.e., insert, re-insert, and delete) and dynamic re-balancing of k-d trees. Compared to the dynamic k-d tree in [145], our implementation supports points delete. Besides the point-wise operations presented in [138–140] and [146], our ikd-Tree further supports the incremental operations of a box of points (i.e., box-wise operations) and simultaneous points down-sampling. The dynamic tree re-balancing strategy of ikd-Tree follows the concept of scapegoat trees in [146], which only re-builds those unbalanced sub-trees. The ikd-Tree is particularly suitable for robotic applications, such as real-time LiDAR mapping and motion planning, where data are sampled sequentially and fast incremental update is necessary.

2.3 ikd-Tree Design and Implementation

In this section, we describe how to design, build, and update an incremental k-d tree in ikd-Tree to allow incremental operations (e.g., insertion, re-insertion, and delete) and dynamic re-balancing.

2.3.1 Data Structure

The attributes of a tree node in ikd-Tree is presented in **Data Structure 1**. Line 2-4 are the common attributes for a standard k-d tree. The attributes $leftchild$ and $rightchild$ are pointers to its left and right child node, respectively. The point information (e.g., point coordinate, intensity) are stored in $point$. Since a point corresponds a single node on a k-d tree, we will use points and nodes interchangeably. The division axis is recorded in $axis$. Line 5-7 are the new attributes designed for incremental updates detailed in Section 2.3.3.

Data Structure 1: Tree node structure

```
1 Struct TreeNode:
      // Common Attributes in Standard K-D trees
2     PointType point;
3     TreeNode * leftchild, rightchild;
4     int axis;
      // New Attributes in ikd-Tree
5     int treesize, invalidnum;
6     bool deleted, treedeleted, pushdown;
7     float range[k][2];
8 end
```

2.3.2 Building An Incremental K-D Tree

Building an incremental k-d tree is similar to building a static k-d tree except maintaining extra information for incremental updates. The entire algorithm is shown in **Algorithm 1**: given a point array V, the points are firstly sorted by the division axis with maximal covariance (Line 4-5). Then the median point is saved to $point$ of a new tree node T (Line 6-7). Points below and above the median are passed to the left and right child node of T, respectively, for recursive building (Line 9-10). The `LazyLabelInit` and `Pullup` in Line 11-12 update all attributes necessary for incremental updates (see **Data Structure 1**, Line 5-7) detailed in Section 2.3.3.

Algorithm 1: Build a balanced k-d tree

Input: V, N ▷ Point Array and Point Number
Output: $RootNode$ ▷ K-D Tree Node

1 $RootNode$ = `Build`$(V, 0, N-1)$;
2 **Function** `Build`(V, l, r)
3 $mid \leftarrow \lfloor (l+r)/2 \rfloor$;
4 $Axis \leftarrow$ Axis with Maximal Covariance;
5 $V \leftarrow sort(V, axis)$;
6 Node T;
7 $T.point \leftarrow V[mid]$;
8 $T.axis \leftarrow Axis$;
9 $T.leftchild \leftarrow$ `Build`$(V, l, mid-1)$;
10 $T.rightchild \leftarrow$ `Build`$(V, mid+1, r)$;
11 `LazyLabelInit`(T);
12 `Pullup`(T);
13 **return** T;
14 **End Function**

2.3.3 Incremental Updates

The incremental updates refer to incremental operations followed by a dynamic re-balancing detailed in Section 2.3.4. The incremental operations include insertion, delete and re-insertion of points to/from the k-d tree. Specifically, the insertion operation appends a new point (i.e., a node) to the k-d tree. In the delete operation, we use a lazy delete strategy. That is, the points are not removed from the tree immediately but only labeled as "deleted" by setting the attribute *deleted* to true (see **Data Structure 1**, Line 6). If all nodes on the sub-tree rooted at T have been deleted, the attribute *treedeleted* of T is set to true. Therefore the attributes *deleted* and *treedeleted* are called lazy labels. If points labeled as "deleted" but not removed are later inserted to the tree, it is referred to as "re-insertion" and is efficiently achieved by simply setting the *deleted* attribute back to false. Otherwise, points labeled as "deleted" will be removed from the tree during re-building process (see Section 2.3.4).

Our incremental updates support two types: point-wise updates and box-wise updates. The point-wise updates insert, delete, or re-insert a single point on the tree while the box-wise updates insert, delete or re-insert all points in a given box aligned with the data coordinate axis. Box-wise updates may require to delete or re-insert an entire sub-tree rooted at T. In this case, recursively updating the lazy labels *deleted* and *treedeleted* for all offspring nodes of T are still inefficient. To address this issue, we use a further lazy strategy to update the lazy labels of the offspring nodes. The lazy

label for lazy labels *deleted* and *treedeleted* is *pushdown* (see **Data Structure 1**, Line 6). The three labels *deleted*, *treedeleted*, and *pushdown* are all initialized as false in `LazyLabelInit` (see **Algorithm 1**, Line 11).

2.3.3.1 Pushdown and Pullup

Two supporting functions, `Pushdown` and `Pullup`, are designed to update attributes on a tree node T. The `Pushdown` function copies the labels *deleted*, *treedeleted*, and *pushdown* of T to its children (but not further offsprings) when the attribute *pushdown* is true. The `Pullup` function summarizes the information of the sub-tree rooted at T to the following attributes of node T: *treesize* (see **Data Structure 1**, Line 5) saving the number of all nodes on the sub-tree, *invalidnum* saving the number of nodes labelled as "deleted" on the sub-tree, and *range* (see **Data Structure 1**, Line 7) summarising the range of all points on the sub-tree along coordinate *axis*, where k is the points dimension.

2.3.3.2 Point-wise Updates

The point-wise updates on the incremental k-d tree are implemented in a recursive way which is similar to the scapegoat k-d tree [146]. For point-wise insertion, the algorithm searches down from the root node recursively and compares the coordinate on division axis of the new point with the points stored on the tree nodes until a leaf node is found to append a new tree node. For delete or re-insertion of a point P, the algorithm finds the tree node storing the point P and modifies the attribute *deleted*. Further details can be found in our Github repository[1].

2.3.3.3 Box-wise Updates

The box-wise insertion is implemented by inserting the new points one by one into the incremental k-d tree. Other box-wise updates (box-wise delete and re-insertion) are implemented utilizing the range information in attribute *range*, which forms a box C_T, and the lazy labels on the tree nodes. The pseudo code is shown in **Algorithm 2**. Given the box of points C_O to be updated on (sub-) tree rooted at T, the algorithm first passes down its lazy labels to its children for further passing-down if visited (Line 2). Then, it searches the k-d tree from its root node recursively and checks whether

Algorithm 2: Box-wise Updates

Input: C_O ▷ Operation box
T ▷ K-D Tree Node
SW ▷ Switch of Parallelly Re-building

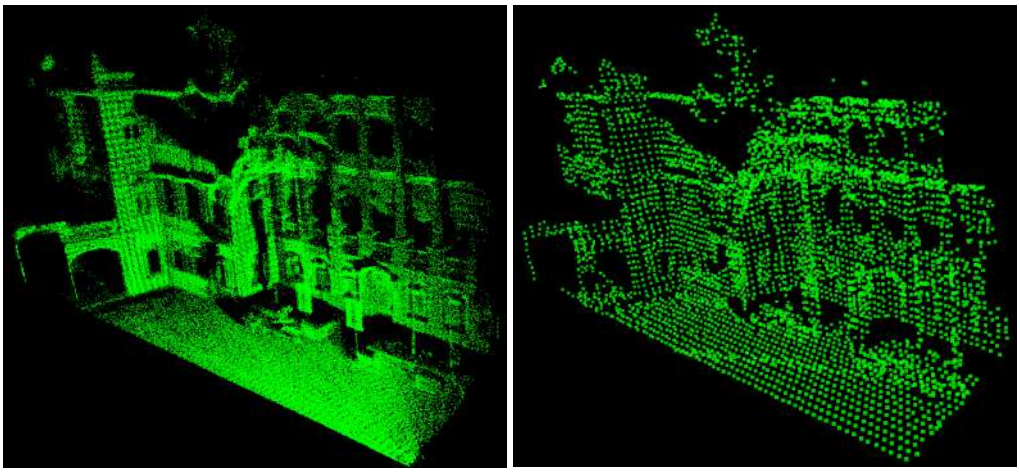
Function `BoxwiseUpdate`(T, C_O, SW)
  `Pushdown`(T);
  $C_T \leftarrow T.range$;
  if $C_T \cap C_O = \varnothing$ **then return**;
  if $C_T \subseteqq C_O$ **then**
    `UpdateLazyLabel()`;
    $T.pushdown$ = true;
    return;
  else
    $P \leftarrow T.point$;
    if $P \subset C_O$ **then** Modify $T.deleted$;
    `BoxwiseUpdate`$(T.leftchild, C_O, SW)$;
    `BoxwiseUpdate`$(T.rightchild, C_O, SW)$;
  end
  `Pullup`(T);
  if `ViolateCriterion`(T) **then**
    if $T.treesize < N_{\max}$ ***or*** *Not* SW **then**
      `Rebuild`(T)
    else
      `ThreadSpawn(ParallelRebuild,` T`)`
    end
  end
End Function

the range C_T on the (sub-)tree rooted at the current node T has an intersection with the box C_O. If there is no intersection, the recursion returns directly without updating the tree (Line 4). If the box C_T is fully contained in the box C_O, the box-wise delete set attributes *deleted* and *treedeleted* to true while the box-wise re-insertion set them to false by function `UpdateLazyLabel` (Line 6). The *pushdown* attribute is set to true indicating that the latest incremental updates have not been applied to the offspring nodes of T. For the condition that C_T intersects but not contained in C_O, the current point P is firstly deleted from or re-inserted to the tree if it is contained in C_O (Line 11), after which the algorithm looks into the child nodes recursively (Line 12-13) and updates all attributes of the current node T (Line 15). Line 16-22 re-balance the tree if certain criterion is violated (Line 16) by re-building the tree in the same (Line 18) or a separate (Line 20) thread. The function `ViolateCriterion`, `Rebuild` and `ParrallelRebuild` are detailed in Section 2.3.4.

Algorithm 3: Downsample

Input: L ▷ Length of Downsample Cube
P ▷ New Point

1 $C_D \leftarrow$ `FindCube`(L, P)
2 $P_{center} \leftarrow$ `Center`(C_D);
3 $V \leftarrow$ `BoxwiseSearch`$(RootNode, C_D)$;
4 $V.push(P)$;
5 $P_{nearest} \leftarrow$ `FindNearest` (V, P_{center});
6 `BoxwiseDelete`$(RootNode, C_D)$
7 `PointwiseInsert`$(RootNode, P_{nearest})$;



(a) (b)

Figure 2.2: Point Cloud Downsample. (a): the point cloud before down-sampling. (b): the point cloud after down-sampling

2.3.3.4 Downsample

Our ikd-Tree further supports down-sampling as detailed in **Algorithm 3**. For the given point P and down-sampling resolution L, the algorithm partitions the space evenly into cubes of length L, then the box C_D that contains point P is found (Line 1). The algorithm only keeps the point that is nearest to the center P_{center} of C_D (Line 2). This is achieved by firstly searching all points contained in C_D on the k-d tree and stores them in a point array V together with the new point P (Line 3-4). The nearest point $P_{nearest}$ is obtained by comparing the distances of each point in V to the center P_{center} (Line 5). Then existing points in C_D are deleted (Line 6), after which the nearest point $P_{nearest}$ is inserted to the k-d tree (Line 7). The implementation of box-wise search is similar to the box-wise delete and re-insertion (see **Algorithm 2**). An example of downsample is shown in Fig. 2.2.

Table 2.1: Comparison of Supported Incremental Updates

		Static K-D Tree	Dynamic K-D Tree	Scapegoat K-D Tree	ikd-Tree
Point-wise	Insert	✗	✓	✓	✓
	Delete	✗	✗	✓	✓
	Re-insert	✗	✗	✗	✓
Box-wise	Insert	✗	✓	✓	✓
	Delete	✗	✗	✗	✓
	Re-insert	✗	✗	✗	✓
Downsample		✗	✗	✗	✓

In summary, Table 2.1 shows the comparison of supported incremental updates on the static k-d tree [92], the dynamic k-d tree [145], the scapegoat k-d tree [146] and our ikd-Tree.

2.3.4 Re-balancing

Our ikd-Tree actively monitors the balance property of the incremental k-d tree and dynamically re-balances it by partial re-building.

2.3.4.1 Balancing Criterion

The balancing criterion is composed of two sub-criterions: α-balanced criterion and α-deleted criterion. Suppose a sub-tree of the incremental k-d tree is rooted at T. The sub-tree is α-balanced if and only if it satisfies the following condition:

$$\begin{aligned} S(T.leftchild) &< \alpha_{bal}\Big(S(T)-1\Big) \\ S(T.rightchild) &< \alpha_{bal}\Big(S(T)-1\Big) \end{aligned} \tag{2.1}$$

where $\alpha_{bal} \in (0.5, 1)$ and $S(T)$ is the $treesize$ attribute of the node T.

The α-deleted criterion of the sub-tree rooted at T is

$$I(T) < \alpha_{del} S(T) \tag{2.2}$$

where $\alpha_{del} \in (0, 1)$ and $I(T)$ denotes the number of invalid nodes on the sub-tree (i.e., the attributes $invalidnum$ of node T).

If a sub-tree of the incremental k-d tree meets both criterion, the sub-tree is balanced. The entire tree is balanced if all sub-trees are balanced. Violation of either criterion will trigger a re-building process to re-balance that (sub-) tree: the α-balanced criterion maintains the maximum height of the (sub-) tree. It can be easily proved that the maximum height of an α-balanced tree is $\log_{1/\alpha_{bal}}(n)$ where n is the tree size; the α-deleted criterion ensures invalid nodes (i.e., labeled as "deleted") on the (sub-) trees are removed to reduce tree size. Reducing height and size of the k-d tree allows highly efficient incremental operations and queries in future. The function `ViolateCriterion` in **Algorithm 2, Line 16** returns true if either criterion is violated.

2.3.4.2 Re-build

Assuming re-building is triggered on a subtree $\mathcal{T}$ (see Fig. 2.3), the sub-tree is firstly flattened into a point storage array V. The tree nodes labeled as "deleted" are discarded during flattening. A new perfectly balanced k-d tree is then built with all points in V by **Algorithm 1**.

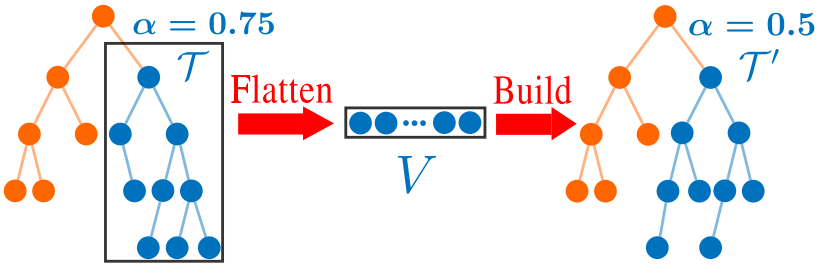


Figure 2.3: Re-build an unbalanced sub-tree

2.3.4.3 Parallel Re-build

An evident degradation of real-time ability is observed when re-building a large sub-tree on the incremental k-d tree. To preserve high real-time ability, we design a double-thread re-building method: the main thread only re-builds sub-trees whose size is smaller than a predetermined value $N_{\max}$ and the second thread re-builds the rest. The key problem is how to avoid information lose and memory conflicts between the main thread and the second thread.

The re-building algorithm on the second thread is shown in **Algorithm 4**. Denote the sub-tree to re-build in the second thread as $\mathcal{T}$ and its root node as T. The second thread will lock all incremental updates (i.e., points insert, re-insert, and delete) but not queries on this sub-tree (Line 2). Then the second thread copies all valid points contained in the sub-tree $\mathcal{T}$ into a point array V (i.e. flatten) while leaving the original sub-tree unchanged for possible queries during the re-building process (Line 3). After the flattening, the sub-tree is unlocked for the main thread to take further requests of incremental updates (Line 4). These requests will be recorded in a queue named as operation logger. Once the second thread completes building a new balanced k-d tree $\mathcal{T}'$ from the point array V (Line 5), the recorded update requests will be performed on the balanced sub-tree $\mathcal{T}'$ by function `IncrementalUpdates` (Line 6-8) where the parallel re-building option is set to false (as it is already in the second thread). After all pending requests are processed, the algorithm locks the node T from both incremental updates and queries and replace it with the new one T' (Line 9-12). Finally, the algorithm frees the memory of the original sub-tree (Line 13). Note that `LockUpdates` does not block queries, which can be conducted parallelly in the main thread. In contrast, `LockAll` blocks all access including queries, but it finishes very quickly (i.e., only one instruction), allowing timely queries in the main thread. The function `LockUpdates` and `LockAll` are

implemented by mutual exclusion (mutex).

Algorithm 4: Parallelly Rebuild for Re-balancing

```
Input: T                                   ▷ Root node of 𝒯 for re-building
Function ParallelRebuild(T)
    LockUpdates(T);
    V ← Flatten(T);
    Unlock(T);
    T' ← Build(V,0,size(V)-1);
    foreach op in OperationLogger do
        IncrementalUpdates(T', op, false)
    end
    T_temp ← T;
    LockAll(T);
    T ← T';
    Unlock(T);
    Free(T_temp);
End Function
```

2.3.5 K-Nearest Neighbor Search

The nearest search on the incremental k-d tree is an accurate nearest search [127] instead of an approximate one as [149]. The function `Pushdown` is applied before searching the sub-tree rooted at node T to pass down its lazy labels. We use the attribute *range* to speed up the search process thus hard real-time ability is preserved. Due to the space limit, the details of k-nearest search algorithm is not presented in this paper. Interested readers can refer to the related codes in our open source library.

2.4 Complexity Analysis

2.4.1 Time Complexity

The time complexity of ikd-Tree breaks into the time for incremental operations (insertion, re-insertion and delete) and re-building.

2.4.1.1 Incremental Operations

The time complexity of point-wise operations is given as

Lemma 1 (Point-wise Operations)**.** *An incremental k-d tree can handle a point-wise incremental operation with time complexity of* $O(\log n)$ *where* n *is the tree size.*

Proof. The maximum height of an incremental k-d tree can be easily proved to be $\log_{1/\alpha_{bal}}(n)$ from Eq. (2.1) while that of a static k-d tree is $\log_2 n$. Hence the lemma is directly obtained from [92] where the time complexity of point insertion and delete on a k-d tree was proved to be $O(\log n)$. The point-wise re-insertion modifies the attribute *deleted* on a tree node thus the time complexity is the same as point-wise delete. □

The time complexity of box-wise operations on an incremental k-d tree is:

Lemma 2 (Box-wise Operations)**.** *An incremental 3-d tree handles box-wise insertion of m points in C_D with time complexity of $O(m \log n)$. Furthermore, suppose points on the 3-d tree are in space $S_x \times S_y \times S_z$ and $C_D = L_x \times L_y \times L_z$. The box-wise delete and re-insertion can be handled with time complexity of $O(H(n))$, where*

$$O(H(n)) = \begin{cases} O(\log n) & \text{if} \Delta_{\min} \geqslant \alpha(\frac{2}{3}) (*) \\ O(n^{1-a-b-c}) & \text{if} \Delta_{\max} \leqslant 1 - \alpha(\frac{1}{3}) (**) \\ O(n^{\alpha(\frac{1}{3}) - \Delta_{\min} - \Delta_{med}}) & \text{if (*) and (**) fail and} \\ & \Delta_{med} < \alpha(\frac{1}{3}) - \alpha(\frac{2}{3}) \\ O(n^{\alpha(\frac{2}{3}) - \Delta_{\min}}) & \text{otherwise.} \end{cases} \tag{2.3}$$

where $a = \log_n \frac{S_x}{L_x}$, $b = \log_n \frac{S_y}{L_y}$ and $c = \log_n \frac{S_z}{L_z}$ with $a, b, c \geqslant 0$. $\Delta_{\min}$, Δ_{med} and $\Delta_{\max}$ are the minimal, median and maximal value among a, b and c. $\alpha(u)$ is the flajolet-puech function with $u \in [0, 1]$, where particular value is provided: $\alpha(\frac{1}{3}) = 0.7162$ and $\alpha(\frac{2}{3}) = 0.3949$

Proof. The box-wise insertion is implemented by point-wise insertion thus the time complexity can be directly obtained from Lemma 1. An asymptotic time complexity for range search on a k-d tree is provided in [151]. The box-wise delete and re-insertion can be regarded as a range search except that lazy labels are attached to the tree nodes. Therefore, the conclusion of range search can be applied to the box-wise delete and re-insertion on the incremental k-d tree. □

The down-sampling method on an incremental k-d tree is composed of box-wise search and delete followed by the point insertion. By applying Lemma 1 and Lemma 2, the time complexity of downsample is $O(\log n)+O(H(n))$. Generally, the downsample

hypercube C_D is very small comparing with the entire space. Therefore, the normalized range Δx, Δy and Δz are small and the value of $\Delta_{\min}$ satisfies the condition (*) for time complexity of $O(\log n)$. Hence, the time complexity of down-sampling is $O(\log n)$.

2.4.1.2 Re-build

Time complexity for re-building breaks into two types: single-thread re-building and parallel two-thread re-building. In the former case, the re-building is performed by the main thread in a recursive way. Each level takes the time of sorting (i.e., $O(n)$) and the total time over $\log n$ levels is $O(n \log n)$ [92] when the the dimension k is low (e.g., 3 in most robotic applications). For parallel re-building, the time consumed in the main thread is only flattening (which suspends the main thread from further incremental updates, **Algorithm 4**, Line 2-4) but not building (which is performed in parallel by the second thread, **Algorithm 4**, Line 5) or tree update (which takes constant time $O(1)$, **Algorithm 4**, Line 9-12), leading to a time complexity of $O(n)$. In summary, the time complexity of re-building an incremental k-d tree is $O(n)$ for two-thread parallel re-building and $O(n \log n)$ for single-thread re-building.

2.4.1.3 Nearest Search

For robotic applications, the points dimension is usually very small. Hence the time complexity of k-nearest search on the incremental k-d tree can be simply approximated as $O(\log n)$ because the maximum height of the incremental k-d tree is maintained no larger than $\log_{\frac{1}{\alpha}} n$.

2.4.2 Space Complexity

As shown Section 2.3, each node on the incremental k-d tree records point information, tree size, invalid point number and point distribution of the tree. Extra flags such as lazy labels are maintained on each node for box-wise operations. For an incremental k-d tree with n nodes, the space complexity is $O(n)$ though the space constant is a few times larger than a static k-d tree.

Table 2.2: The Parameters Setup of ikd-Tree in Experiments

	Randomized Data	LiDAR Inertial-Odometry and Mapping
$N_{\max}$	1500	1500
α_{bal}	0.6	0.6
α_{del}	0.5	0.5
C_D	-	0.2 m×0.2 m×0.2 m

2.5 Application Experiments

2.5.1 Randomized Data Experiments

The efficiency of our ikd-Tree is fully investigated by two experiments on randomized incremental data sets. The first experiment generates 5,000 points randomly in a 10 m×10 m×10 m space (i.e., the workspace) to initialize the incremental k-d tree. Then 1,000 test operations are conducted on the k-d tree. In each test operation, 200 new points randomly sampled in the workspace are inserted (point-wise) to the k-d tree. Then another 200 points are randomly sampled in the workspace and searched on (but not inserted to) the k-d tree for 5 nearest points of each. For every 50 test operations, 4 cubes are sampled in the workspace with side length of 1.5 m and points contained in these 4 cubes are deleted (box-wise) from the k-d tree. For every 100 test operations, 2,000 new points are sampled in the workspace and inserted (point-wise) to the k-d tree. We compare the ikd-Tree with the static k-d tree used in point cloud library [137] where at each test operation the k-d tree is entirely re-built. The experiments are performed on a PC with Intel i7-10700 CPU at 2.90 GHz and only 2 threads running. The parameters of the incremental k-d tree are summarized in Table 2.2 where no down-sampling is used to allow a fair comparison. Also the maximal point number allowed to store on the leaf node of a static k-d tree is set to 1 while the original setting in point cloud library is 15.

The results of the first experiment are shown in Fig. 2.4, where the point number increases from 5,000 to approximate 200,000. In this process, the time for incremental updates (including both incremental operations and re-building) on the ikd-Tree remains stably around 1.6 ms while that for the static k-d tree grows linearly with number of the points (see Fig. 2.4(a)). The high peaks in the time consumption are resulted from the large-scale point-wise insertion (and associated re-balancing) and the low peaks are resulted from box-wise delete (and associated re-balancing). As shown in Fig. 2.4(b),

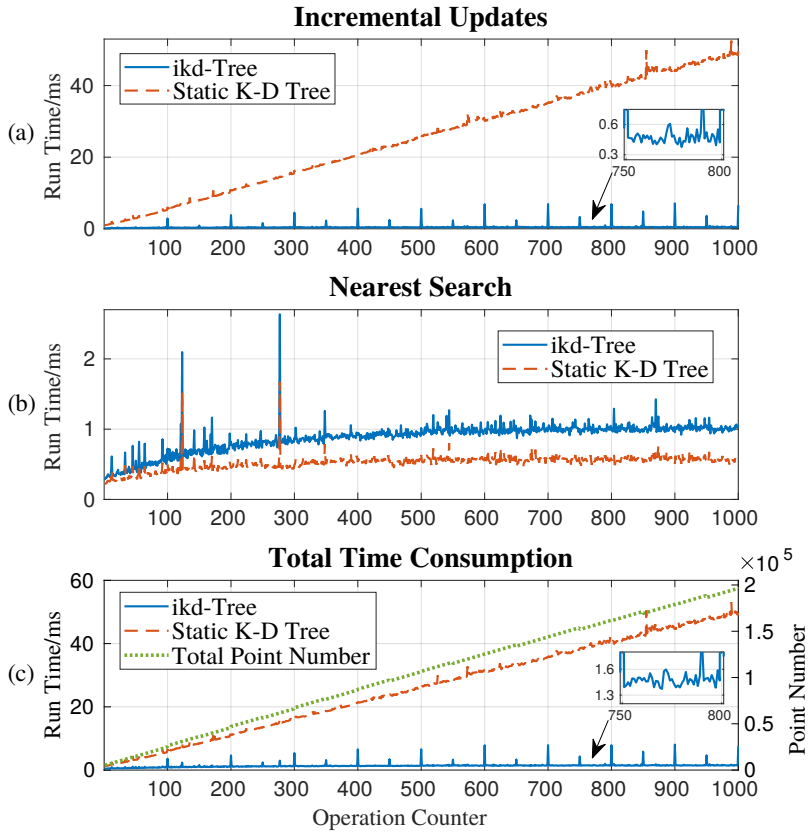


Figure 2.4: The time performance comparison between an ikd-Tree and a static k-d tree.

the time performance of the k-nearest search on the ikd-Tree is slightly slower than an static k-d tree, possibly due to the highly optimized implementation of the PCL library. Despite of the slightly lower efficiency in query, the overall time consumption of ikd-Tree outperforms the static k-d tree by one order of magnitude (See Fig. 2.4(c)).

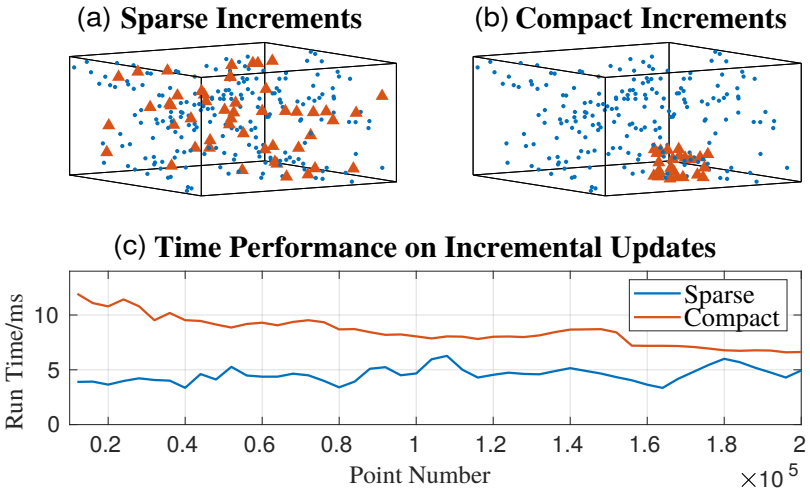


Figure 2.5: Fig. (a) and (b) illustrate new points (orange triangles) and points already on the k-d tree (blue dots). Fig. (c) shows the time for incremental updates of sparse and compact data on k-d trees of different size.

The second experiment investigates the time performance of incremental updates for new points of different distribution. In the experiment, we sample two sets of 4,000 new points in a 10 m×10 m×10 m space (i.e., the workspace): one is evenly distributed (i.e., sparse data, see Fig. 2.5 (a)) and the other concentrated in a 2.5 m×2.5 m×2.5 m space (i.e., compact data, see Fig. 2.5 (b)). The sparse and compact data are inserted to an existing incremental k-d tree of different size but all sampled in the workspace. Fig. 2.5(c) shows the running time of sparse and compact point-wise insertion on k-d trees of different size. As expected, the incremental updates for compact data are slower than the sparse one because re-building are more likely to be triggered when inserting a large amount of points into a small sub-tree of a k-d tree.

2.5.2 LiDAR Inertial-Odometry and Mapping

We test our developed ikd-Tree in an actual robotic application: lidar-inertial odometry (and mapping) presented in [33, 43, 50, 128–130]. In this application, k-d tree-based nearest point search is crucial to match a point in a new LiDAR scan to its correspondences in the map (or the previous scan). Since the map is dynamically growing by

matching and merging new scans, the k-d tree has to be re-built every time a new scan is merged. Existing methods [33, 43, 50, 128–130] commonly used static kd-tree from PCL and rebuilds the entire tree based on all points in the map (or a submap). This leads to a significant computation costs severely limiting the map update rate (e.g., from 1 Hz [33, 43, 50] to 10 Hz [130]).

In this experiment, we replace the static k-d tree (build, update, and query) by our ikd-Tree, which enables incremental update of the map by updating the new points only. We test ikd-Tree on the lidar-inertial mapping package FAST-LIO in [130][2]. The experiment is conducted on a real-world outdoor scene using a Livox Avia LiDAR[3] [152] with 70° FoV and a high frame rate of 100 Hz. All the algorithms are running on the DJI Manifold 2-C[4] with a 1.8 GHz quad-core Intel i7-8550U CPU and 8 GB RAM.

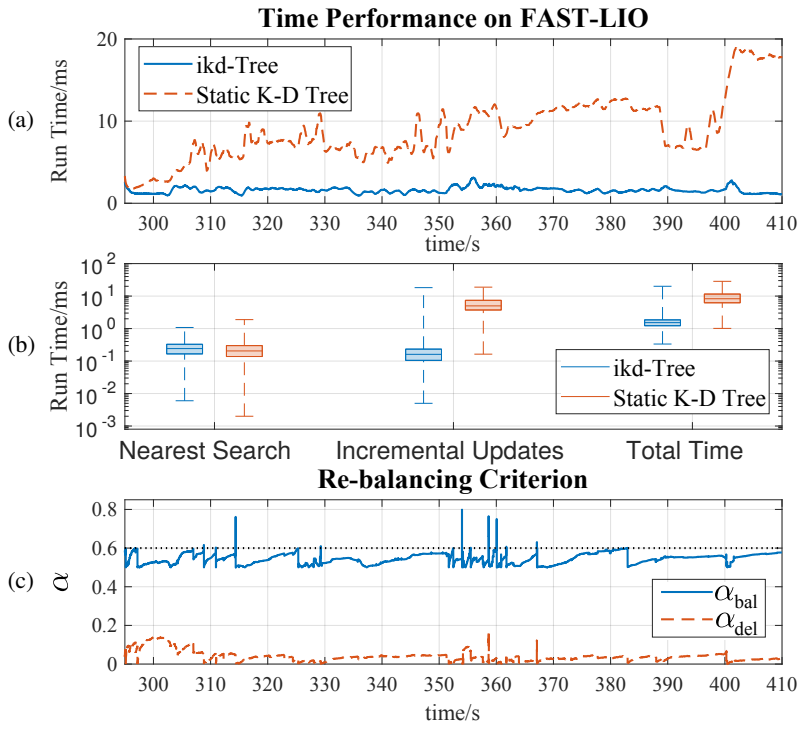


Figure 2.6: Fig. (a) shows the average running time of fusing one new lidar scan in FAST-LIO using the ikd-Tree and a static k-d tree. Fig. (b) shows time for nearest search, incremental updates, and total time in fusing one lidar scan. Fig. (c) shows the balance property after re-building on main thread.

[2] https://github.com/hku-mars/FAST_LIO
[3] https://www.livoxtech.com/de/avia
[4] https://www.dji.com/manifold-2/specs

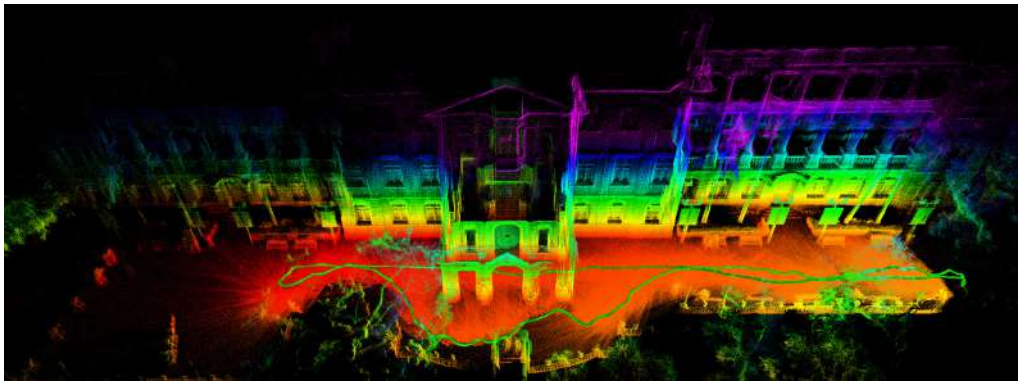


Figure 2.7: Mapping Result of the Main Building, University of Hong Kong. The green line is the path of the lidar computed by FAST-LIO.

The time of fusing a new lidar scan in FAST-LIO is shown in Fig. 2.6(a). The time is the averaged time of the most recent 100 scans. It is seen that the ikd-Tree achieves a nearly constant time performance around 1.6 ms, which fundamentally enables a mapping rate up to 100 Hz (against 10 Hz presented in the original work [130] using static k-d tree). On the other hand, the time with the static k-d tree is overall increasing linearly, and decreases occasionally due to the small overlap between the lidar current FoV and the map. The resultant processing time with the static k-d tree exceeds 10 ms from 366s on, which is more than the collection time of one lidar scan.

The time for fusing a new lidar scan consists of many operations, such as point registration, state estimation, and kd-tree related operations (including queries and update). The time breakdown of k-d tree-related operations is shown in Fig. 2.6(b). The average time of incremental updates for ikd-Tree is 0.23 ms which is only 4 % of that using an static k-d tree (5.71 ms). The average time of nearest search using the ikd-Tree and an static k-d tree are at the same level.

Furthermore, Fig. 2.6(c) investigates the balancing property of the incremental kd-tree by examining the two criterions α_{bal} and α_{del}, which are defined as:

$$\begin{aligned} \alpha_{bal}(T) &= \frac{\max\left\{S(T.leftchild), S(T.rightchild)\right\}}{S(T)-1} \\ \alpha_{del}(T) &= \frac{I(T)}{S(T)} \end{aligned} \tag{2.4}$$

As expected, the two criterion are maintained below the prescribed thresholds (see Table

2.2) due to re-building, indicating that the kd-tree is well-balanced through the incremental updates. The peaks over the thresholds is resulted from parallely re-building, which drop quickly when the re-building is done. Finally, Fig. 2.7 shows the 100 Hz mapping results.

2.6 Conclusion

This paper proposed an efficient data structure, ikd-Tree, to incrementally update a k-d tree in robotic applications. The ikd-Tree supports incremental operations in robotics while maintaining balanced by partial re-building. We provided a complete analysis of time and space complexity to prove the high efficiency of the proposed dynamic structure. The ikd-Tree was tested on randomized experiments and an outdoor LiDAR inertial-odometry and mapping experiment. In all tests, the proposed data structure achieved two orders of magnitude higher efficiency.

Chapter 3

FAST-LIO2: Fast Direct LiDAR-inertial Odometry

Building on the highly efficient data structure ikd-Tree in the preceding chapter, this chapter presents FAST-LIO2, a novel LiDAR-inertial Odometry framework allows fast, robust, and accurate LiDAR navigation (and mapping). FAST-LIO2 has two key novelties compared to existing works. The first one is directly registering raw points to the map (and subsequently update the map, i.e., mapping) without extracting features. This enables the exploitation of subtle features in the environment and hence increases the accuracy. The elimination of a hand-engineered feature extraction module also makes it naturally adaptable to emerging LiDARs of different scanning patterns; The second main novelty is maintaining a map by ikd-Tree that enables incremental updates (i.e., point insertion, delete) and dynamic re-balancing. Compared with existing dynamic data structures (octree, R*-tree, *nanoflann* k-d tree), ikd-Tree achieves superior overall performance while naturally supports downsampling on the tree. We conduct an exhaustive benchmark comparison in 19 sequences from a variety of open LiDAR datasets. FAST-LIO2 achieves consistently higher accuracy at a much lower computation load than other state-of-the-art LiDAR-inertial navigation systems. Various real-world experiments on solid-state LiDARs with small FoV are also conducted. Overall, FAST-LIO2 is computationally efficient (e.g., up to 100 Hz odometry and mapping in large outdoor environments), robust (e.g., reliable pose estimation in cluttered indoor environments with rotation up to 1000 °/s), versatile (i.e., applicable to both multi-line spinning and solid-state LiDARs, UAV and handheld platforms, and Intel

and ARM-based processors), while still achieving higher accuracy than existing methods. Our implementation of the system FAST-LIO2, and the data structure ikd-Tree is open-sourced on Github: https://github.com/hku-mars/FAST_LIO.

3.1 Introduction

Building a dense 3-dimension (3D) map of an unknown environment in real-time and simultaneously localizing in the map (i.e., SLAM) is crucial for autonomous robots to navigate in the unknown environment safely. The localization provides state feedback for the robot onboard controllers, while the dense 3D map provides necessary information about the environment (i.e., free space and obstacles) for trajectory planning. Vision-based SLAM [11, 153–155] is very accurate in localization but maintains only a sparse feature map and suffers from illumination variation and severe motion blur. On the other hand, real-time dense mapping [156–159] based on visual sensors at high resolution and accuracy with only the robot onboard computation resources is still a grand challenge.

Due to the ability to provide direct, dense, active, and accurate depth measurements of environments, 3D light detection and ranging (LiDAR) sensor has emerged as another essential sensor for robots [19, 20]. Over the last decade, LiDARs have been playing an increasingly important role in many autonomous robots, such as self-driving cars [160] and autonomous UAVs [135, 161]. Recent developments in LiDAR technologies have enabled the commercialization and mass production of more lightweight, cost-effective (in a cost range similar to global shutter cameras), and high performance (centimeter accuracy at hundreds of meters measuring range) solid-state LiDARs [21, 162], drawing much recent research interests [46, 50, 163–165]. The considerably reduced cost, size, weight, and power of these LiDARs hold the potential to benefit a broad scope of existing and emerging robotic applications.

The central requirement for adopting LiDAR-based SLAM approaches to these widespread applications is to obtain accurate, low-latency state estimation and dense 3D map with limited onboard computation resources. However, efficient and accurate LiDAR odometry and mapping are still challenging problems: 1) Current LiDAR sensors produce a large amount of 3D points from hundreds of thousands to millions per second. Processing such a large amount of data in real-time and on limited onboard computing

resources requires a high computation efficiency of the LiDAR odometry methods; 2) To reduce the computation load, features points, such as edge points or plane points, are usually extracted based on local smoothness. However, the performance of the feature extraction module is easily influenced by the environment. For example, in structure-less environments without large planes or long edges, the feature extraction will lead to few feature points. This situation is considerably worsened if the LiDAR Field of View (FoV) is small, a typical phenomenon of emerging solid-state LiDARs [50]. Furthermore, the feature extraction also varies from LiDAR to LiDAR, depending on the scanning pattern (e.g., spinning, prism-based [21], MEMS-based [162]) and point density. So the adoption of a LiDAR odometry method usually requires much hand-engineering work; 3) LiDAR points are usually sampled sequentially while the sensor undergoes continuous motion. This procedure creates significant motion distortion influencing the performance of the odometry and mapping, especially when the motion is severe. Inertial measurement units (IMUs) could mitigate this problem but introduces additional states (e.g., bias, extrinsic) to estimate; 4) LiDAR usually has a long measuring range (e.g., hundreds of meters) but with quite low resolution between scanning lines in a scan. The resultant point cloud measurements are sparsely distributed in a large 3D space, necessitating a large and dense map to register these sparse points. Moreover, the map needs to support efficient inquiry for correspondence search while being updated in real-time incorporating new measurements. Maintaining such a map is a very challenging task and very different from visual measurements, where an image measurement is of high resolution, so requiring only a sparse feature map because a feature point in the map can always find correspondence as long as it falls in the FoV.

In this work, we address these issues by two key novel techniques: incremental k-d tree and direct points registration. More specifically, our contributions are as follows: 1) We leverage an incremental k-d tree data structure, ikd-Tree, to represent a large dense point cloud map efficiently. Besides efficient nearest neighbor search, ikd-Tree supports incremental map update (i.e., point insertion, on-tree downsampling, points delete) and dynamic re-balancing at minimal computation cost. These features make the ikd-Tree very suitable for LiDAR odometry and mapping application, leading to 100 Hz odometry and mapping on computationally-constrained platforms such as an Intel i7-based micro-UAV onboard computer and even ARM-based processors. 2) Allowed by the increased

computation efficiency of ikd-Tree, we directly register raw points to the map, which enables more accurate and reliable scan registration even with aggressive motion and in very cluttered environments. We term this raw points-based registration as *direct method* in analogy to visual SLAM [166]. The elimination of a hand-engineered feature extraction makes the system naturally applicable to different LiDAR sensors; 3) We integrate these two key techniques into a full tightly-coupled lidar-inertial odometry system FAST-LIO [130] we recently developed. The system uses an IMU to compensate each point's motion via a rigorous back-propagation step and estimates the system's full state via an on-manifold iterated Kalman filter. The new system is termed as FAST-LIO2 and is open-sourced at GitHub to benefit the community; 4) We conduct various experiments to evaluate the effectiveness of the developed ikd-Tree, the direct point registration, and the overall system. Experiments on 18 sequences of various sizes show that ikd-Tree achieves superior performance against existing dynamic data structures (octree, R*-tree, *nanoflann* k-d tree) in the application of LiDAR odometry and mapping. Exhaustive benchmark comparison on 19 sequences from various open LiDAR datasets shows that FAST-LIO2 achieves consistently higher accuracy at a much lower computation load than other state-of-the-art LiDAR-inertial navigation systems. We finally show the effectiveness of FAST-LIO2 on challenging real-world data collected by emerging solid-state LiDARs with very small FoV, including aggressive motion (e.g., rotation speed up to 1000 °/s) and structure-less environments.

The remaining paper is organized as follows: In Section 3.2, we discuss relevant research works. We give an overview of the complete system pipeline and the details of each key components in Section 3.3, 3.4 and 3.5, respectively. The benchmark comparison on open datasets are presented in Section 3.6 and the real-world experiments are reported in Section 3.7, followed by conclusions in Section 3.9.

3.2 Related Works

3.2.1 LiDAR(-Inertial) Odometry

Existing works on 3D LiDAR SLAM typically inherit the LOAM structure proposed in [33]. It consists of three main modules: feature extraction, *odometry*, and *mapping*. In order to reduce the computation load, a new LiDAR scan first goes through feature points

(i.e., edge and plane) extraction based on the local smoothness. Then the *odometry* module (scan-to-scan) matches feature points from two consecutive scans to obtain a rough yet real-time (e.g., 10 Hz) LiDAR pose odometry. With the odometry, multiple scans are combined into a sweep which is then registered and merged to a global map (i.e., *mapping*). In this process, the map points are used to build a k-d tree which enables a very efficient k-nearest neighbor search (kNN search). Then, the point cloud registration is achieved by the Iterative Closest Point (ICP) [129, 167, 168] method. In order to lower the time for k-d tree building, the map points are downsampled at a prescribed resolution. The optimized mapping process is typically performed at a much low rate (1 Hz-2 Hz).

Subsequent LiDAR odometry works keep a framework similar to LOAM. For example, Lego-LOAM [43] introduces a ground point segmentation to lower the computation load and a loop closure module to reduce the long-term drift. Furthermore, LOAM-Livox [50] adopts the LOAM to an emerging solid-state LiDAR. In order to deal with the small FoV and non-repetitive scanning, where the features points from two consecutive scans have very few correspondences, the *odometry* of LOAM-Livox is obtained by directly registering a new scan to the global map. Such a direct scan to map registration improves odometry accuracy at the cost of increased computation load for building a k-d tree of the updated map points at every step.

Incorporating an IMU can considerably increase the accuracy and robustness of LiDAR odometry by compensating for the motion distortion in a LiDAR scan and providing a good initial pose required by ICP. More tightly-coupled LiDAR-inertial fusion works [38, 40, 163, 169] perform *odometry* in a small size local map consisting of a fixed number of recent LiDAR scans (or keyframes). Compared to scan-to-scan registration, the scan to local map registration is usually more accurate by using more recent information. More specifically, LIOM [40] presents a tightly-coupled LiDAR inertial fusion method where the IMU preintegrations are introduced into the *odometry*. LILI-OM [163] develops a new feature extraction method for non-repetitive scanning LiDARs and performs scan registration in a small map consisting of 20 recent LiDAR scans for the *odometry*. The *odometry* of LIO-SAM [169] requires a 9-axis IMU to produce attitude measurement as the prior of scan registration within a small local map. LINS [38] introduces a tightly-coupled iterated Kalman filter and robocentric formula

into the LiDAR pose optimization in the *odometry*. Since the local map in the above works is usually small to obtain real-time performance, the odometry drifts quickly, necessitating a low-rate *mapping* process, such as map refining (LINS [38]), sliding window joint optimization (LILI-OM [163] and LIOM [40]) and factor graph smoothing [170] (LIO-SAM [169]). Compared to the above methods, FAST-LIO [130] introduces a formal back-propagation that precisely considers the sampling time of LiDAR points and compensates the motion distortion via a rigorous kinematic model driven by IMU measurements. Furthermore, a new Kalman gain formula is used to reduce the computation complexity from the dimension of the measurements to the dimension of the state. The considerably increased computation efficiency allows a direct and real-time scan to map registration in *odometry* and update the map (i.e., *mapping*) at every step. However, to prevent the growing time of building a k-d tree of the updated map, the system can only work in small environments (e.g., hundreds of meters).

FAST-LIO2 builds on FAST-LIO [130] hence inheriting the tightly-coupled fusion framework, especially the back-propagation resolving motion distortion and fast Kalman gain computation boosting the efficiency. To systematically address the growing computation issue, we leverage a new data structure ikd-Tree which supports incremental map update at every step and efficient kNN inquiries. Benefiting from the drastically decreased computation load, the *odometry* is performed by directly registering raw LiDAR points to the map, such that it improves accuracy and robustness of odometry and mapping, especially when a new scan contains no prominent features (e.g., due to small FoV and/or structure-less environments). Compared to the above tightly-coupled LiDAR-inertial methods, which all use feature points, our method is more lightweight and achieves increased mapping rate and odometry accuracy, and eliminates the need for parameter tuning for feature extraction.

The idea of directly registering raw points in our work has been explored in LION [171], which is however a loosely-coupled method as reviewed above. This idea is also very similar to the generalized-ICP (G-ICP) proposed in [129], where a point is registered to a small local plane in the map. This ultimately assumes that the environment is smooth and hence can be viewed as a plane locally. However, the computation load of generalized-ICP is usually large [172]. Other works based on Normal Distribution Transformation (NDT) [79, 81, 84] also register raw points, but NDT has lower stability

compared to ICP and may diverge in some scenes [84].

3.2.2 Dynamic Data Structure in Mapping

In order to achieve real-time mapping, a dynamic data structure is required to support both incremental updates and kNN search with high efficiency. Generally, the kNN search problem can be solved by building spatial indices for data points, which can be divided into two categories: partitioning the data and splitting the space. A well-known instance to partition the data is R-tree [125] which clusters the data into potential overlapped axis-aligned cuboids based on data proximity in space. Various R-trees splits the nodes by linear, quadratic, and exponential complexities, all supporting nearest neighbor search and point-wise updating (insertion, delete, and re-insertion). Furthermore, R-trees also support searching target data points in a given search area or satisfying a given condition. Another version of R-trees is R*-tree which outperforms the original ones [173]. The R*-tree handles insertion by minimum overlap criteria and applies a forced re-insertion principle for the node splitting algorithm.

Octree [174] and k-dimensional tree (k-d tree) [92] are two well-known types of data structures to split the space for kNN search. The octree organizes 3-D point clouds by splitting the space equally into eight axis-aligned cubes recursively. The subdivision of a cube stops when the cube is empty, or a stopping rule (e.g., minimal resolution or minimal point number) is met. New points are inserted to leaf nodes on the octree while a further subdivision is applied if necessary. The octree supports both kNN search and box-wise search, which returns data points in a given axis-aligned cuboid.

The k-d tree is a binary tree whose nodes represent an axis-aligned hyperplane to split the space into two parts. In the standard construction rule, the splitting node is chosen as the median point along the longest dimension to achieve a compact space division [175]. When considering the data characteristics of low dimensionality and storage on main memory in mapping, comparative studies show that k-d trees achieve the best performance in kNN problem [94, 176]. However, inserting new points to and deleting old points from a k-d tree deteriorates the tree's balance property; thus, re-building is required to re-balance the tree. Mapping methods using k-d tree libraries, such as ANN [177], *libnabo* [176] and FLANN [149], fully re-build the k-d trees to update

the map, which results in considerable computation. Though hardware-based methods to re-build k-d trees have been thoroughly investigated in 3D graphic applications [141–144], the proposed methods rely heavily on the computational sources which are usually limited on onboard computers for robotic applications. Instead of re-building the tree in full scale, Galperin *et al.* proposed a scapegoat k-d tree where re-building is applied partially on the unbalanced sub-trees to maintain a loose balance property of the entire tree [146]. Another approach to enable incremental operations is maintaining a set of k-d trees in a logarithmic method similar to [145, 178] and re-building a carefully chosen sub-set. The Bkd-tree maintains a k-d tree $\mathcal{T}_0$ with maximal size M in the main memory and a set of k-d trees $\mathcal{T}_i$ on the external memory where the i-th tree has a size of $2^{(i-1)}M$ [147]. When the tree $\mathcal{T}_0$ is full, the points are extracted from $\mathcal{T}_0$ to $\mathcal{T}_{k-1}$ and inserted into the first empty tree $\mathcal{T}_k$. The state-of-the-art implementation *nanoflann* k-d tree leverages the logarithmic structure for incremental updates, whereas lazy labels only mark the deleted points without removing them from the trees (hence memory) [179].

We leverage the new data structure, ikd-Tree, to achieve real-time mapping. The ikd-Tree supports point-wise insertion with on-tree downsampling which is a common requirement in mapping, whereas downsampling must be done outside before inserting new points into other dynamic data structures [173, 174, 179]. When it is required to remove unnecessary points in a given area with regular shapes (e.g., cuboids), the existing implementations of R-trees and octrees search the points within the given space and delete them one by one while common k-d trees use a radius search to obtain point indices. Compared to such an indirect and inefficient method, the ikd-Tree deletes the points in given axis-aligned cuboids directly by maintaining range information and lazy labels. Points labeled as "deleted" are removed during the re-building process. Furthermore, though incremental updates are available after applying the partial re-balancing methods as the scapegoat k-d tree [146] and *nanoflann* k-d tree [179], the mapping methods using k-d trees suffers from intermittent delay when re-building on a large number of points. In order to overcome this, the significant delay in ikd-Tree is avoided by parallel re-building while the real-time ability and accuracy in the main thread are guaranteed.

3.3 System Overview

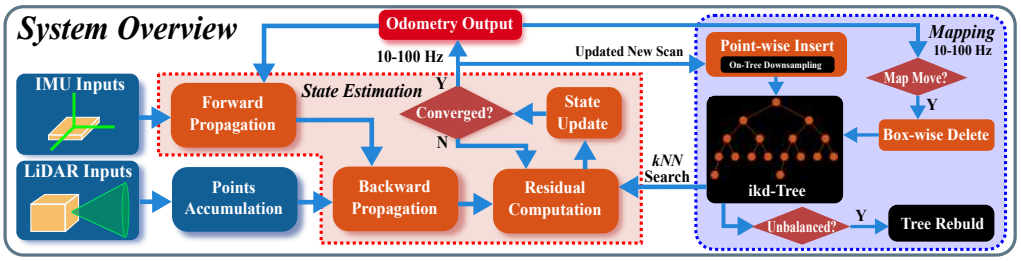


Figure 3.1: System overview of FAST-LIO2. The overall system consists of a state estimation module, which estimates the full LiDAR state by registering raw points in a scan to the map points via a tightly-coupled iterated Kalman filter, and a mapping module, which incrementally adds the new points in each scan to a k-d tree structure (i.e., ikd-Tree) and re-balances the tree when necessary.

The pipeline of FAST-LIO2 is shown in Fig. 3.1. The sequentially sampled LiDAR raw points are first accumulated over a period between 10 m sec (for 100 Hz update) and 100 m sec (for 10 Hz update). The accumulated point cloud is called a scan. In order to perform state estimation, points in a new scan are registered to map points (i.e., *odometry*) maintained in a large local map via a tightly-coupled iterated Kalman filter framework (big dashed block in red, see Section 3.4). Global map points in the large local map are organized by an incremental k-d tree structure ikd-Tree (big dashed block in blue, see Section 3.5). If the FoV range of current LiDAR crosses the map border, the historical points in the furthest map area to the LiDAR pose will be deleted from ikd-Tree. As a result, the ikd-Tree tracks all map points in a large cube area with a certain length (referred to as "map size" in this paper) and is used to compute the residual in the state estimation module. The optimized pose finally registers points in the new scan to the global frame and merges them into the map by inserting to the ikd-Tree at the rate of odometry (i.e., *mapping*).

3.4 State Estimation

The state estimation of FAST-LIO2 is a tightly-coupled iterated Kalman filter inherited from FAST-LIO [130] but further incorporates the online calibration of LiDAR-IMU extrinsic parameters. Here we briefly explain the essential formulations and workflow of the filter and refer readers to [130] for more details. To ease the explanation, we use the notations summarized in Table I. Moreover, we encapsulate two operations, $\boxplus$

("boxplus") and its inverse $\boxminus$ ("boxminus") from [130, 180], to parameterize the state error on a manifold $\mathcal{M}$ with dimension n:

$$\begin{aligned} & \boxplus : \mathcal{M} \times \mathbb{R}^n \to \mathcal{M}; \quad \boxminus : \mathcal{M} \times \mathcal{M} \to \mathbb{R}^n \\ SO(3): \quad & \mathbf{R} \boxplus \mathbf{r} = \mathbf{R}\mathrm{Exp}(\mathbf{r}); \quad \mathbf{R}_1 \boxminus \mathbf{R}_2 = \mathrm{Log}(\mathbf{R}_2^{\mathrm{T}}\mathbf{R}_1) \\ \mathbb{R}^n: \quad & \mathbf{a} \boxplus \mathbf{b} = \mathbf{a} + \mathbf{b}; \qquad \mathbf{a} \boxminus \mathbf{b} = \mathbf{a} - \mathbf{b} \end{aligned} \tag{3.1}$$

where $\mathrm{Exp}(\mathbf{r}) = \mathbf{I} + \frac{\mathbf{r}}{\|\mathbf{r}\|}\sin(\|\mathbf{r}\|) + \frac{\mathbf{r}^2}{\|\mathbf{r}\|^2}(1 - \cos(\|\mathbf{r}\|))$ is the exponential map on $SO(3)$ and $\mathrm{Log}(\cdot)$ is its inverse map. For a compound manifold $\mathcal{M} = SO(3) \times \mathbb{R}^n$ that is the Cartesian product between its sub-manifold components, we have:

$$\begin{bmatrix} \mathbf{R} \\ \mathbf{a} \end{bmatrix} \boxplus \begin{bmatrix} \mathbf{r} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{R} \boxplus \mathbf{r} \\ \mathbf{a} + \mathbf{b} \end{bmatrix}; \quad \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{a} \end{bmatrix} \boxminus \begin{bmatrix} \mathbf{R}_2 \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_1 \boxminus \mathbf{R}_2 \\ \mathbf{a} - \mathbf{b} \end{bmatrix} \tag{3.2}$$

Table 3.1: Notations

Symbols	Meaning
L	The LiDAR body frame at the LiDAR scan-end time.
$\mathbf{x}_i$	The state $\mathbf{x}$ at the i-th IMU sample time;
$\mathbf{x}_k$	The state $\mathbf{x}$ at the k-th LiDAR scan-end time.
$\mathbf{x}, \widehat{\mathbf{x}}, \bar{\mathbf{x}}$	The ground-true, propagated, and updated value of state $\mathbf{x}$.
$\widetilde{\mathbf{x}}$	The error between ground-true state $\mathbf{x}$ and its estimation $\hat{\mathbf{x}}$.
$\widehat{\mathbf{x}}^{\kappa}$	The estimate of the state $\mathbf{x}$ in the κ-th iteration of the iterated Kalman filter.

3.4.1 Kinematic Model

We first derive the system model, which consists of a state transition model and a measurement model.

3.4.1.1 State Transition Model

Take the first IMU frame (denoted as I) as the global frame (denoted as G) and denote ${}^{I}\mathbf{T}_L = \left({}^{I}\mathbf{R}_L,\ {}^{I}\mathbf{p}_L\right)$ the unknown extrinsic between LiDAR and IMU, the kinematic

model is:

$$
\begin{aligned}
{}^{G}\dot{\mathbf{R}}_I &= {}^{G}\mathbf{R}_I \lfloor \boldsymbol{\omega}_m - \mathbf{b}_{\boldsymbol{\omega}} - \mathbf{n}_{\boldsymbol{\omega}} \rfloor_{\wedge},\ {}^{G}\dot{\mathbf{p}}_I = {}^{G}\mathbf{v}_I, \\
{}^{G}\dot{\mathbf{v}}_I &= {}^{G}\mathbf{R}_I \left(\mathbf{a}_m - \mathbf{b}_{\mathbf{a}} - \mathbf{n}_{\mathbf{a}}\right) + {}^{G}\mathbf{g} \\
\dot{\mathbf{b}}_{\boldsymbol{\omega}} &= \mathbf{n}_{\mathbf{b}\boldsymbol{\omega}},\ \dot{\mathbf{b}}_{\mathbf{a}} = \mathbf{n}_{\mathbf{ba}}, \\
{}^{G}\dot{\mathbf{g}} &= \mathbf{0},\ {}^{I}\dot{\mathbf{R}}_L = \mathbf{0},\ {}^{I}\dot{\mathbf{p}}_L = \mathbf{0}
\end{aligned}
\tag{3.3}
$$

where ${}^{G}\mathbf{p}_I$, ${}^{G}\mathbf{R}_I$ denote the IMU position and attitude in the global frame, ${}^{G}\mathbf{v}_I$ is the IMU velocity in the global frame, ${}^{G}\mathbf{g}$ is the gravity vector in the global frame, $\mathbf{a}_m$ and $\boldsymbol{\omega}_m$ are IMU measurements, $\mathbf{n}_{\mathbf{a}}$ and $\mathbf{n}_{\boldsymbol{\omega}}$ denote the measurement noise of $\mathbf{a}_m$ and $\boldsymbol{\omega}_m$, $\mathbf{b}_{\mathbf{a}}$ and $\mathbf{b}_{\boldsymbol{\omega}}$ are the IMU biases modeled as random walk process driven by $\mathbf{n}_{\mathbf{ba}}$ and $\mathbf{n}_{\mathbf{b}\boldsymbol{\omega}}$, and the notation $\lfloor \mathbf{a} \rfloor_{\wedge}$ denotes the skew-symmetric cross product matrix of vector $\mathbf{a} \in \mathbb{R}^3$.

Denote i the index of IMU measurements, the continuous kinematic model (3.3) can be discretized at the IMU sampling period Δt [181]:

$$
\mathbf{x}_{i+1} = \mathbf{x}_i \boxplus \left(\Delta t \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i, \mathbf{w}_i)\right) \tag{3.4}
$$

where the function $\mathbf{f}$, state $\mathbf{x}$, input $\mathbf{u}$ and noise $\mathbf{w}$ are defined as below:

$$
\begin{aligned}
\mathbf{x} &\triangleq \begin{bmatrix} {}^{G}\mathbf{R}_I^T & {}^{G}\mathbf{p}_I^T & {}^{G}\mathbf{v}_I^T & \mathbf{b}_{\boldsymbol{\omega}}^T & \mathbf{b}_{\mathbf{a}}^T & {}^{G}\mathbf{g}^T & {}^{I}\mathbf{R}_L^T & {}^{I}\mathbf{p}_L^T \end{bmatrix}^T \in \mathcal{M} \\
\mathbf{u} &\triangleq \begin{bmatrix} \boldsymbol{\omega}_m^T & \mathbf{a}_m^T \end{bmatrix}^T,\ \mathbf{w} \triangleq \begin{bmatrix} \mathbf{n}_{\boldsymbol{\omega}}^T & \mathbf{n}_{\mathbf{a}}^T & \mathbf{n}_{\mathbf{b}\boldsymbol{\omega}}^T & \mathbf{n}_{\mathbf{ba}}^T \end{bmatrix}^T \\
\mathbf{f}\left(\mathbf{x}, \mathbf{u}, \mathbf{w}\right) &= \begin{bmatrix} \boldsymbol{\omega}_m - \mathbf{b}_{\boldsymbol{\omega}} - \mathbf{n}_{\boldsymbol{\omega}} \\ {}^{G}\mathbf{v}_I + \frac{1}{2}\left({}^{G}\mathbf{R}_I \left(\mathbf{a}_m - \mathbf{b}_{\mathbf{a}} - \mathbf{n}_{\mathbf{a}}\right) + {}^{G}\mathbf{g}\right)\Delta t \\ {}^{G}\mathbf{R}_I \left(\mathbf{a}_m - \mathbf{b}_{\mathbf{a}} - \mathbf{n}_{\mathbf{a}}\right) + {}^{G}\mathbf{g} \\ \mathbf{n}_{\mathbf{b}\boldsymbol{\omega}} \\ \mathbf{n}_{\mathbf{ba}} \\ \mathbf{0}_{3\times 1} \\ \mathbf{0}_{3\times 1} \\ \mathbf{0}_{3\times 1} \end{bmatrix},
\end{aligned}
\tag{3.5}
$$

and the operation $\boxplus$ is defined on the state manifold $\mathcal{M}$ below (see (3.2))

$$
\mathcal{M} \triangleq SO(3) \times \mathbb{R}^{15} \times SO(3) \times \mathbb{R}^3;\ \dim(\mathcal{M}) = 24 \tag{3.6}
$$

which is the Cartesian products of each components. Notice that when compared to FAST-LIO, we further included the extrinsic parameters ${}^{I}\mathbf{T}_{L} = \left({}^{I}\mathbf{R}_{L},\ {}^{I}\mathbf{p}_{L}\right)$ into the state $\mathbf{x}$ in (3.4), enabling the extrinsic to be estimated online along with other states detailed below.

3.4.1.2 Measurement Model

LiDAR typically samples points one after another. The resultant points are therefore sampled at different poses when the LiDAR undergoes continuous motion. To correct this in-scan motion, we employ the back-propagation proposed in [130], which estimates the LiDAR pose of each point in the scan with respect to the pose at the scan end time based on IMU measurements. The estimated relative pose enables us to project all points to the scan end-time based on the exact sampling time of each individual point in the scan. As a result, points in the scan can be viewed as all sampled simultaneously at the scan end-time.

Denote k the index of LiDAR scans and $\{{}^{L}\mathbf{p}_{j}, j = 1, \cdots, m\}$ the points in the k-th scan which are sampled at the local LiDAR coordinate frame L at the scan end-time. Due to the LiDAR measurement noise, each measured point ${}^{L}\mathbf{p}_{j}$ is typically contaminated by a noise ${}^{L}\mathbf{n}_{j}$ consisting of the ranging and beam-directing noise. Removing this noise leads to the true point location in the local LiDAR coordinate frame ${}^{L}\mathbf{p}_{j}^{\mathrm{gt}}$:

$$ {}^{L}\mathbf{p}_{j}^{\mathrm{gt}} = {}^{L}\mathbf{p}_{j} + {}^{L}\mathbf{n}_{j}. \tag{3.7} $$

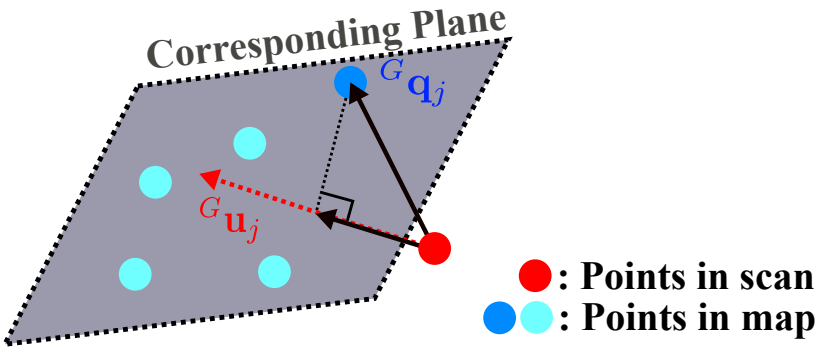


Figure 3.2: The measurement model: a LiDAR point is assumed to lie on a small plane formed by its nearby map points. The ${}^{G}\mathbf{u}_{j}$ is the normal vector of the plane and ${}^{G}\mathbf{q}_{j}$ is a point lying on the plane.

This true point, after projecting to the global frame using the corresponding LiDAR pose ${}^{G}\mathbf{T}_{I_k} = \left({}^{G}\mathbf{R}_{I_k},\ {}^{G}\mathbf{p}_{I_k}\right)$ and extrinsic ${}^{I}\mathbf{T}_{L}$, should lie exactly on a local small plane patch in the map, i.e., the measurement model is:

$$\mathbf{0} = {}^{G}\mathbf{u}_j^T \left({}^{G}\mathbf{T}_{I_k}\,{}^{I}\mathbf{T}_L \left({}^{L}\mathbf{p}_j + {}^{L}\mathbf{n}_j\right) - {}^{G}\mathbf{q}_j\right) \tag{3.8}$$

where ${}^{G}\mathbf{u}_j$ is the normal vector of the corresponding plane and ${}^{G}\mathbf{q}_j$ is a point lying on the plane (see Fig. 3.2). It should be noted that the ${}^{G}\mathbf{T}_{I_k}$ and ${}^{I}\mathbf{T}_{L}$ are all contained in the state vector $\mathbf{x}_k$. The measurement contributed by the j-th point measurement ${}^{L}\mathbf{p}_j$ can therefore be summarized from (3.8) to a more compact form as below:

$$\mathbf{0} = \mathbf{h}_j\left(\mathbf{x}_k, {}^{L}\mathbf{n}_j\right) \triangleq {}^{G}\mathbf{u}_j^T \left({}^{G}\mathbf{T}_{I_k}\,{}^{I}\mathbf{T}_L({}^{L}\mathbf{p}_j + {}^{L}\mathbf{n}_j) - {}^{G}\mathbf{q}_j\right), \tag{3.9}$$

which defines an implicit measurement model for the state vector $\mathbf{x}_k$.

3.4.2 Iterated Kalman Filter

Based on the state model (3.4) and measurement model (3.9) formulated on manifold $\mathcal{M}$, we employ an iterated Kalman filter directly operating on the manifold $\mathcal{M}$ following the procedures in [130] and [181]. It consists of two key steps: propagation upon each IMU measurement and iterated update upon each LiDAR scan, both step estimates the state naturally on the manifold $\mathcal{M}$ thus avoiding any re-normalization. Since the IMU measurements are typically at a higher frequency than a LiDAR scan (e.g., 200 Hz for IMU measurement and 10 Hz∼100 Hz for LiDAR scans), multiple propagation steps are usually performed before an update.

3.4.2.1 Propagation

Assume the optimal state estimate after fusing the last (i.e., $k-1$-th) LiDAR scan is $\bar{\mathbf{x}}_{k-1}$ with covariance matrix $\bar{\mathbf{P}}_{k-1}$. The forward propagation is performed upon the arrival of an IMU measurement. More specifically, the state and covariance are

propagated following (3.4) by setting the process noise $\mathbf{w}_i$ to zero:

$$\begin{aligned}\widehat{\mathbf{x}}_{i+1} &= \widehat{\mathbf{x}}_i \boxplus (\Delta t \mathbf{f}(\widehat{\mathbf{x}}_i, \mathbf{u}_i, \mathbf{0}))\,;\ \widehat{\mathbf{x}}_0 = \bar{\mathbf{x}}_{k-1}, \\ \widehat{\mathbf{P}}_{i+1} &= \mathbf{F}_{\widetilde{\mathbf{x}}_i} \widehat{\mathbf{P}}_i \mathbf{F}_{\widetilde{\mathbf{x}}_i}^T + \mathbf{F}_{\mathbf{w}_i} \mathbf{Q}_i \mathbf{F}_{\mathbf{w}_i}^T\,;\ \widehat{\mathbf{P}}_0 = \bar{\mathbf{P}}_{k-1},\end{aligned} \tag{3.10}$$

where $\mathbf{Q}_i$ is the covariance of the noise $\mathbf{w}_i$ and the matrix $\mathbf{F}_{\widetilde{\mathbf{x}}_i}$ and $\mathbf{F}_{\mathbf{w}_i}$ are computed as below (see more abstract derivation in [181] and more concrete derivation in [130]):

$$\begin{aligned}\mathbf{F}_{\widetilde{\mathbf{x}}_i} &= \frac{\partial (\mathbf{x}_{i+1} \boxminus \widehat{\mathbf{x}}_{i+1})}{\partial \widetilde{\mathbf{x}}_i} |_{\widetilde{\mathbf{x}}_i = \mathbf{0},\ \mathbf{w}_i = \mathbf{0}} \\ \mathbf{F}_{\mathbf{w}_i} &= \frac{\partial (\mathbf{x}_{i+1} \boxminus \widehat{\mathbf{x}}_{i+1})}{\partial \mathbf{w}_i} \Big|_{\widetilde{\mathbf{x}}_i = \mathbf{0},\ \mathbf{w}_i = \mathbf{0}}\end{aligned} \tag{3.11}$$

The forward propagation continues until reaching the end time of a new (i.e., k-th) scan where the propagated state and covariance are denoted as $\widehat{\mathbf{x}}_k, \widehat{\mathbf{P}}_k$.

3.4.2.2 Residual Computation

Assume the estimate of state $\mathbf{x}_k$ at the current iterated update (see Section 3.4.2.3) is $\widehat{\mathbf{x}}_k^\kappa$, when $\kappa = 0$ (i.e., before the first iteration), $\widehat{\mathbf{x}}_k^\kappa = \widehat{\mathbf{x}}_k$, the predicted state from the propagation in (3.10). Then, we project each measured LiDAR point ${}^L\mathbf{p}_j$ to the global frame ${}^G\widehat{\mathbf{p}}_j = {}^G\widehat{\mathbf{T}}_{I_k}^\kappa {}^I\widehat{\mathbf{T}}_{L_k}^\kappa {}^L\mathbf{p}_j$ and search its nearest 5 points in the map represented by ikd-Tree (see Section 3.5.1). The found nearest neighbouring points are then used to fit a local small plane patch with normal vector ${}^G\mathbf{u}_j$ and centroid ${}^G\mathbf{q}_j$ that were used in the measurement model (see (3.8) and (3.9)). Moreover, approximating the measurement equation (3.9) by its first order approximation made at $\widehat{\mathbf{x}}_k^\kappa$ leads to

$$\begin{aligned}\mathbf{0} &= \mathbf{h}_j \left(\mathbf{x}_k, {}^L\mathbf{n}_j\right) \simeq \mathbf{h}_j \left(\widehat{\mathbf{x}}_k^\kappa, \mathbf{0}\right) + \mathbf{H}_j^\kappa \widetilde{\mathbf{x}}_k^\kappa + \mathbf{r}_j \\ &= \mathbf{z}_j^\kappa + \mathbf{H}_j^\kappa \widetilde{\mathbf{x}}_k^\kappa + \mathbf{r}_j\end{aligned} \tag{3.12}$$

where $\widetilde{\mathbf{x}}_k^\kappa = \mathbf{x}_k \boxminus \widehat{\mathbf{x}}_k^\kappa$ (or equivalently $\mathbf{x}_k = \widehat{\mathbf{x}}_k^\kappa \boxplus \widetilde{\mathbf{x}}_k^\kappa$), $\mathbf{H}_j^\kappa$ is the Jacobin matrix of the measurement model $\mathbf{h}_j \left(\mathbf{x}_k, {}^L\mathbf{n}_j\right)$ in (3.9) with respect to $\widetilde{\mathbf{x}}_k^\kappa$, evaluated at zero, $\mathbf{z}_j^\kappa$ is the residual:

$$\mathbf{z}_j^\kappa = \mathbf{h}_j \left(\widehat{\mathbf{x}}_k^\kappa, \mathbf{0}\right) = \mathbf{u}_j^T \left({}^G\widehat{\mathbf{T}}_{I_k}^\kappa {}^I\widehat{\mathbf{T}}_{L_k}^\kappa {}^L\mathbf{p}_j - {}^G\mathbf{q}_j\right), \tag{3.13}$$

and $\mathbf{r}_j = {}^{G}\mathbf{u}_j^{T\,G}\mathbf{T}_{I_k}\,{}^{I}\mathbf{T}_{L}\,{}^{L}\mathbf{n}_j \in \mathcal{N}(\mathbf{0}, \mathbf{R}_j)$ is the total measurement noise with a covariance $\mathbf{R}_j$ due to the raw LiDAR measurement noise ${}^{L}\mathbf{n}_j$. In practice, we set $\mathbf{R}_j$ to a constant value and found it works very well.

3.4.2.3 Iterated Update

The propagated state $\widehat{\mathbf{x}}_k$ and covariance $\widehat{\mathbf{P}}_k$ from Section 3.4.2.1 impose a prior Gaussian distribution for the unknown state $\mathbf{x}_k$. More specifically, $\widehat{\mathbf{P}}_k$ represents the covariance of the following error state:

$$\begin{aligned} \mathbf{x}_k \boxminus \widehat{\mathbf{x}}_k &= (\widehat{\mathbf{x}}_k^{\kappa} \boxplus \widetilde{\mathbf{x}}_k^{\kappa}) \boxminus \widehat{\mathbf{x}}_k = \widehat{\mathbf{x}}_k^{\kappa} \boxminus \widehat{\mathbf{x}}_k + \mathbf{J}^{\kappa}\widetilde{\mathbf{x}}_k^{\kappa} \\ &\sim \mathcal{N}(\mathbf{0}, \widehat{\mathbf{P}}_k) \end{aligned} \tag{3.14}$$

where $\mathbf{J}^{\kappa}$ is the partial differentiation of $(\widehat{\mathbf{x}}_k^{\kappa} \boxplus \widetilde{\mathbf{x}}_k^{\kappa}) \boxminus \widehat{\mathbf{x}}_k$ with respect to $\widetilde{\mathbf{x}}_k^{\kappa}$ evaluated at zero:

$$\mathbf{J}^{\kappa} = \begin{bmatrix} \mathbf{A}\left(\delta^{G}\boldsymbol{\theta}_{I_k}\right)^{-T} & \mathbf{0}_{3\times15} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{15\times3} & \mathbf{I}_{15\times15} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times15} & \mathbf{A}\left(\delta^{I}\boldsymbol{\theta}_{L_k}\right)^{-T} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times15} & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} \end{bmatrix} \tag{3.15}$$

where $\mathbf{A}(\cdot)^{-1}$ is defined in [130, 181], $\delta^{G}\boldsymbol{\theta}_{I_k} = {}^{G}\widehat{\mathbf{R}}_{I_k}^{\kappa} \boxminus {}^{G}\widehat{\mathbf{R}}_{I_k}$ and $\delta^{I}\boldsymbol{\theta}_{L_k} = {}^{I}\widehat{\mathbf{R}}_{L_k}^{\kappa} \boxminus {}^{I}\widehat{\mathbf{R}}_{L_k}$ is the error states of IMU's attitude and rotational extrinsic, respectively. For the first iteration, $\widehat{\mathbf{x}}_k^{\kappa} = \widehat{\mathbf{x}}_k$, then $\mathbf{J}^{\kappa} = \mathbf{I}$.

Besides the prior distribution, we also have a distribution of the state due to the measurement (3.12):

$$-\mathbf{r}_j = \mathbf{z}_j^{\kappa} + \mathbf{H}_j^{\kappa}\widetilde{\mathbf{x}}_k^{\kappa} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_j) \tag{3.16}$$

Combining the prior distribution in (3.14) with the measurement model from (3.16) yields the posteriori distribution of the state $\mathbf{x}_k$ equivalently represented by $\widetilde{\mathbf{x}}_k^{\kappa}$ and its maximum a-posteriori estimate (MAP):

$$\min_{\widetilde{\mathbf{x}}_k^{\kappa}} \left(\|\mathbf{x}_k \boxminus \widehat{\mathbf{x}}_k\|_{\widehat{\mathbf{P}}_k}^2 + \sum\nolimits_{j=1}^{m} \|\mathbf{z}_j^{\kappa} + \mathbf{H}_j^{\kappa}\widetilde{\mathbf{x}}_k^{\kappa}\|_{\mathbf{R}_j}^2 \right) \tag{3.17}$$

where $\|\mathbf{a}\|^2_{\mathbf{M}} \triangleq \mathbf{a}^T\mathbf{M}^{-1}\mathbf{a}$ for any invertible matrix $\mathbf{M}$ and vector $\mathbf{a}$ of the proper dimension, and m is the number of measured points. This MAP problem can be solved by iterated Kalman filter as below (to simplify the notation, let $\mathbf{H} = [\mathbf{H}_1^{\kappa^T}, \cdots, \mathbf{H}_m^{\kappa^T}]^T$, $\mathbf{R} = \text{diag}\,(\mathbf{R}_1, \cdots \mathbf{R}_m)$,$\mathbf{P} = (\mathbf{J}^\kappa)^{-1}\,\widehat{\mathbf{P}}_k(\mathbf{J}^\kappa)^{-T}$, and $\mathbf{z}_k^\kappa = \left[\mathbf{z}_1^{\kappa^T}, \cdots, \mathbf{z}_m^{\kappa^T}\right]^T$):

$$\begin{aligned} \mathbf{K} &= \left(\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H} + \mathbf{P}^{-1}\right)^{-1}\mathbf{H}^T\mathbf{R}^{-1}, \\ \widehat{\mathbf{x}}_k^{\kappa+1} &= \widehat{\mathbf{x}}_k^\kappa \boxplus \left(-\mathbf{K}\mathbf{z}_k^\kappa - (\mathbf{I} - \mathbf{K}\mathbf{H})(\mathbf{J}^\kappa)^{-1}\left(\widehat{\mathbf{x}}_k^\kappa \boxminus \widehat{\mathbf{x}}_k\right)\right). \end{aligned} \tag{3.18}$$

Notice that the Kalman gain $\mathbf{K}$ computation needs to invert a matrix of the state dimension instead of the measurement dimension used in previous works.

The above process repeats until convergence (i.e., $\|\widehat{\mathbf{x}}_k^{\kappa+1} \boxminus \widehat{\mathbf{x}}_k^\kappa\| < \epsilon$). After convergence, the optimal state and covariance estimates are:

$$\bar{\mathbf{x}}_k = \widehat{\mathbf{x}}_k^{\kappa+1},\ \bar{\mathbf{P}}_k = (\mathbf{I} - \mathbf{K}\mathbf{H})\,\mathbf{P} \tag{3.19}$$

With the state update $\bar{\mathbf{x}}_k$, each LiDAR point (${}^L\mathbf{p}_j$) in the k-th scan is then transformed to the global frame via:

$${}^G\bar{\mathbf{p}}_j = {}^G\bar{\mathbf{T}}_{I_k}\,{}^I\bar{\mathbf{T}}_{L_k}\,{}^L\mathbf{p}_j;\ j = 1, \cdots, m. \tag{3.20}$$

The transformed LiDAR points $\{{}^G\bar{\mathbf{p}}_j\}$ are inserted to the map represented by ikd-Tree (see Section 3.5). Our state estimation is summarized in **Algorithm 5**.

3.5 Mapping

In this section, we describe how to incrementally maintain a map (i.e., insertion and delete) and perform k-nearest search on it by ikd-Tree.

Algorithm 5: State Estimation

Input : Last output $\bar{\mathbf{x}}_{k-1}$ and $\bar{\mathbf{P}}_{k-1}$;
LiDAR raw points in current scan;
IMU inputs ($\mathbf{a}_m$, $\boldsymbol{\omega}_m$) during current scan.

Forward propagation to obtain state prediction $\widehat{\mathbf{x}}_k$ and its covariance $\widehat{\mathbf{P}}_k$ via (3.10);
Backward propagation to compensate motion [130];
$\kappa = -1$, $\widehat{\mathbf{x}}_k^{\kappa=0} = \widehat{\mathbf{x}}_k$;
repeat
  $\kappa = \kappa + 1$;
  Compute $\mathbf{J}^\kappa$ via (3.15) and $\mathbf{P} = (\mathbf{J}^\kappa)^{-1}\widehat{\mathbf{P}}_k(\mathbf{J}^\kappa)^{-T}$;
  Compute residual $\mathbf{z}_j^\kappa$ and Jacobin $\mathbf{H}_j^\kappa$ via (3.12) (3.13);
  Compute the state update $\widehat{\mathbf{x}}_k^{\kappa+1}$ via (3.18);
until $\|\widehat{\mathbf{x}}_k^{\kappa+1} \boxminus \widehat{\mathbf{x}}_k^{\kappa}\| < \epsilon$;
$\bar{\mathbf{x}}_k = \widehat{\mathbf{x}}_k^{\kappa+1}$; $\bar{\mathbf{P}}_k = (\mathbf{I} - \mathbf{K}\mathbf{H})\,\mathbf{P}$;
Obtain the transformed LiDAR points $\{{}^G\bar{\mathbf{p}}_j\}$ via (3.20).

Output: Current optimal estimate $\bar{\mathbf{x}}_k$ and $\bar{\mathbf{P}}_k$;
The transformed LiDAR points $\{{}^G\bar{\mathbf{p}}_j\}$.

3.5.1 Map Management

The map points are organized into an ikd-Tree, which dynamically grows by merging a new scan of point cloud at the odometry rate. To prevent the size of the map from going unbound, only map points in a large local region of length L around the LiDAR current position are maintained on the ikd-Tree. A 2D demonstration is shown in Fig. 3.3. The map region is initialized as a cube with length L, which is centered at the initial LiDAR position $\mathbf{p}_0$. The detection area of LiDAR is assumed to be a detection ball centered at the LiDAR current position obtained from (3.19). The radius of the detection ball is assumed to be $r = \gamma R$ where R is the LiDAR FoV range, and γ is a relaxation parameter larger than 1. When the LiDAR moves to a new position $\mathbf{p}'$ where the detection ball touches the boundaries of the map, the map region is moved in a direction that increases the distance between the LiDAR detection area and the touching boundaries. The distance that the map region moves is set to a constant $d = (\gamma - 1)R$. All points in the subtraction area between the new map region and the old one will be deleted from the ikd-Tree by a box-wise delete operation detailed in Section 3.5.3.

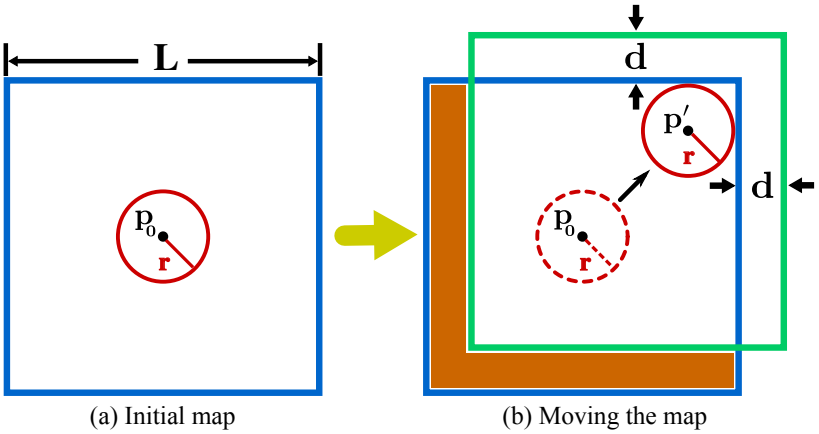


Figure 3.3: 2D demonstration of map region management. In (a), the blue rectangle is the initial map region with length L. The red circle is the initial detection area centered at the initial LiDAR position $\mathbf{p}_0$. In (b), the detection area (dashed red circle) moves to a new position $\mathbf{p}'$(circle with solid red line) where the map boundaries are touched. The map region is moved to a new position (green rectangle) by distance d. The points in the subtraction area (orange area) are removed from the map (i.e., ikd-Tree).

3.5.2 Tree Structure and Construction

3.5.2.1 Data Structure

Different from many existing implementations of k-d trees which store a "bucket" of points only on leaf nodes [147, 149, 176, 177, 179], ikd-Tree stores points on both leaf nodes and internal nodes to better support dynamic point insertion and tree re-balancing. Such storing mode has also shown to be more efficient in kNN search when a single k-d tree is used [175], which is the case of the ikd-Tree. Since a point corresponds to a single node on the ikd-Tree, we will use points and nodes interchangeably.

To enhance the readability for our readers, we provide a comprehensive explanation about the attributes of the tree nodes here, as shown in **Data Structure**. The point information (e.g., point coordinates, intensity) are stored in `point`. The attributes `leftchild` and `rightchild` are pointers to its left and right child node, respectively. The division axis to split the space is recorded in `axis`. The number of tree nodes, including both valid and invalid nodes, of the (sub-)tree rooted at the current node is maintained in attribute `treesize`. When points are removed from the map, the nodes are not deleted from the tree immediately, but only setting the boolean variable `deleted` to be true (see Section 3.5.3.2 for details). If the entire (sub-)tree rooted at the current node is removed, `treedeleted` is set to true. The number of points deleted

from the (sub-)tree is summed up into attribute `invalidnum`. The attribute `range` records the range information of the points on the (sub-)tree, which is interpreted as a circumscribed axis-aligned cuboid containing all the points. The circumscribed cuboid is represented by its two diagonal vertices with minimal and maximal coordinates on each dimension, respectively. Note that since re-insert functions are not used in FAST-LIO2, the attribute `pushdown` and related `Pushdown` operation presented in Chapter 2 are no longer required.

Data Structure: Tree node structure

```
Struct TreeNode:
    PointType point;
    TreeNode * leftchild, * rightchild;
    int axis;
    int treesize, invalidnum;
    bool deleted, treedeleted;
    CuboidVertices range;
end
```

3.5.2.2 Construction

Building the ikd-Tree is similar to building a static k-d tree in [92]. The ikd-Tree splits the space at the median point along the longest dimension recursively until there is only one point in the subspace. The attributes in **Data Structure** are initialized during the construction, including calculating the tree size and range information of (sub-)trees. The detailed implementation of constructing an ikd-Tree is provided in Algorithm 1, Section 2.3.2.

3.5.3 Incremental Updates

The incremental updates in FAST-LIO2 requires two incremental operations in ikd-Tree: the point-wise insertion with on-tree down-sampling and the box-wise delete. We explain the specialized algorithms with details, followed by dynamic re-balancing detailed in Section 3.5.4.

3.5.3.1 Point Insertion with On-tree Downsampling

In consideration of robotic applications, ikd-Tree supports simultaneous point insertion and map downsampling. The algorithm is detailed in **Algorithm 6**. For a given

Table 3.2: Attributes Initialization of a New Tree Node to Insert

Attribute	Value	Attribute	Value
`point`	$\mathbf{p}$	`axis`[1]	(`father.axis` + 1) **mod** k
`leftchild`	NULL	`rightchild`	NULL
`treesize`	1	`invalidnum`	0
`deleted`	false	`treedeleted`	false
`range`[2]	$[\mathbf{p}, \mathbf{p}]$		

[1] The *axis* is initialized using the division axis of its father node.
[2] The cuboid is initialized by setting minimal and maximal vertices as the point to insert.

point $\mathbf{p}$ in $\{{}^{G}\bar{\mathbf{p}}_j\}$ from the state estimation module (see **Algorithm 5**) and downsample resolution l, the algorithm partitions the space evenly into cubes of length l, then the cube $\mathbf{C}_D$ that contains the point $\mathbf{p}$ is found (Line 2). The algorithm only keeps the point that is nearest to the center $\mathbf{p}_{center}$ of $\mathbf{C}_D$ (Line 3). This is achieved by firstly searching all points contained in $\mathbf{C}_D$ on the k-d tree and stores them in a point array V together with the new point $\mathbf{p}$ (Line 4-5). The nearest point $\mathbf{p}_{nearest}$ is obtained by comparing the distances of each point in V to the center $\mathbf{p}_{center}$ (Line 6). Then existing points in $\mathbf{C}_D$ are deleted (Line 7), after which the nearest point $\mathbf{p}_{nearest}$ is inserted into the k-d tree (Line 8). The implementation of box-wise search is similar to the box-wise delete as introduced in Section 3.5.3.2.

The point insertion (Line 11-24) on the ikd-Tree is implemented recursively. The algorithm searches down from the root node until an empty node is found to append a new node (Line 12-14). The attributes of the new leaf node are initialized as Table 3.2. At each non-empty node, the new point is compared with the point stored on the tree node along the division axis for further recursion (Line 15-20). The attributes (e.g., `treesize`, `range`) of those visited nodes are updated with the latest information (Line 21) as introduced in Section 3.5.3.3. A balance criterion is checked and maintained for sub-trees updated with the new point to keep the balance property of ikd-Tree (Line 22) as detailed in Section 3.5.4.

3.5.3.2 Box-wise Delete using Lazy Labels

In the delete operation, we use a lazy delete strategy. That is, the points are not removed from the tree immediately but only labeled as "deleted" by setting the attribute

Algorithm 6: Point Insertion with On-tree Downsampling

Input: Downsample Resolution l,
New Point to Insert $\mathbf{p}$,
Switch of Parallelly Re-building SW

Algorithm Start
  $\mathbf{C}_D \leftarrow$ `FindCube(`$l, \mathbf{p}$`)`
  $\mathbf{p}_{center} \leftarrow$ `Center(`$\mathbf{C}_D$`)`;
  $V \leftarrow$ `BoxwiseSearch(`*RootNode*`,`$\mathbf{C}_D$`)`;
  $V.push(\mathbf{p})$;
  $\mathbf{p}_{nearest} \leftarrow$ `FindNearest` $(V, \mathbf{p}_{center})$;
  `BoxwiseDelete(`*RootNode*`,`$\mathbf{C}_D$`)`
  `Insert(`*RootNode*`,`$\mathbf{p}_{nearest}$`,`*NULL*`,`*SW*`)`;
Algorithm End

Function `Insert(`*T*, $\mathbf{p}$, *father*`,`*SW*`)`
  if *T is empty* **then**
    `Initialize(`*T*`,`$\mathbf{p}$`,`*father*`)`;
  else
    `ax` $\leftarrow$ `T.axis`;
    if $\mathbf{p}[ax] <$ *T.point*$[ax]$ **then**
      `Insert(`*T.leftchild*`,`$\mathbf{p}$`,`*T*`,`*SW*`)`;
    else
      `Insert(`*T.rightchild*`,`$\mathbf{p}$`,`*T*`,`*SW*`)`;
    end
    `AttributeUpdate(`*T*`)`;
    `Rebalance(`*T*`,`*SW*`)`;
  end
End Function

`deleted` to true (see **Data Structure**, Line 6). If all nodes on the sub-tree rooted at node T have been deleted, the attribute `treedeleted` of T is set to true. Therefore the attributes `deleted` and `treedeleted` are called lazy labels. Points labeled as "deleted" will be removed from the tree during a re-building process (see Section 3.5.4).

Box-wise delete is implemented utilizing the range information in attribute `range` and the lazy labels on the tree nodes. As mentioned in 3.5.2, the attribute *range* is represented by a circumscribed cuboid $\mathbf{C}_T$. The pseudo-code is shown in **Algorithm 7**. Given the cuboid of points $\mathbf{C}_O$ to be deleted from a (sub-)tree rooted at T, the algorithm searches down the tree recursively and compares the circumscribed cuboid $\mathbf{C}_T$ with the given cuboid $\mathbf{C}_O$. If there is no intersection between $\mathbf{C}_T$ and $\mathbf{C}_O$, the recursion returns directly without updating the tree (Line 2). If the circumscribed cuboid $\mathbf{C}_T$ is fully contained in the given cuboid $\mathbf{C}_O$, the box-wise delete set attributes `deleted` and `treedeleted` to true (Line 5). As all points on the (sub-)tree are deleted, the attribute `invalidnum` is equal to the `treesize` (Line 6). For the condition that $\mathbf{C}_T$ intersects but not contained in $\mathbf{C}_O$, the current point $\mathbf{p}$ is firstly deleted from the tree if it is contained in $\mathbf{C}_O$ (Line 9), after which the algorithm looks into the child nodes recursively (Line 10-11). The attribute update of the current node T and the balance maintenance is applied after the box-wise delete operation (Line 12-13).

Algorithm 7: Box-wise Delete

Input : Operation Cuboid $\mathbf{C}_O$,
k-d Tree Node T,
Switch of Parallelly Re-building SW

```
1  Function BoxwiseDelete(T,C_O,SW)
2      C_T ← T.range;
3      if C_T ∩ C_O = ∅ then return;
4      if C_T ⊆ C_O then
5          T.treedelete, T.delete ← true;
6          T.invalidnum = T.treesize;
7      else
8          p ← T.point;
9          if p ⊂ C_O then T.treedelete = true;
10         BoxwiseDelete(T.leftchild,C_O,SW);
11         BoxwiseDelete(T.rightchild,C_O,SW);
12         AttributeUpdate(T);
13         Rebalance(T,SW);
14     end
15 End Function
```

3.5.3.3 Attribute Update

After each incremental operation, attributes of the visited nodes are updated with the latest information using function `AttributeUpdate`. The function calculates the attributes `treesize` and `invalidnum` by summarizing the corresponding attributes on its two child nodes and the point information on itself; the attribute `range` is determined by merging the range information of the two child nodes and the point information stored on it; `treedeleted` is set true if the `treedeleted` of both child nodes are true and the node itself is deleted.

3.5.4 Re-balancing

As we discussed in Chapter 2, ikd-Tree actively monitors the balance property and dynamically re-balances itself by only re-building the relevant sub-trees after each incremental operation. To ensure comprehensive coverage, we present the entire process of re-building method in **Algorithm 8**.

When the balance criterion in Section 2.3.4.1 is violated, the sub-tree is rebuilt either in the main thread or the second thread, depending on the size of the tree. Specifically, if the tree size is smaller than a predetermined value $N_{\max}$, the re-building process occurs in the main thread. Conversely, if the size exceeds $N_{\max}$, the re-building algorithm is executed in the second thread using the `ParRebuild` function. The second thread locks all incremental updates (e.g., point insertion and deletion) pertaining to the sub-tree, while still permitting queries (Line 12). Subsequently, the second thread proceeds to flatten all valid points contained within the sub-tree $\mathcal{T}$ into a point array V, ensuring the original sub-tree remains unchanged for potential queries during the re-building process (Line 13). After the flattening operation, the original sub-tree is unlocked, allowing the main thread to continue processing incremental update requests (Line 14). These requests are concurrently recorded in a queue named operation logger. Once the second thread successfully constructs a new balanced k-d tree $\mathcal{T}'$ from the point array V (Line 15), the recorded update requests are re-applied to $\mathcal{T}'$ using the `IncrementalUpdates` function (Line 16-18). It is important to note that the parallel re-building switch is set to false, given that the execution is already taking place in the second thread. Upon completion of all pending requests, the point information in

the original sub-tree $\mathcal{T}$ is entirely equivalent to that in the new sub-tree $\mathcal{T}'$, except for the improved balance of the latter in its tree structure. The algorithm proceeds to lock node T against incremental updates and queries, subsequently replacing it with the newly constructed node T' (Line 20-22). Finally, the algorithm releases the memory of the original sub-tree (Line 23). This design ensures that during the re-building process in the second thread, the mapping process in the main thread continues uninterrupted at the odometry rate, albeit with reduced efficiency due to the temporary imbalance in the k-d tree structure. Notably, the `LockUpdates` function does not block queries, which can be conducted in parallel within the main thread. In contrast, the `LockAll` function blocks all forms of access, including queries, but executes rapidly (i.e., requiring only one instruction), facilitating timely queries within the main thread. Both `LockUpdates` and `LockAll` are implemented using mutual exclusion (mutex).

Algorithm 8: Rebuild (sub-) tree for re-balancing

```
   Input: Root node T of (sub-) tree 𝒯 for re-building,
          Re-build Switch SW
 1 Function Rebalance(T,SW)
 2 |  if ViolateCriterion(T) then
 3 |  |  if T.treesize < N_max or Not SW then
 4 |  |  |  Rebuild(T)
 5 |  |  else
 6 |  |  |  ThreadSpawn(ParRebuild,T)
 7 |  |  end
 8 |  end
 9 End Function
10
11 Function ParRebuild(T)
12 |  LockUpdates(T);
13 |  V ← Flatten(T);
14 |  Unlock(T);
15 |  T' ← Build(V);
16 |  foreach op in OperationLogger do
17 |  |  IncrementalUpdates(T',op,false)
18 |  end
19 |  T_temp ← T;
20 |  LockAll(T);
21 |  T ← T';
22 |  Unlock(T);
23 |  Free(T_temp);
24 End Function
```

3.5.5 K-Nearest Neighbor Search

Though being similar to existing implementations in those well-known k-d tree libraries [149, 176, 177], the nearest search algorithm on ikd-Tree is thoroughly optimized in FAST-LIO2. The range information on the tree nodes is well utilized to speed up our nearest neighbor search using a "bounds-overlap-ball" test detailed in [175]. A priority queue $\mathtt{q}$ is maintained to store the k-nearest neighbors so far encountered and their distance to the target point. When recursively searching down the tree from its root node, the minimal distance $d_{\min}$ from the target point to the cuboid $\mathbf{C}_T$ of the tree node is calculated firstly. If the minimal distance $d_{\min}$ is no smaller than the maximal distance in $\mathtt{q}$, there is no need to process the node and its offspring nodes. Furthermore, in FAST-LIO2 (and many other LiDAR odometry), only when the neighbor points are within a given threshold around the target point would be viewed as inliers and hence used in the state estimation, this naturally provides a maximal search distance for a ranged search of k-nearest neighbors [176]. In either case, the ranged search prunes the algorithm by comparing $d_{\min}$ with the maximal distance, thus reducing the amount of backtracking to improve the time performance. It should be noted that our ikd-Tree supports multi-thread k-nearest neighbor search for parallel computing architectures.

3.6 Benchmark Results

In this section, extensive experiments in terms of accuracy, robustness, and computational efficiency are conducted on various open datasets. We first evaluate our data structure, i.e., ikd-Tree, against other data structures for kNN search on 18 dataset sequences of different sizes. Then in Section 3.6.3, we compare the accuracy and processing time of FAST-LIO2 on 19 sequences. All the sequences are chosen from 5 different datasets collected by both solid-state LiDAR [21] and spinning LiDARs. The first dataset is from the work LILI-OM [163] and is collected by a solid-state 3D LiDAR Livox Horizon[3], which has non-repetitive scan pattern and 81.7° (Horizontal) × 25.1° (Vertical) FoV, at a typical scan rate of 10 Hz, referred to as *lili*. The gyroscope and accelerometer measurements are sampled at 200 Hz by a 6-axis Xsens MTi-670 IMU. The data is recorded in the university campus and urban streets with structured scenes. The second

[3] https://www.livoxtech.com/horizon

dataset is from the work LIO-SAM [169] in MIT campus and contains several sequences collected by a VLP-16 LiDAR[4] sampled at 10 Hz and a MicroStrain 3DM-GX5-25 9-axis IMU sampled at 1000 Hz, referred to as *liosam*. It contains different kinds of scenes, including structured buildings and forests on campus. The third dataset "*utbm*" [182] is collected with a human-driving robocar in maximum 50 km/h speed which has two 10 Hz Velodyne HDL-32E LiDAR[5] and 100 Hz Xsens MTi-28A53G25 IMU. In this paper, we only consider the left LiDAR. The fourth dataset "*ulhk*" [183] contains the 10 Hz LiDAR data from Velodyne HDL-32E and 100 Hz IMU data from a 9-axis Xsens MTi-10 IMU. All the sequences of *utbm* and *ulhk* are collected in structured urban areas by a human-driving vehicle while *ulhk* also contains many moving vehicles. The last one, "*nclt*" [184] is a large-scale, long-term autonomy UGV (unmanned ground vehicle) dataset collected in the University of Michigan's North Campus. The *nclt* dataset contains 10 Hz data from a Velodyne HDL-32E LiDAR and 50 Hz data from Microstrain MS25 IMU. The *nclt* dataset has a much longer duration and amount of data than other datasets and contains several open scenes such as a large open parking lot. The datasets information including the sensors' type and data rate is summarized in Table 3.3. The details about all the 37 sequences used in this section, including name, duration, and distance, are listed in Table 3.4.

Table 3.3: The Datasets for Benchmark

	LiDAR		IMU	
	Type	Line	Type	Rate
lili	Solid-state	—	6-axis	200 Hz
utbm	Spinning	32	6-axis	100 Hz
ulhk	Spinning	32	9-axis	100 Hz
nclt	Spinning	32	9-axis	100 Hz
liosam	Spinning	16	9-axis	1000 Hz

[1] In order to make LIO-SAM works, the IMU rate in dataset *nclt* is increased from 50 Hz to 100 Hz through zero-order interpolation.

3.6.1 Implementation

We implemented the proposed FAST-LIO2 system in C++ and Robots Operating System (ROS). The iterated Kalman filter is implemented based on the *IKFOM* toolbox presented in our previous work [181]. In the default configuration, the local map size

[4] https://velodynelidar.com/products/puck-lite/
[5] https://velodynelidar.com/products/hdl-32e/

Table 3.4: Details of all the sequences for the Benchmark

	Name	Duration (min:sec)	Distance (km)
lili_1	FR-IOSB-Tree	2:58	0.36
lili_2	FR-IOSB-Long	6:00	1.16
lili_3	FR-IOSB-Short	4:39	0.49
lili_4	KA-URBAN-Campus-1	5:58	0.50
lili_5	KA-URBAN-Campus-2	2:07	0.20
lili_6	KA-URBAN-Schloss-1	10:37	0.65
lili_7	KA-URBAN-Schloss-2	12:17	1.10
lili_8	KA-URBAN-East	20:52	3.70
utbm_1	20180713	16:59	5.03
utbm_2	20180716	15:59	4.99
utbm_3	20180717	15:59	4.99
utbm_4	20180718	16:39	5.00
utbm_5	20180720	16:45	4.99
utbm_6	20190110	10:59	3.49
utbm_7	20190412	12:11	4.82
utbm_8	20180719	15:26	4.98
utbm_9	20190131	16:00	6.40
utbm_10	20190418	11:59	5.11
ulhk_1	HK-Data20190316-1	2:55	0.23
ulhk_2	HK-Data20190426-1	2:30	0.55
ulhk_3	HK-Data20190317	5:18	0.62
ulhk_4	HK-Data20190117	5:18	0.60
ulhk_5	HK-Data20190316-2	6:05	0.66
ulhk_6	HK-Data20190426-2	4:20	0.74
nclt_1	20120118	93:53	6.60
nclt_2	20120122	87:19	6.36
nclt_3	20120202	98:37	6.45
nclt_4	20120115	111:46	4.01
nclt_5	20120429	43:17	1.86
nclt_6	20120511	84:32	3.13
nclt_7	20120615	55:10	1.62
nclt_8	20121201	75:50	2.27
nclt_9	20130110	17:02	0.26
nclt_10	20130405	69:06	1.40
liosam_1	park	9:11	0.66
liosam_2	garden	5:58	0.46
liosam_3	campus	16:26	1.44

L is chosen as 1000 m, and the LiDAR raw points are directly fed into state estimation after a 1:4 (one out of four LiDAR points) temporal downsampling. Besides, the spatial downsample resolution (see **Algorithm 6**) is set to $l = 5$m for all the experiments. The parameter of ikd-Tree is set to $\alpha_{bal} = 0.6$, $\alpha_{del} = 0.5$ and $N_{\max} = 1500$. The parameter of Kalman filter is set to $\mathbf{R}_j = 0.01$. The computation platform for benchmark comparison is a lightweight UAV onboard computer: DJI Manifold 2-C[6] with a 1.8 GHz quad-core Intel i7-8550U CPU and 8 GB RAM. For FAST-LIO2, we also test it on an ARM processor that is typically used in embedded systems with reduced power and cost. The ARM platform is Khadas VIM3[7] which has a low-power 2.2 GHz quad-core Cortex-A73 CPU and 4 GB RAM, denoted as the keyword "ARM". We denote "FAST-LIO2 (ARM)" as the implementation of FAST-LIO2 on the ARM-based platform.

3.6.2 Data structure Evaluation

3.6.2.1 Evaluation Setup

We select three state-of-art implementations of dynamic data structure to compare with our ikd-Tree: The boost geometry library implementation of R*-tree [185], the Point Cloud Library implementation of octree [186] and the *nanoflann* [179] implementation of dynamic k-d tree. These tree data structure implementations are chosen because of their high implementation efficiency. Moreover, they support dynamic operations (i.e., point insertion, delete) and range (or radius) search that is necessary to be integrated with FAST-LIO2 for a fair comparison with ikd-Tree. For the map downsampling, since the other data structures do not support on-tree downsampling as ikd-Tree, we apply a similar approach as detailed in 3.5.3 by utilizing their ability of range search (for octree and R*-tree) or radius search (for *nanoflann* k-d tree). More specifically, for octree and R*-tree, their range search directly returns points within a downsampling cube $\mathbf{C}_D$ (see **Algorithm 6**). For *nanoflann* k-d tree, the points inside the circumcircle of the downsampling cube $\mathbf{C}_D$ are obtained by radius search, after which points outside the cube are filtered out via a linear approach while points inside the cube $\mathbf{C}_D$ are retained. Finally, similar to **Algorithm 6**, points in $\mathbf{C}_D$ other than the nearest point to the center are removed from the map. For the box-wise delete operation required by map move

[6] https://www.dji.com/cn/manifold-2/specs
[7] https://www.khadas.com/vim3

(see Section 3.5.1), it is achieved by removing points within the specified cuboid $\mathbf{C}_O$ one by one according to the point indices obtained from the respective range or radius search.

All the four data structure implementations are integrated with FAST-LIO2 and their time performance are evaluated on 18 sequences of different sizes. We run the FAST-LIO2 with each data structure for each sequence and record the time for kNN search, point insertion (with map downsampling), box-wise delete due to map move, the number of new scan points, and the number of map points (i.e., tree size) at each step. The number of nearest neighbors to find is 5.

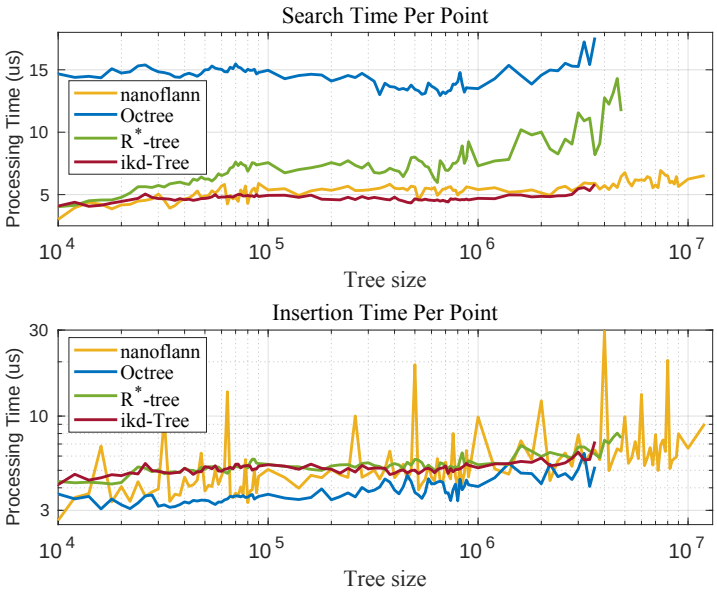


Figure 3.4: Data structure comparison over different tree size. The upper figure shows the average processing time of searching five nearest neighbors. The bottom figure shows the average processing time of inserting one point to the data structure.

3.6.2.2 Comparison Results

We first compare the time consumption of point insertion (with map downsampling) and kNN search at different tree sizes across all the 18 sequences. For each evaluated tree size S, we collect the processing time at tree size in range of $[0.95S, 1.05S]$ to obtain a sufficient number of samples. Fig. 3.4 shows the average time consumption of insertion and kNN search per single target point. The octree achieves the best performance in point insertion, albeit the gap with the other is small (below 1 μs), but its inquiry time is much higher due to the unbalanced tree structure. For *nanoflann* k-d tree, the insertion

Table 3.5: The Comparison of Average Time Consumption Per Scan on Incremental Updates, kNN Search and Total Time

	Incremental Update[1][ms]				kNN Search[2][ms]				Total [ms]			
	ikd-Tree	nanoflann	Octree	R*-tree	ikd-Tree	nanoflann	Octree	R*-tree	ikd-Tree	nanoflann	Octree	R*-tree
utbm_1	3.23	3.43	**2.12**	3.94	**15.19**	15.80	42.88	22.56	**18.42**	19.22	45.00	26.50
utbm_2	3.40	3.65	**2.24**	4.18	**15.52**	16.09	44.70	23.46	**18.93**	19.75	46.94	27.64
utbm_3	3.77	4.17	**2.36**	4.52	**16.83**	18.54	45.72	23.12	**20.60**	22.70	48.08	27.64
utbm_4	3.52	3.70	**2.26**	4.32	**16.53**	17.60	44.80	24.74	**20.06**	21.30	47.06	29.06
utbm_5	3.34	3.60	**2.21**	4.21	**15.51**	16.65	45.42	23.38	**18.85**	20.25	47.63	27.58
utbm_6	3.61	4.12	**2.34**	4.60	**16.25**	17.14	43.06	23.49	**19.86**	21.27	45.40	28.09
utbm_7	3.82	4.62	**2.55**	5.26	**15.42**	16.97	42.06	25.87	**19.24**	21.59	44.61	31.13
ulhk_1	1.97	1.87	**1.12**	2.30	**18.23**	21.73	48.30	23.45	**20.20**	23.60	49.43	25.75
ulhk_2	3.51	3.43	**2.32**	4.23	**22.26**	26.07	64.56	31.75	**25.77**	29.49	66.88	35.98
ulhk_3	1.60	1.58	**1.10**	1.93	**13.62**	14.87	42.65	20.49	**15.22**	16.45	43.74	22.42
nclt_1	1.14	1.59	**0.99**	2.07	**14.50**	18.83	41.58	28.07	**15.64**	20.41	42.57	30.14
nclt_2	**1.35**	2.04	1.36	2.66	**14.68**	18.99	46.56	29.20	**16.03**	21.03	47.91	31.86
nclt_3	**1.00**	1.42	1.03	2.20	**14.41**	19.25	46.19	30.10	**15.42**	20.67	47.22	32.29
lili_1	1.41	1.42	**0.83**	1.79	**9.20**	9.71	26.31	12.65	**10.61**	11.13	27.15	14.44
lili_2	1.53	1.50	**0.84**	1.81	**8.94**	9.27	26.18	13.43	**10.47**	10.77	27.02	15.24
lili_3	1.10	1.14	**0.63**	1.38	**8.46**	8.87	25.45	13.18	**9.57**	10.00	26.08	14.56
lili_4	0.96	0.99	**0.62**	1.39	**10.69**	11.97	32.55	15.71	**11.65**	12.96	33.17	17.10
lili_5	1.22	1.28	**0.80**	1.63	**10.23**	11.34	33.53	12.78	**11.45**	12.62	34.33	14.41

[1] Average time consumption per scan of incremental updates, including point-wise insertion with on-tree downsampling and box-wise delete.
[2] Average time consumption per scan of single-thread kNN search.

time is often slightly shorter than the ikd-Tree and R*-tree, but huge peaks occasionally occur due to its logarithmic structure of organizing a series of k-d trees. Such peaks severely degrade the real-time ability, especially when maintaining a large map. For k-nearest neighbor search, *nanoflann* k-d tree consumes slightly higher time than our ikd-Tree, especially when the tree size becomes large ($10^5 \sim 10^6$). The R*-tree achieves a similar insertion time with ikd-Tree but with a significantly higher search time for large tree sizes. Finally, we can see that the time of insertion with on-tree downsampling and kNN search of ikd-Tree is indeed proportional to $\log n$, which is consistent with the time complexity analysis in Section 2.4.

For any map data structure to be used in LiDAR odometry and mapping, the total time for map inquiry (i.e., kNN search) and incremental map update (i.e., point insertion with downsampling and box-delete due to map move) ultimately affects the real-time ability. This total time is summarized in Table 3.5. It is seen that octree performs the best in incremental updates in most datasets, followed closely by the ikd-Tree and the *nanoflann* k-d tree. In kNN search, the ikd-Tree has the best performance while the ikd-Tree and *nanoflann* k-d tree outperforms the other two by large margins, which is consistent with the past comparative study [94, 176]. The ikd-Tree achieves the best overall performance among all other data structures.

We should remark that while the *nanoflann* k-d tree achieves seemly similar performance with ikd-Tree, the peak insertion time has more profound causes, and its impact on LiDAR odometry and mapping is severe. The *nanoflann* k-d tree deletes a point by only masking it without actually deleting it from the tree. Consequently, even with map downsampling and map move, the deleted points remain on the tree affecting the subsequent inquiry and insertion performance. The resultant tree size grows much quicker than ikd-Tree and others, a phenomenon also observed from Fig. 3.4. The effect could be small for short sequences (*ulhk* and *lili*) but becomes evident for long sequences (*utbm* and *nclt*). The tree size of *nanoflann* k-d tree exceeds 6×10^6 in *utbm* datasets and 10^7 in *nclt* datasets, whereas the maximal tree size of ikd-Tree reaches 2×10^6 and 3.6×10^6, respectively. The maximal processing time of incremental updates on *nanoflann* all exceeds 3 s in seven *utbm* datasets and 7 s in three *nclt* datasets while our ikd-Tree keeps the maximal processing time at 214.4 ms in *nclt_2* and smaller than 150 ms in the rest 17 sequences. While this peaked processing time of *nanoflann* k-d tree does not heavily

affect the overall real-time ability due to its low occurrence, it causes a catastrophic delay for subsequent control.

3.6.3 Accuracy Evaluation

In this section, we compare the overall system FAST-LIO2 against other state-of-the-art LiDAR-inertial odometry and mapping systems, including LILI-OM [163], LIO-SAM [169], and LINS [38]. Since FAST-LIO2 is an odometry without any loop detection or correction, for the sake of fair comparison, the loop closure module of LILI-OM and LIO-SAM was deactivated, while all other functions such as sliding window optimization are enabled. We also perform ablation study on FAST-LIO2: to understand the influence of the map size, we run the algorithm in various map sizes L of 2000 m, 800 m, 600 m, besides the default 1000 m; to evaluate the effectiveness of direct method against feature-based methods, we add a feature extraction module from FAST-LIO [130] (optimized for solid-state LiDAR) and BALM [46] (optimized for spinning LiDAR). The results are reported under the keyword "Feature". All the experiments are conducted in the Manifold 2-C platform (Intel).

We perform evaluations on all the five datasets: *lili, lisam, utbm, ulhk*, and *nclt*. Since not all sequences have ground truth (affected by the weather, GPS quality, etc.), we select a total of 19 sequences from the five datasets. These 19 sequences either have a good ground truth trajectory (as recommended by the dataset author) or end at the starting position. Therefore, two criteria, absolute translational error (RMSE) and end-to-end error, are computed and evaluated.

3.6.3.1 RMSE Benchmark

The RMSE are computed and reported in Table 3.6. It is seen that increasing the map size of FAST-LIO2 increases the overall accuracy as the new can is registered to older historical points. When the map size is over 2000 m, the accuracy increment is not persistent as the odometry drift may cause possible false point matches with too old map points, a typical phenomenon of any odometry. Moreover, the direct method outperforms the feature-based variant of FAST-LIO2 in most sequences except for two, *nclt_4* and *nclt_6*, where the difference is tiny and negligible. This proves the effectiveness of the direct method.

Table 3.6: Absolute translational errors (RMSE, meters) in sequences with good quality ground truth

	utbm_8	*utbm_9*	*utbm_10*	*ulhk_4*	*nclt_4*	*nclt_5*	*nclt_6*	*nclt_7*	*nclt_8*	*nclt_9*	*nclt_10*	*liosam_1*
FAST-LIO2 (2000m)	**25.3**	**51.6**	16.89	2.57	3.15	5.41	7.54	2.59	8.21	5.72	1.68	4.62
FAST-LIO2 (1000m)	27.29	**51.6**	**16.8**	2.57	3.21	5.42	7.55	**2.21**	**5.88**	5.56	**1.62**	**4.58**
FAST-LIO2 (800m)	25.8	51.86	17.23	2.57	3.25	**5.4**	7.55	2.49	6.49	5.95	1.67	**4.58**
FAST-LIO2 (600m)	27.75	52.09	17.3	2.57	3.05	5.41	7.58	2.47	6.47	5.8	1.69	**4.58**
FAST-LIO2 (Feature)	27.21	53.81	22.59	2.61	**2.86**	6.43	**7.41**	2.63	9.36	6.09	1.71	7.85
LILI-OM	59.48	782.11	17.59	**2.29**	11.6	11.5	275	13.1	21.5	**5.2**	68.9	18.78
LIO-SAM	—[1]	—	—	3.52	1163	6.82	×[2]	23.3	27.0	7.9	1525	4.75
LINS	48.17	54.35	60.48	3.11	60.8	1135	128.76	397.2	107.3	11.86	3155	880.92

[1] Dataset *utbm* does not produce the attitude quaternion data which is necessary for LIO-SAM, therefore LIO-SAM does not work on all the sequences in *utbm* dataset, denoted as —.

[2] × denotes that the system totally failed.

Compared with other LIO methods, FAST-LIO2 or its variant achieves the best performances in 17 of all 19 data sequences and is the most robust LIO method among all the experiments. The only two exceptions are on *ulhk_4* and *nclt_9* where LILI-OM shows slightly higher accuracy than FAST-LIO. Notably, LILI-OM shows very large drift in *utbm_9*, *nclt_4*, *nclt_6*, *nclt_8* and *nclt_10*. The reason is that its sliding-window back-end fusion (*mapping*) fails as the map point number grows large. Hence its pose estimation relies solely on the front-end *odometry* which quickly accumulates the drift. LINS works similarly badly in *nclt_5*, *nclt_6*, *nclt_7*, *nclt_10*. LIO-SAM also shows large drift at *nclt_4*, *nclt_10* due to the failure of back-end factor graph optimization with the very long time and long-distance data. The video of an example, *nclt_10* sequence, is available at https://youtu.be/2OvjGnxszf8. Besides, on other sequences where LILI-OM, LIO-SAM, and LINS can work normally, their performance is still outperformed by FAST-LIO2 with large margins. Finally, it should be noted that the sequence *liosam_1* is directly drawn from the work LIO-SAM [169] so the algorithm has been well-tuned for the data. However, FAST-LIO2 still achieves higher accuracy.

3.6.3.2 Drift Benchmark

The end-to-end errors are reported in Table 3.7. The overall trend is similar to the RMSE benchmark results. FAST-LIO2 or its variants achieves the lowest drift in 5 of the total 7 sequences. We show an example, *ulhk_6* sequence, in the video available at https://youtu.be/2OvjGnxszf8. It should be noted that the LILI-OM has tuned parameters for each of their own sequences *lili* while parameters of FAST-LIO2 are kept the same among all the sequences. LIO-SAM shows good performance in its own sequences *liosam_2* and *liosam_3* but cannot keep it on other sequences such as *ulhk*. The LINS performs worse than LIO-SAM in *liosam* and *ulhk* datasets and failed in *liosam_2* (garden sequence from [169]) because the two sequences are recorded with large rotation speeds while the feature points used by LINS are too few. Also, in most of the sequences, the feature-based FAST-LIO performs similarly to the direct method except for the sequence *lili_7*, which contains many trees and large open areas that feature extraction will remove many effective points from trees and faraway buildings.

Table 3.7: End to end errors (meters)

	lili_6	*lili_7*	*lili_8*	*ulhk_5*	*ulhk_6*	*liosam_2*	*liosam_3*
FAST-LIO2(2000m)	0.14	1.92	21.35	0.33	0.12	**<0.1**	9.23
FAST-LIO2(1000m)	**<0.1**	1.63	17.39	0.39	**<0.1**	**<0.1**	9.50
FAST-LIO2(800m)	**<0.1**	1.88	21.59	0.40	**<0.1**	**<0.1**	9.49
FAST-LIO2(600m)	0.22	**1.37**	23.74	0.39	**<0.1**	**<0.1**	9.23
FAST-LIO2(Feature)	0.20	3.89	21.99	**0.32**	**<0.1**	**<0.1**	12.11
LILI-OM	0.80	4.13	**15.60**	1.84	7.89	1.95	13.79
LIO-SAM	—[1]	—	—	0.83	2.88	**<0.1**	**8.61**
LINS	—	—	—	0.90	6.92	×[2]	29.90

[1] Since the LIO-SAM and LINS are both developed only for spinning LiDAR, they do not work on the *lili* dataset which is recorded by a solid-state LiDAR Livox Horizon.
[2] × denotes that the system totally failed.

3.6.4 Processing Time Evaluation

Table 3.8 shows the processing time of FAST-LIO2 with different configurations, LILI-OM, LIO-SAM, and LINS in all the sequences. The FAST-LIO2 is an integrated odometry and mapping architecture, where at each step the map is updated following immediately the odometry update. Therefore, the total time ("Total" in Table 3.8) includes all possible procedures occurred in the odometry, including feature extraction if any (e.g., for the feature-based variant), motion compensation, kNN search, and state estimation, and mapping. It should be noted that the mapping includes point insertion, box-wise delete, and tree re-balancing. On the other hand, LILI-OM, LIO-SAM, and LINS all have separate odometry (including feature extraction, and rough pose estimation) and mapping (such as back-end fusion in LILI-OM [163], incremental smoothing and mapping in LIO-SAM [169] and Map-refining in LINS [38]), whose average processing time per LiDAR scan are referred to as "Odo." and "Map." respectively in Table 3.8. The two processing time is summed up to compare with FAST-LIO2.

From Table 3.8, we can see that the FAST-LIO2 consumes considerably less time than other LIO methods, being ×8 faster than LILI-OM, ×10 faster than LIO-SAM, and ×6 faster than LINS. Even if only considering the processing time for odometry of other methods, FAST-LIO2 is still faster in most sequences except for four. The overall processing time of fast-LIO2, including both odometry and mapping, is almost the same as the odometry part of LIO-SAM, ×3 faster than LILI-OM and over ×2

Table 3.8: The Benchmark Comparison of Average Processing Time per Scan in Milliseconds

	FAST-LIO2 (2000)	FAST-LIO2 (1000)	FAST-LIO2 (800)	FAST-LIO2 (600)	FAST-LIO2 (Feature)	FAST-LIO2 (ARM)	LILI-OM		LIO-SAM		LINS	
	Total	Total	Total	Total	Total	Total	Odo.	Map.	Odo.	Map.	Odo.	Map.
lili_6	13.15	**12.56**	13.22	15.92	15.35	45.58	68.95	58.46	—	—	—	—
lili_7	**16.93**	17.61	20.39	19.72	21.13	65.89	40.01	83.71	—	—	—	—
lili_8	**14.73**	15.31	17.73	17.15	18.37	57.29	61.80	79.11	—	—	—	—
utbm_8	21.72	22.05	21.39	**20.82**	21.16	100.00	65.29	84.76	—	—	37.44	153.92
utbm_9	28.26	25.44	21.41	21.35	**17.46**	91.05	68.94	97.90	—	—	38.82	154.06
utbm_10	23.90	22.48	23.09	20.74	**15.30**	94.62	66.10	97.29	—	—	33.61	166.12
ulhk_4	20.86	20.14	19.96	**20.04**	29.35	91.12	52.40	74.80	39.50	95.29	34.72	93.70
ulhk_5	24.10	23.90	23.96	**23.75**	28.70	68.04	53.56	47.68	25.68	127.63	28.01	99.13
ulhk_6	30.52	31.56	30.15	29.25	31.94	92.38	64.46	70.43	**15.16**	164.36	41.54	199.96
nclt_4	15.65	15.72	15.79	15.75	19.98	69.09	62.49	98.46	**13.38**	184.03	46.43	188.40
nclt_5	16.56	16.60	16.61	16.58	**13.54**	68.95	67.64	83.34	19.09	184.46	47.83	198.88
nclt_6	15.92	15.84	15.83	15.68	**14.72**	66.64	76.10	133.25	×	×	54.48	195.31
nclt_7	16.79	16.87	16.82	16.63	**15.16**	70.24	67.65	81.69	29.50	211.18	56.94	197.71
nclt_8	14.29	14.25	14.32	14.14	**7.94**	57.03	53.54	57.54	16.30	163.09	53.53	144.95
nclt_9	13.73	13.65	13.60	13.64	**10.30**	54.82	42.84	68.86	12.79	118.35	46.12	149.45
nclt_10	21.85	21.79	21.78	21.61	**20.62**	89.65	82.92	130.96	23.13	324.62	83.12	252.68
liosam_1	16.95	14.77	14.65	16.19	15.93	60.60	48.45	84.28	**13.47**	135.39	24.13	179.44
liosam_2	**11.11**	11.47	11.52	11.19	19.68	45.27	42.58	99.01	13.09	154.69	20.71	160.66
liosam_3	19.38	16.64	12.00	13.01	12.37	44.26	38.42	64.02	**11.32**	124.35	40.47	117.25

faster than LINS. Comparing the different variants of FAST-LIO2, the processing time for different map sizes are very similar, meaning that the mapping and kNN search with our ikd-Tree is insensitive to map size. Furthermore, the feature-based variant and direct method FAST-LIO2 have roughly similar processing times. Although feature extraction takes additional processing time to extract the feature points, it leads to much fewer points (hence less time) for the subsequent kNN search and state estimation. On the other hand, the direct method saves the feature extraction time for points registration. Allowed by the superior computation efficiency of FAST-LIO2, we further implemented it with the default map size (1000 m, see 3.6.3) on the Khadas VIM3 (ARM) embedded computer. The run time results show that FAST-LIO2 can also achieve 10 Hz real-time performance that has not been demonstrated on an ARM-based platform by any prior work.

3.7 Real-world Experiments

3.7.1 Platforms

Figure 3.5: Three different platforms: (a) 280 *mm* wheelbase small scale quadrotor UAV carrying a forward-looking Livox Avia LiDAR, (b) handheld platforms, (c) 750 mm wheelbase quadrotor UAV carrying a down-facing Livox Avia LiDAR. All three platforms carry the same DJI Manifold-2C onboard computer. The video of real-world experiments is available at `https://youtu.be/2OvjGnxszf8`.

Besides the benchmark evaluation where the datasets are mainly collected on the ground, we also test our FAST-LIO2 in a variety of challenging data collected by other platforms (see Fig. 3.5), including a 280 mm wheelbase quadrotor for the application of UAV navigation, a handheld platform for the application of mobile mapping, and a GPS-navigated 750 *mm* wheelbase quadrotor UAV for the application of aerial mapping. The 280 *mm* wheelbase quadrotor is used for indoor aggressive flight test, see section 3.7.2.2, so that the LiDAR is installed face-forward. The 750 mm wheelbase quadrotor

UAV, developed by Ambit-Geospatial company[8], is used for the aerial scanning, see section 3.7.3, so that the LiDAR is facing down to the ground. In all platforms, we use a solid-state 3D LiDAR Livox Avia[9] which has a built-in IMU (model BMI088), a 70.4° (Horizontal) × 77.2° (Vertical) circular FoV, and an unconventional non-repetitive scan pattern that is different from the Livox Horizon or Velodyne LiDARs used previously in Section 3.6. Since FAST-LIO2 does not extract features, it is naturally adaptable to this new LiDAR. In all the following experiments, FAST-LIO2 uses the default configurations (i.e., direct method with map size 1000 m). Unless stated otherwise, the scan rate is set at 100 Hz, and the computation platform is the DJI manifold 2-C used in the previous section.

3.7.2 Private Dataset

3.7.2.1 Detail Evaluation of Processing Time

In order to validate the real-time performance of FAST-LIO2, we use the handheld platform to collect a sequence at 100 Hz scan rate in a large-scale outdoor-indoor hybrid scene. The sensor returns to the starting position after traveling around 650 m. It should be noted that the LILI-OM also supports solid-state LiDAR, but it fails in this data since its feature extraction module produces too few features at the 100 Hz scan rate. The map built by FAST-LIO2 in real-time is shown in Fig. 3.6, which shows small drift (i.e., 0.14 m) and good agreement with satellite maps.

For the computation efficiency, we compare FAST-LIO2 with its predecessor FAST-LIO [130] on the Intel (Manifold 2-C) computer. For FAST-LIO2, we additionally test on the ARM (Khadas VIM3) onboard computer. The difference between these two methods is that FAST-LIO is a feature-based method, and it retrieves map points in the current FoV to build a new static k-d tree for kNN search at every step. The detailed time consumption of individual components for processing a scan is shown in Table 3.9. The preprocessing refers to data reception and formatting, which are identical for FAST-LIO and FAST-LIO2 and are below 0.1 ms. The feature extraction of FAST-LIO is 0.9 ms per scan, which is saved by FAST-LIO2. The feature extraction leads to fewer point numbers than FAST-LIO2 (447 versus 756), hence less time spent in state estimation

[8] http://www.ambit-geospatial.com.hk
[9] https://www.livoxtech.com/de/avia

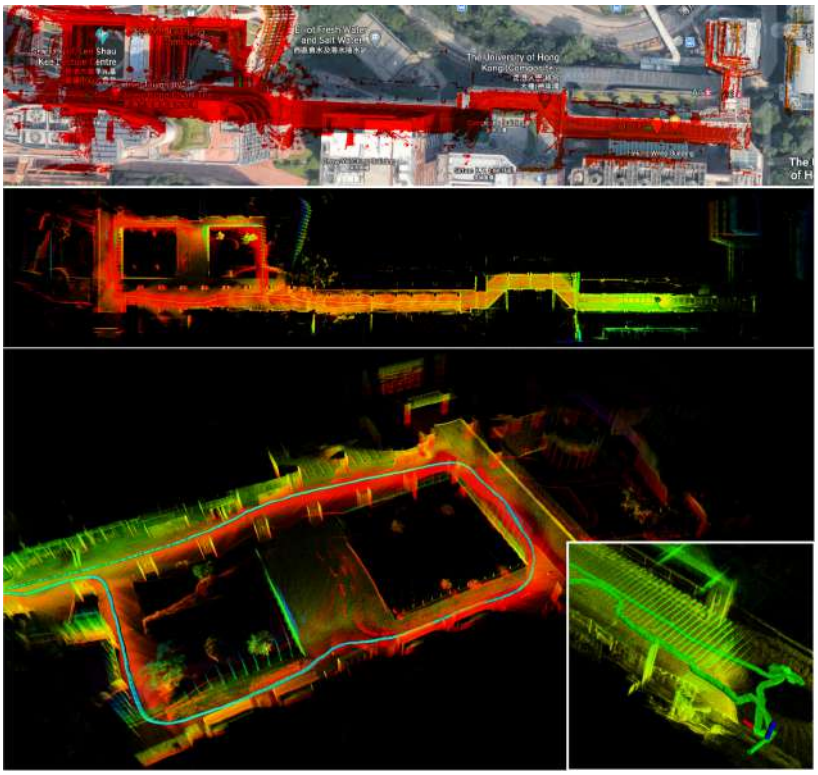


Figure 3.6: Large-scale scene experiment. The handheld platform is used to collect a sequence at 100 Hz scan rate in a large-scale outdoor-indoor hybrid scene (the Centennial campus of the University of Hong Kong).

(0.99 ms versus 1.66 ms). As a result, the overall odometry time of the two methods is nevertheless very close (1.92 ms for FAST-LIO versus 1.69 ms for FAST-LIO2). The difference between these two methods becomes drastic when looking at the mapping module, which includes map points retrieve and k-d tree building for FAST-LIO, and point insertion, box-wise delete due to map move and tree rebalancing for FAST-LIO2. As can be seen, the averaging mapping time per scan for FAST-LIO exceeds 10 ms hence cannot be processed in real-time for this large scene. On the other hand, the mapping time for FAST-LIO2 is well below the sampling period. The overall time for FAST-LIO2 when processing 756 points per scan, including both odometry and mapping, is only 1.82 ms for the Intel processor and 5.23 ms for the ARM processor.

Table 3.9: Mean Time Consumption in Miliseconds by Individual Components when Processing A LiDAR Scan

	FAST-LIO	FAST-LIO2	
	Intel	Intel	ARM
Preprocessing	0.03 ms	0.03 ms	0.05 ms
Feature extraction	0.90 ms	0 ms	0 ms
State estimation	0.99 ms	1.66 ms	4.75 ms
Mapping	13.81 ms	0.13 ms	0.43 ms
Total	15.83 ms	1.82 ms	5.23 ms
Num. of points used	447	756	756
Num. of threads	4	4	2

The time consumption and the number of map points at each scan are shown in Fig. 3.7. As can be seen, the processing time for FAST-LIO2 running on the ARM processor occasionally exceeds the sampling period 10 ms, but this occurred very few and the average processing time is well below the sampling period. The occasional timeout usually does not affect a subsequent controller since the IMU propagated state estimate could be used during this short period. On the Intel processor, the processing time for FAST-LIO2 is always below the sampling period. On the other hand, the processing time for FAST-LIO quickly grows above the sampling period due to the growing number of map points. Notice that the considerably reduced processing time for FAST-LIO2 is achieved even at a much higher number of map points. Since FAST-LIO only retains map points within its current FoV, the number could drop if the LiDAR faces a new area containing few previously sampled map points. Even with fewer map points, the processing time for FAST-LIO is still much higher, as analyzed above. Moreover, since

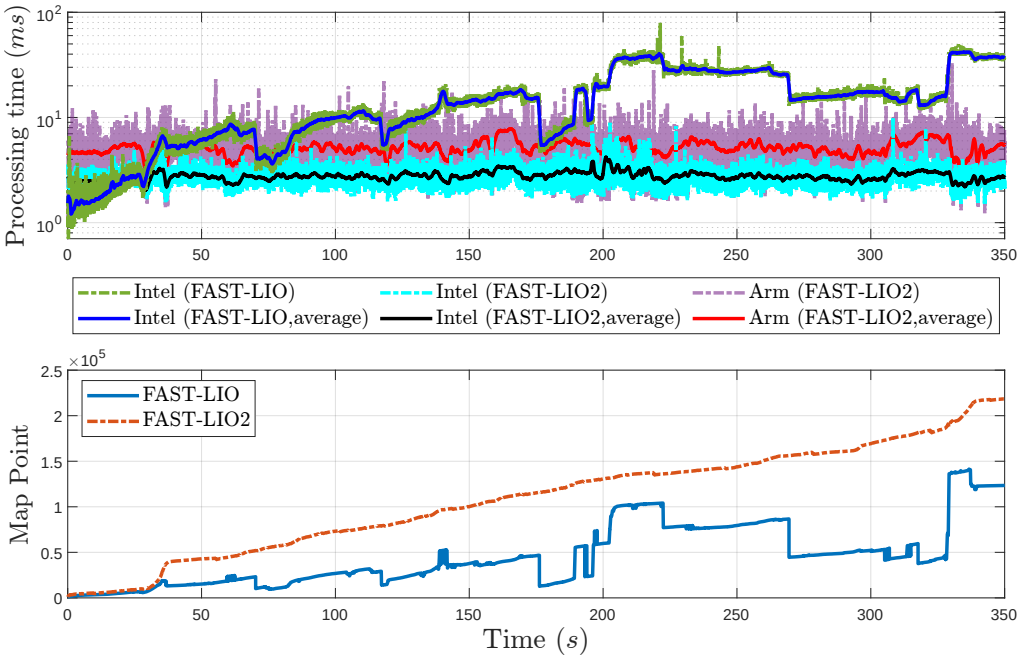


Figure 3.7: The processing time for each LiDAR scan of FAST-LIO and FAST-LIO2.

FAST-LIO builds a new k-d tree at every step, the building time has a time complexity $O(n \log n)$ [92] where n is the number of map points in the current FoV. This is why the processing time for FAST-LIO is almost linearly correlated to the map size. In contrast, the incremental updates of our ikd-Tree has a time complexity of $O(\log n)$, leading to a much slower increment in processing time over map size.

3.7.2.2 Aggressive UAV Flight Experiment

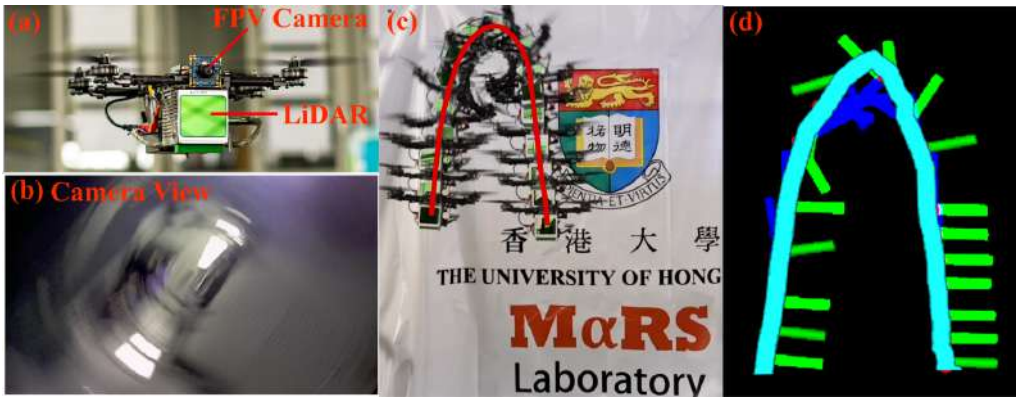


Figure 3.8: The flip experiment. (a) the small scale UAV; (b) the onboard camera showing first person view (FPV) images during the flip; (c) the third person view images of the UAV during the flip; (d) the estimated UAV pose with FAST-LIO2.

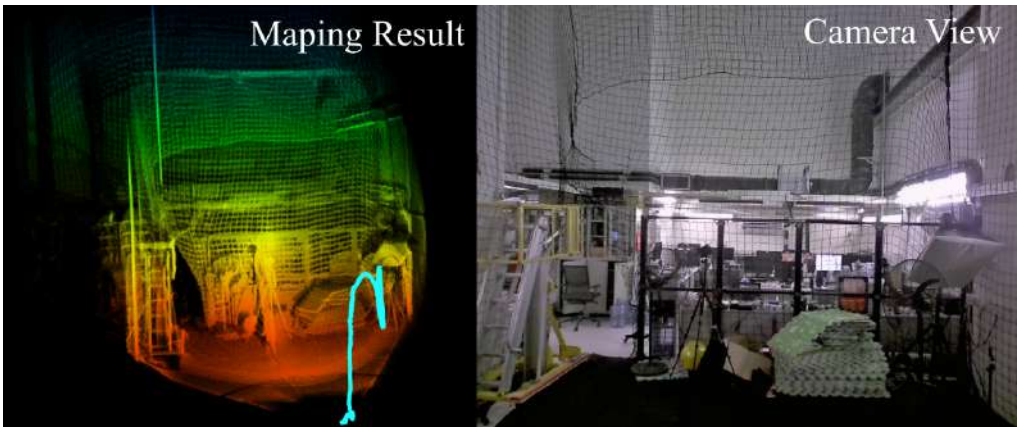


Figure 3.9: The actual environment and the 3D map built by FAST-LIO2 during the flip.

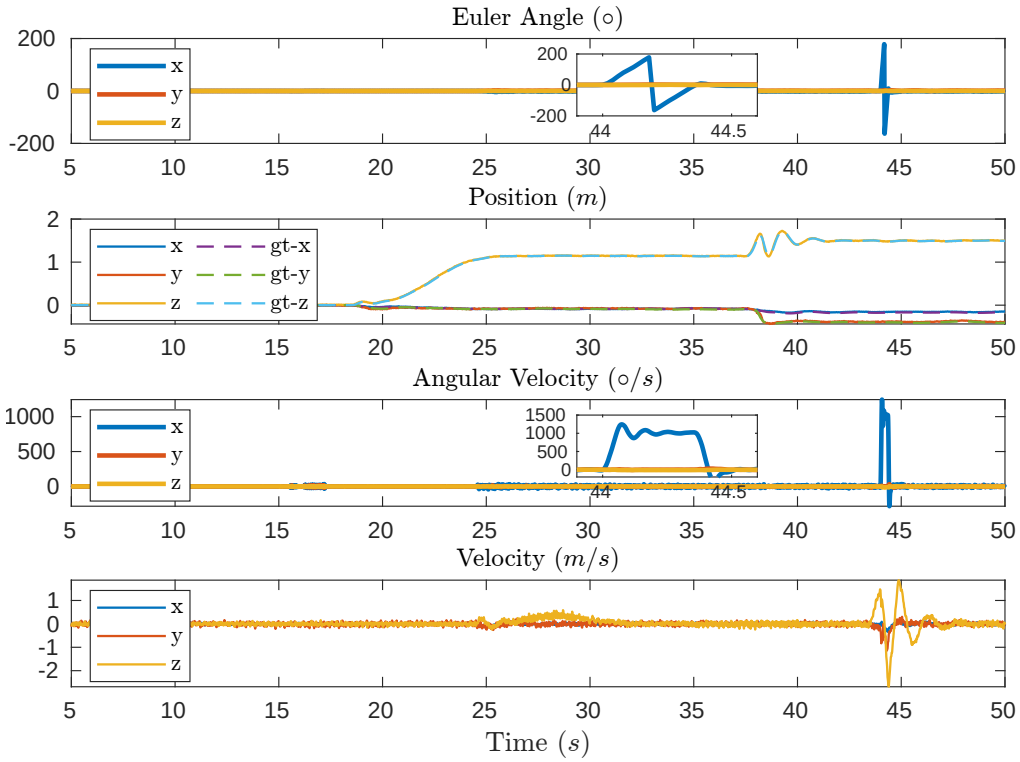


Figure 3.10: The attitude, position, angular velocity and linear velocity in the UAV flip experiment. The notation "gt" stands for the position ground truth collected by a VICON motion capture system.

In order to show the application of FAST-LIO2 in mobile robotic platforms, we deploy a small-scale quadrotor UAV carrying the Livox AVIA LiDAR sensor and conduct an aggressive flip experiment as shown in Fig. 3.8. In this experiment, the UAV first takes off from the ground and hovers at the height of 1.2 m for a while, then it performs a quick flip, after which it returns to the hover flight under the control of an on-manifold model predictive controller [187] that takes state feedback from the FAST-LIO2. The pose estimated by FAST-LIO2 is shown in Fig. 3.8 (d), which agrees well with the actual UAV pose. The real-time mapping (the point clouds are accumulated from the beginning, including all the scans during the flip) of the environment is shown in Fig. 3.9. In addition, Fig. 3.10 shows the position, attitude, angular velocity, and linear velocity during the experiments. The average and maximum angular velocity during the flip reaches 1023 deg/s and 1242 deg/s, respectively (from 44 s to 44.5 s). The RMSE error during the flight (from 18.0 s to 50.0 s) is 0.0186 m when compared to the ground-truth trajectory measured by a VICON motion capture system. FAST-LIO2 takes only 2.21 ms on average per scan, which suffices the real-time requirement of controllers. By providing high-accuracy odometry and a high-resolution 3D map of the environment at 100 Hz, FAST-LIO2 is very suitable for a robots' real-time control and obstacle avoidance. For example, our prior work [188] demonstrated the application of FAST-LIO2 on an autonomous UAV avoiding dynamic small objects (down to 9 mm) in complex indoor and outdoor environments.

3.7.2.3 Fast Motion Handheld Experiment

Here we test FAST-LIO2 in a challenging fast motion with large velocity and angular velocity. The sensor is held on hands while rushing back and forth on a footbridge (see Fig. 3.11). Fig. 3.12 shows the attitude, position, angular velocity, linear velocity and extrinsic estimates in the fast motion handheld experiments. It is seen that the maximum velocity reaches 7 m/s and angular velocity varies around ± 100 deg/s. The initial states of rotational and translation extrinsic are set to (5°, 5°, 5°) and (0, 0, 0). It can be seen that both the rotational and translation extrinsic converge close to the ground truth with small differences possibly due to the manufacturing imperfections. In order to show the performance of FAST-LIO2, the experiment starts and ends at the same point. The end-to-end error in this experiment is less than 0.06 m (see Fig. 3.12)

while the total trajectory length is 81 m.

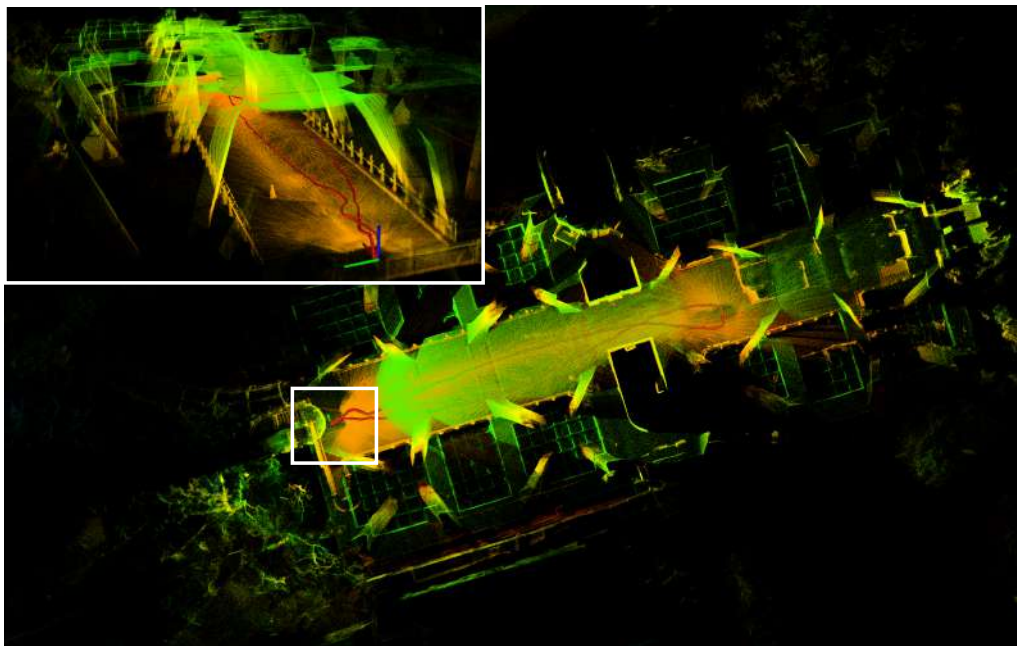


Figure 3.11: The mapping results of FAST-LIO2 in the fast motion handheld experiment.

3.7.3 Outdoor Aerial Experiment

One important application of 3D LiDARs is airborne mapping. In order to validate FAST-LIO2 for this possible application, an aerial experiment is conducted. A larger UAV carrying our LiDAR sensor is deployed. The UAV is equipped with GPS, IMU, and other flight avionics and can perform automatic waypoints following based on the onboard GPS/IMU navigation. Note that the UAV-equipped GPS and IMU are only used for the UAV navigation, but not for FAST-LIO2, which uses data only from the LiDAR sensor. The LiDAR scan rate is set to 10 Hz in this experiment. A few flights are conducted in several locations in the Hong Kong Wetland Park at Nan Sang Wai, Hong Kong. The real-time mapping results are shown in Fig. 3.13. It is seen that FAST-LIO2 works quite well in these vegetation environments. Many fine structures such as tree crowns, lane marks on the road, and road curbs can be clearly seen. Fig. 3.13 also shows the flight trajectories computed by FAST-LIO2. We have visually compared these trajectories with the trajectories estimated by the UAV onboard GPS/IMU navigation, and they show good agreement. Due to technical difficulties, the GPS trajectories are

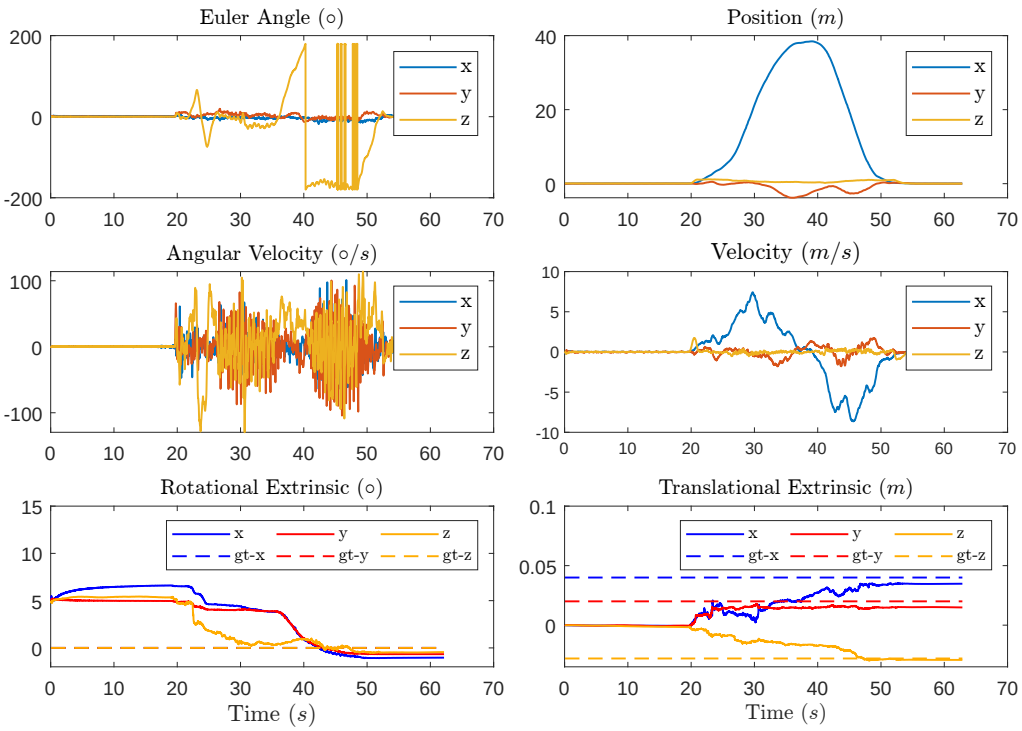


Figure 3.12: The attitude, position, angular velocity, linear velocity, rotational and translational extrinsic in the fast motion handheld experiment. The extrinsic ground truth (denoted as gt in the figure) is obtained from the manufactuer's manual. Notice that the ground truth rotational extrinsic are all zeros, causing the gt-x, gt-y and gt-z to overlap.

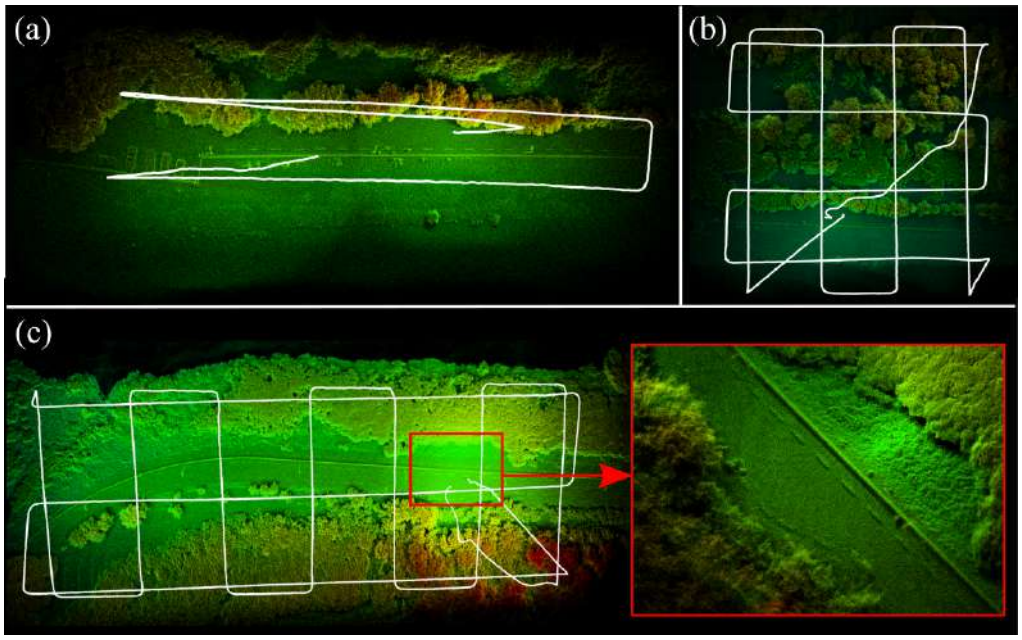


Figure 3.13: Real-time mapping results with FAST-LIO2 for airborne mapping. The data is collected in the Hong Kong Wetland Park by a UAV with a down-facing Livox Avia LiDAR. The flight heights are 30 m (a), 30 m (b) and 30 m (c).

not available here for quantitative evaluation. Finally, the average processing time per scan for these three environments is 19.6 ms, 23.9 ms, and 23.7 ms, respectively. It should be noted that the LILI-OM fails in all these three data sequences because the extracted features are too few when facing the ground.

3.8 Discussion

In this section we discuss the proposed FAST-LIO2 framework in terms of efficiency, accuracy, robustness and possible applications.

3.8.1 Efficiency

Existing ICP-based LiDAR(-inertial) odometry methods, such as LOAM [33], LIO-SAM [169], LILI-OM [163], LINS [38] and our prior work [130], typically build a k-d tree of the point-cloud map from scratch, causing the building time to grow dynamically with the exploration of new areas. To bound this building time, only points in a local area are usually used to build the k-d tree, leading to a constant (and often significant) building time. In contrast, FAST-LIO2 uses an incremental k-d tree structure, ikd-Tree, which dynamically adds new points to the existing tree structure and only re-balances the updated (sub-)tree partially. This incremental update has a considerably lower computation cost than a complete k-d tree building and constitutes the key to the efficiency of FAST-LIO2. The resultant system is able to run at a very high odometry and mapping rate (e.g., up to 100 Hz) on computationally-constrained platforms (UAV onboard computer and ARM-based processors).

3.8.2 Accuracy

The accuracy of FAST-LIO2 benefits from twofold: direct registration of the raw LiDAR points (as opposed to feature points) and the use of larger local map (e.g., $1km$) in scan registration, both are enabled by our efficient map representation, ikd-Tree. The use of raw points enables to exploit more subtle features in the environments while a larger local map establishes more geometrical constraints for registering a new scan. Since LiDARs points have extremely high temporal resolution, a temporal downsampling (e.g., one out of four LiDAR points) can effectively lower the computation load without

affecting the accuracy. We also notice that further increasing the local map does not persistently improve the accuracy due to the possible false point matches caused by odometry drift.

The use of the center-most point in a downsampling cube leads to a more evenly distributed point map which also contributes to the accuracy of FAST-LIO2. We also considered other alternative choices (e.g., mean or median point) and found they usually have lower accuracy. The primary reason is that, the odometry drift will cause the new points, when added to the map, to bias the mean or median point in a downsampling cube. The biased map points will in turn further bias the plane estimation and hence the subsequent state estimate. Another drawback of using mean or median points is that they cause constant point update on the ikd-Tree, which leads to frequent tree re-build and hence a higher computation cost.

Another possible way to improve the system accuracy is splitting a scan into multiple smaller sub-scans to be processed sequentially. The scan split will reduce the propagation time of IMU measurements, which in turn could improve the accuracy of the forward propagation for state prediction and the backward propagation for motion distortion compensation. However, fewer measurements in a sub-scan also make the state estimate more sensitive to false point matches. In FAST-LIO2, we choose 100 Hz update rate.

3.8.3 Robustness

In Section 3.7, the FAST-LIO2 was proven to work stably in high angular speed (over 1000 deg/s) and high frame rate (100 Hz). Such robustness comes from two reasons: firstly, the direct raw point registration uses more LiDAR measurements in high frame rate and aggressive motion compared to the feature based methods; secondly, the timely updated (at the frame rate) map allows each new scan to be registered to map points of the most recent scans, which share large FoVs with the new scan. This allows FAST-LIO2 to reliably track even very fast robots' motions.

3.8.4 Applications

FAST-LIO2 is a computationally efficient, robust, and accurate odometry suitable for robots navigation and control. The dense point map it builds in real-time can be

used for collision check in trajectory generation even in the presence of small dynamic obstacles [188], and the high-rate odometry provides low-latency feedback to controllers [187]. FAST-LIO2 could also be used in small-scale mapping applications where the odometry drift is not significant. For large-scale mapping, FAST-LIO2 can be used with additional back-end optimization such as sliding window bundle adjustment [46] or incremental smoothing and mapping techniques [170] to achieve better long-term accuracy. A loop-closure module also can be easily integrated with FAST-LIO2.

FAST-LIO2 cannot work in completely degenerated environments where no geometrical structures exists in the LiDAR FoV (hence no or very few LiDAR points). Examples include facing the LiDAR to open sea, the sky, the ground, a single large wall, or in close proximity to objects (e.g., within 1 m) causing no points measurements. In these scenarios, FAST-LIO2 can be augmented by other sensors such as GPS and cameras [189].

3.9 Conclusion

This paper proposed FAST-LIO2, a direct and robust LIO framework significantly faster than the current state-of-the-art LIO algorithms while achieving highly competitive or better accuracy in various datasets. The gain in speed is due to removing the feature extraction module and the highly efficient mapping, which is enabled by ikd-Tree. A large amount of experiments in open datasets shows that the proposed ikd-Tree can achieve the best overall performance among the state-of-the-art data structure for kNN search in LiDAR odometry. As a result of the mapping efficiency, the accuracy and the robustness in fast motion and sparse scenes are also increased by utilizing more points in the odometry. A further benefit of FAST-LIO2 is that it is naturally adaptable to different LiDARs due to the removal of feature extraction, which has to be carefully designed for different LiDARs according to their respective scanning pattern and density.

Chapter 4

Occupancy Grid Mapping without Ray-Casting for High-resolution LiDAR Sensors

With FAST-LIO2 presented in the previous chapter, we have successfully resolved the problem of self-localization for mobile robots in unknown environments. This chapter goes on to address another challenge for autonomous navigation in unknown environments for robotics, which is to reason about the unknown and known regions of the environment, commonly known as occupancy mapping. This chapter presents an efficient occupancy mapping framework for high-resolution LiDAR sensors, termed D-Map. The framework introduces three main novelties to address the computational efficiency challenges of occupancy mapping. Firstly, we use a depth image to determine the occupancy state of regions instead of the traditional ray-casting method. Secondly, we introduce an efficient on-tree update strategy on a tree-based map structure. These two techniques avoid redundant visits to small cells, significantly reducing the number of cells to be updated. Thirdly, we remove known cells from the map at each update by leveraging the low false alarm rate of LiDAR sensors. This approach enhances our framework's update efficiency by shrinking the map size. As this approach updates the map in a decremental manner by reducing the size, we name our framework D-Map. To support

our design, we provide theoretical analyses of the accuracy of the depth image projection and time complexity of occupancy updates. Furthermore, we conduct extensive benchmark experiments on various LiDAR sensors in both public and private datasets. Our framework demonstrates superior efficiency in comparison with other state-of-the-art methods while maintaining comparable mapping accuracy and high memory efficiency. We demonstrate two real-world applications of D-Map for real-time occupancy mapping on a handheld device and an aerial platform carrying a high-resolution LiDAR. In addition, we open-source the implementation of D-Map on GitHub to benefit society: github.com/hku-mars/D-Map.

4.1 Introduction

In recent years, advancements in light detection and ranging (LiDAR) sensors have led to the commercialization of lightweight, low cost, and high accuracy 3D LiDARs, raising tremendous popularity in various applications such as robotics [135, 190–192], autonomous driving [160, 193, 194], 3D reconstruction [33, 78, 195], etc. Over the past decade, a clear trend has emerged to develop 3D LiDARs with a smaller size, longer detection range, and higher resolution, approaching image-level quality [22]. This trend has not only opened up new possibilities for deploying LiDARs in diverse applications but has also necessitated the development of new techniques to exploit the potential of LiDAR-based applications.

Occupancy grid mapping provides an efficient means for autonomous systems to navigate in unknown environments, which is crucial in a variety of applications such as obstacle avoidance [133, 188], path planning [196, 197], and autonomous exploration [3, 12, 198]. When adapted to LiDARs with increasing performance, occupancy grid mapping is faced with additional challenges in efficiency, especially in the two core steps: ray-casting and occupancy state updates.

Ray-casting is challenging because LiDARs provide increasingly dense depth measurements (e.g., over one million points per second) with long ranges (e.g., over 300 m). Therefore, the number of rays to be processed increases with the number of points. Secondly, the number of cells traversed by a single ray is affected by the sensor's detection range, with LiDARs traversing dozens of times more cells than depth cameras with

smaller detection range (e.g., less than 5 m). Consequently, the tremendous amount of cells results in a costly computation load. Finally, the dense LiDAR measurements can lead to redundant visits to the same cells in ray-casting, which can consume unnecessary computation resources in the subsequent occupancy state updates.

The second challenge in LiDAR-based occupancy mapping arises from the need for high-resolution occupancy maps to fully exploit the high accuracy of LiDAR sensors. This challenge involves the selection of map structures to achieve high-resolution occupancy mapping where a trade-off between computational and memory efficiency has to be balanced. The two commonly used map structures are grid-based [199] and tree-based [96]. Grid-based maps offer highly efficient updates in $\mathcal{O}(1)$ time complexity but suffer from extensive memory consumption when applied in large-scale environments or with high resolutions. Tree-based maps are more memory-efficient than grid-based maps, but the trade-off is the higher computational complexity when updating the tree. The update efficiency of a tree-based map structure is directly influenced by the tree height, which is determined by both the mapping environment and map resolution.

In addition to the challenges mentioned above, some essential features of LiDAR sensors have not yet been fully utilized in existing occupancy mapping approaches, such as the low false alarm rate which provides high confidence in identifying free and occupied space. The existing occupancy mapping approaches commonly utilize occupancy probabilities to handle the false detection of depth measurements. Nevertheless, in LiDAR-based occupancy mapping, the low false alarm rate (e.g., three out of a million points [200]) offers an opportunity to enhance efficiency by directly eliminating known space from the map without requiring any probability updates.

In this work, we present a novel mapping framework that effectively addresses the aforementioned issues in LiDAR-based occupancy mapping. Our contributions can be summarized as follows:

- We propose an occupancy state determination method based on depth image projection to alleviate the computation load in the traditional ray-casting technique. This projection-based approach enables occupancy state determination of cells at any size, allowing subsequent efficient updates in large-scale environments.

- We present a novel on-tree update strategy for updating occupancy states based on a hybrid map structure, providing a superior balance between computation and memory efficiency. The hybrid map structure stores unknown space on an octree, which enables a memory-efficient representation for the large unknown space, while the occupied space is stored on a hashing grid map. In terms of efficiency, the proposed strategy allows occupancy state determination of large cells on an octree, thereby avoiding unnecessary updates on small cells and increasing efficiency.
- We leverage the low false alarm rate of LiDAR measurements to directly remove cells with determined states (i.e., occupied or free) at each update. This approach renders our map structure a decremental property, for which we term our framework as D-Map, providing higher computational efficiency and less memory usage.
- We conduct an in-depth analysis of the accuracy of the proposed occupancy state determination method and the time complexity of updates and queries in D-Map. Specifically, we derive an analytical function to quantify the accuracy loss relative to depth image resolution. The time complexity analysis of updates on D-Map provides theoretical support to our superior performance over state-of-the-art methods that rely on ray-casting.
- We make the implementation of D-Map available on Github: `github.com/hku-mars/D-Map` to promote the reproducibility and further development of our work.

The remainder of this chapter is organized as follows. Section 4.2 provides an overview of related works in the field of occupancy mapping. We then present the complete mapping framework and the details of each key component in Sections 4.3, 4.4, and 4.5, respectively. Section 4.7 presents the benchmark experiments and ablation studies conducted on open datasets. In Section 4.8, we demonstrate two real-world experiments, followed by a discussion in Section 4.10. Finally, we conclude this article in Section 4.11.

4.2 Related Works

In this section, we review the previous research on occupancy mapping and discuss their update methods.

4.2.1 Occupancy Mapping Approaches

We categorize the occupancy mapping approaches into two classes: continuous maps and discrete maps, based on their assumptions of modeling the environments.

Continuous maps assume an implicit correlation of locality in space and model environments through continuous occupancy distribution. In the early years, Thrun *et al.* introduced the concept of learning inverse sensor models to generate local occupancy maps by means of sampling [95]. In more recent years, O'Callaghan *et al.* proposed using Gaussian Process regression to learn a continuous representation of 2-dimensional environments, which has since been extended to 3-dimensional environments using Gaussian mixture models or sparse Gaussian processes to reduce computational load during training [101, 104, 201]. However, the time complexity of these methods is often prohibitive for real-time applications, with a computational complexity of $\mathcal{O}(N^3)$ in the number of depth measurements N. Therefore, considerable efforts have been devoted to improving the update efficiency of these methods [106, 109, 110, 114]. Unlike previous approaches maintaining dense occupancy states, Duong *et al.* developed a sparse Bayesian formulation for occupancy mapping using a sparse set of relevance vectors [202]. While the aforementioned methods can be applied to LiDAR sensors, there is a substantial body of research focused on learning-based occupancy mapping using image sequences, including Occupancy Networks [203], Neural Radiance Fields (NeRF) [204], DeepSDF [205], Semantic Mapping [17, 206], NICE-SLAM [207], among others.

As opposed to continuous maps, occupancy grid mapping approaches assume independent occupancy at different locations, resulting in a discrete representation of environments using grids. Historically, a uniform grid map structure was imposed to discretize the environments and represents the occupancy probabilities independently [199, 208, 209]. However, this approach is impractical for generating high-resolution maps or working in large-scale environments for its significant memory consumption. To address this limitation, tree-based map structures such as quadtree [210] and octree [93] were applied in occupancy grid mapping approaches with great effect. These structures recursively divide space into equal subspaces for updates and merge subspaces with equivalent states for a more compact representation [96, 211]. However, the updates on the tree-based maps are much more expensive. A voxel hashing technique [117] has recently been

adapted to occupancy mapping [13, 14]. Though alleviating the memory consumption in uniform grids, the memory consumption remains prohibitive to high-resolution maps and large-scale environments. The use of discrete representations of environments has also been applied in Euclidean Signed Distance Field (ESDF), an alternative approach to occupancy mapping [99, 100]. However, the construction of ESDF heavily relies on ray-casting and encounters similar challenges to other existing occupancy mapping techniques.

While the continuous mapping approaches provide dense occupancy maps that can reason about measured and unmeasured areas, concerns remain about the reliability of the inferred regions, particularly in safety-critical robotics applications. Besides, continuous mapping approaches face significant challenges in real-time ability due to the complicated training and querying process, as integration over non-trivial space is required to update and query on a continuous map which necessitates timely numerical evaluation. Compared with the continuous mapping approaches methods, occupancy grid mapping is preferable in robotic applications with limited computation resources because of its higher update efficiency resulting from less complex map representation. Additionally, occupancy grid mapping is regarded as a more reliable technique, as it updates the occupancy map solely based on actual depth measurements without any inferences. Our D-Map is one kind of occupancy grid mapping approach. It achieves higher computational efficiency than existing grid-based approaches while maintaining comparable memory consumption with tree-based methods.

4.2.2 Update Methods

Regardless of different map representations, various research has been conducted to improve the efficiency of map updating, most of which focuses on alleviating the computation load of ray-casting. Wurm *et al.* present a hierarchical structure of octrees with multiple resolutions, allowing not complete but adequate resolution in different sub-maps depending on objects to reduce the computation load in ray-casting [120]. Similarly, [121] propose an adaptive resolution mapping method based on statistical measurements. However, the resultant maps generated from this method are affected by the parameters of adaptive resolution, which may differ from the original results. In Octomap [96], a batch-based method is utilized to avoid redundant cells resulting from

ray-casting, which effectively reduces the effort in subsequent tree updates. Besides, a clamping policy is used in Octomap, initially proposed to handle dynamic environments [212], allowing the acceleration of map updates by pruning the octree. Duberg *et al.* proposed two possible approaches to simplify the ray-casting process, including downsampling point clouds to map resolution and ray marching with adaptive steps. However, both of these approaches introduce accuracy loss and require handcraft parameters [122]. In contrast, motivated by the idea of coherent rays [213], Kwon *et al.* combine rays traversing equivalent cells into a super ray to reduce the number of rays in ray-casting [214] while maintaining full accuracy. Besides super rays, the number of cells to be traversed is reduced by culling regions. However, these two techniques provide less benefit at high resolution due to the ineffectiveness of merging rays into a super ray and the inefficiency of culling region construction.

In comparison with the aforementioned update approaches, D-Map discards the timely ray-casting for occupancy mapping. Alternatively, we design an occupancy state determination method based on depth image projection. We develop an on-tree update strategy for efficient map updates. This strategy determines the occupancy states of cells at various resolutions by projecting cells to the depth image, which is more reliable compared to the adaptive resolution mapping approach [121] that relies on handcraft parameters. Moreover, unlike existing occupancy mapping methods, D-Map does not require occupancy probability updates. Instead, we assume the environment is static and directly remove known space (i.e., free and occupied) from the tree map by leveraging the low false alarm rate of LiDAR sensors. This approach results in a decreasing size of the map during the update process, thus further enhancing the efficiency of D-Map.

4.3 Overview

Figure 4.1 presents an overview of the proposed D-Map framework. The map structure delineated in the orange block is explained in Section 4.5.1. It consists of two parts: the occupied map and the unknown map. The green block represents the pipeline of the occupancy update strategy. At each update, a depth image is rasterized from the incoming point clouds at the sensor pose (see Section 4.4.1). Subsequently, a 2-D segment tree is constructed on the depth image to enable efficient occupancy state determination

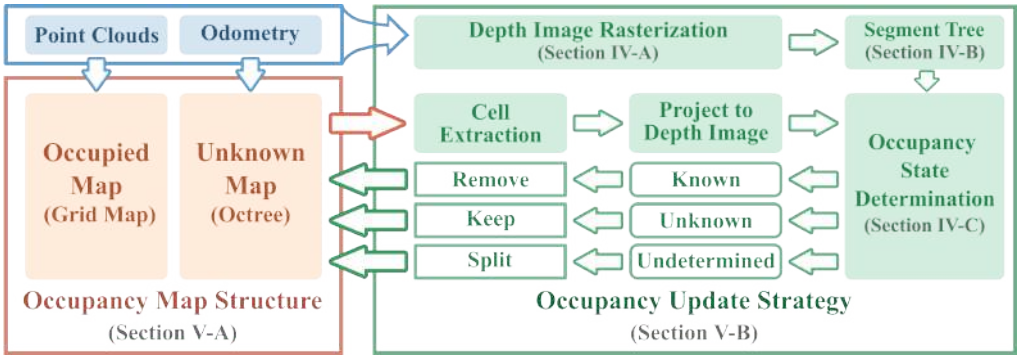


Figure 4.1: The framework overview of D-Map. The blue block shows the input to D-Map, including the point clouds and the corresponding sensor odometry. The orange block is the occupancy map structure of D-Map, which is composed of a hashing grid map for maintaining occupied space and an octree for maintaining unknown space. The occupancy update strategy is presented in the green block, which extracts the cells inside the sensing area on the octree and conducts operations depending on the occupancy state determination method using a depth image.

(see Section 4.4.2 and Section 4.4.3). The entire procedure to update the unknown map is described in Section 4.5.2 and summarized as follows. The cell extraction module retracts the unknown cells on the octree from the largest to the smallest size, projects them to the depth image, and determines their occupancy states. The cells determined as known are directly removed from the map, while the unknown ones remain, and the undetermined ones are split into smaller cells for further occupancy state determination. This coarse-to-find process facilitates updates on large cells directly without queries each small cell. Moreover, the removal of known cells endows our framework with a decremental property, providing high efficiency in both computation and memory.

4.4 Occupancy State Determination on Depth Image

This section describes how to determine the occupancy states on a depth image rasterized from incoming point clouds.

4.4.1 Depth Image Rasterization

In preparation for occupancy state determination, the point cloud captured by a LiDAR sensor is rasterized into a depth image at the current sensor pose. To ensure the accuracy of state determination, the depth image resolution should be sufficiently small so that the projected area of a cell from the map to the depth image is larger than one

pixel. As illustrated in Fig. 4.2, we determine the depth image resolution $\psi_{\texttt{map}}$ relative to the map resolution d and LiDAR's detection range R using the following equation:

$$\psi_{\texttt{map}} = 2\arcsin(\frac{d}{2R}) \approx \frac{d}{R} \tag{4.1}$$

However, high-resolution maps would result in a high-resolution depth image of enormous size, with many empty pixels due to the much smaller number of point clouds than the size of the depth image. To address this issue, we bound the depth image resolution by the LiDAR angular resolution, which is the minimum angle between two laser pulses emitted and received in a rotating manner. Specifically, we define the standard depth image resolution ψ_I as

$$\psi_I = \max\{\psi_{\texttt{map}}, \psi_{\texttt{lidar}}\} \approx \max\{\frac{d}{R}, \psi_{\texttt{lidar}}\} \tag{4.2}$$

where $\psi_{\texttt{lidar}}$ is the angular resolution of the LiDAR. Note that we do not distinguish between the vertical and horizontal angular resolution for simplicity of definition. Moreover, we keep the minimum depth value when neighboring points are projected to the same pixel.

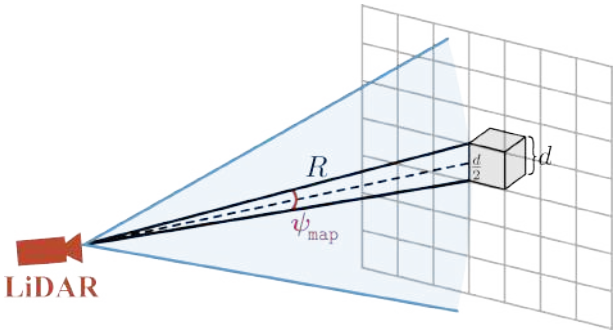


Figure 4.2: This figure illustrates the spatial relationship among the map resolution d, the detection range R, and the depth image resolution $\psi_{\texttt{map}}$.

4.4.2 2-D Segment Tree

To determine the occupancy state of a cell in the map, a two-step process is employed, whereby the cell is first projected onto the depth image, followed by a comparison of the projected depth against the minimum and maximum depth values of the corresponding area. Since the projected area of a cell on the depth image varies with the cell

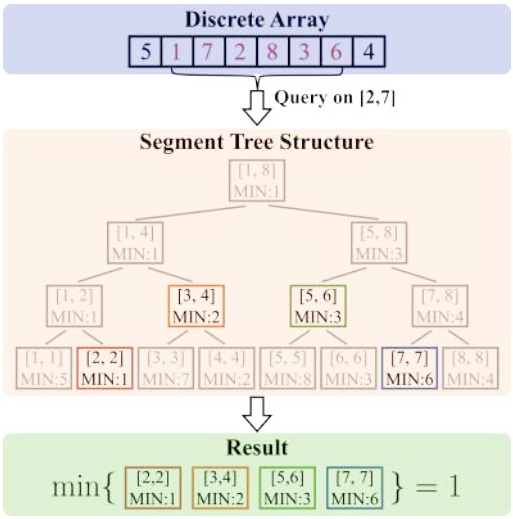


Figure 4.3: This figure illustrates an example of a fast query of the minimum value in the pixel range of $[2, 7]$ on a 1-D segment tree. Starting from the root of the segment tree, the range query searches along the tree recursively until the current node range is completely covered by the queried range, where the minimum value of the node range that has been saved on the node during the tree construction will be returned. In this example, the range $[2, 7]$ leads to four nodes representing the range of $[2, 2]$, $[3, 4]$, $[5, 6]$, and $[7, 7]$, respectively. The minimum value of the range is efficiently obtained from these four nodes instead of counting the six elements in the array.

location, a 2-D segment tree structure is employed to expedite efficient queries of the minimum and maximum values on the depth image, as detailed below.

A segment tree is a perfectly balanced binary tree that efficiently provides range queries by representing a set of intervals [215]. Figure 4.3 describes the process of querying the minimum value by a 1-D segment tree. The segment tree is constructed by recursively splitting the array in half until each node contains a single element. As each node on the segment tree represents an interval of the array, the summarized information of the values in the interval, such as the minimum and maximum, is preprocessed during the construction procedure to accelerate the subsequent query. When querying, the segment tree retrieves a minimal representation of the queried interval using a subset of nodes (colored nodes in Fig. 4.3). The result is obtained by summarizing information from the retrieved nodes with fewer operations than direct queries. The time complexity of query on a 1-D segment tree is $\mathcal{O}(\log N)$ where N is the number of elements in the discrete array, while direct query leads to a time complexity of $\mathcal{O}(N)$.

The approach to extending a 1-Dimensional segment tree to a 2-Dimensional structure involves constructing a "segment tree of segment trees", as proposed in [216]. The

segment tree in the outer layer splits the 2-D array by row, and on each node of the outer segment tree, a 1-D inner segment tree is constructed to maintain the column information on the covered rows. The query on a 2-D segment tree first searches the outer tree for the nodes representing rows covered by the queried region and then traverses the inner segment trees on the corresponding nodes to retrieve the covered columns. Finally, the result over the queried region is summarized from the information stored on retrieved nodes. The time complexity of a 2-D segment tree to query on a 2-D array of size $N \times M$ is $\mathcal{O}(\log N \log M)$, while direct queries lead to a time complexity of $\mathcal{O}(N^2)$.

Given a depth image rasterized from point clouds, we build a 2-D segment tree to maintain the minimum and maximum depth values on each tree node, denoted as `dMin` and `dMax`, respectively. Additionally, we keep track of the number of pixels occupied by point clouds within the covered area of each node, denoted as `dSum`.

4.4.3 Occupancy State Determination

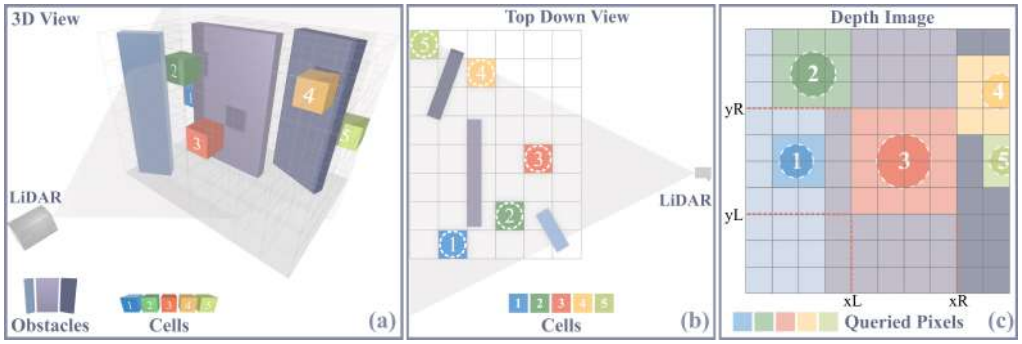


Figure 4.4: This figure demonstrates an example of occupancy state determination on five cells in the map. (a) The 3D view shows the relative position of the five cells with respect to (w.r.t.) the LiDAR and the objects in the environment. (b) The top-down view helps to understand the occlusion between the cells and the objects when seeing from LiDAR. (c) The cells are projected to the depth image by their circumsphere radius, after which the projected areas are queried for the depth values in the depth image, which are finally used to determine the cells' occupancy states.

We introduce the principle of our method for determining the occupancy state of a cell using five cells as an example, depicted in Fig. 4.4 and numbered from 1 to 5. We commence by classifying the cells based on whether they are completely located inside the LiDAR's sensing area. As illustrated in the top-down view in Fig. 4.4(b), Grid 1, Grid 2, and Grid 3 are entirely located within the sensing area, while Grid 4 and Grid 5 only have part of them inside. Among the cells completely inside, Grid 3 is determined

Algorithm 9: Occupancy State Determination

Params: Completeness threshold ε
Input : Grid Center C, Grid Size L, LiDAR Pose T, 2D Segment Tree $\mathcal{T}$
Output: State S

Function DetermineOccupancy(T,C,L,$\mathcal{T}$)
  $r, \theta, \phi \leftarrow$ Projection(T,C);
  BoxMin $= r - L/2$, BoxMax $= r + L/2$;
  xL, xR, yL, yR $\leftarrow$Coverage(r, θ, ϕ, L);
  ProjPixels $= (\text{yR} - \text{yL} + 1)(\text{xR} - \text{xL} + 1)$;
  dMin, dMax, dSum $\leftarrow \mathcal{T}$.query(xL, xR, yL, yR);
  $\alpha =$ dSum/ProjPixels;
  if dSum $== 0$ **then** return(Unknown);
  if $\alpha > \varepsilon$ **then**
    if dMax<BoxMin **then**
      S $\leftarrow$ Unknown;
    else
      if dMin>BoxMax **then**
        S $\leftarrow$ Unknown;
      else
        S $\leftarrow$ Undetermined;
      end
    end
  else
    if dMax<BoxMin **then**
      S $\leftarrow$ Unknown;
    else
      S $\leftarrow$ Undetermined;
    end
  end
  return(S);
End Function

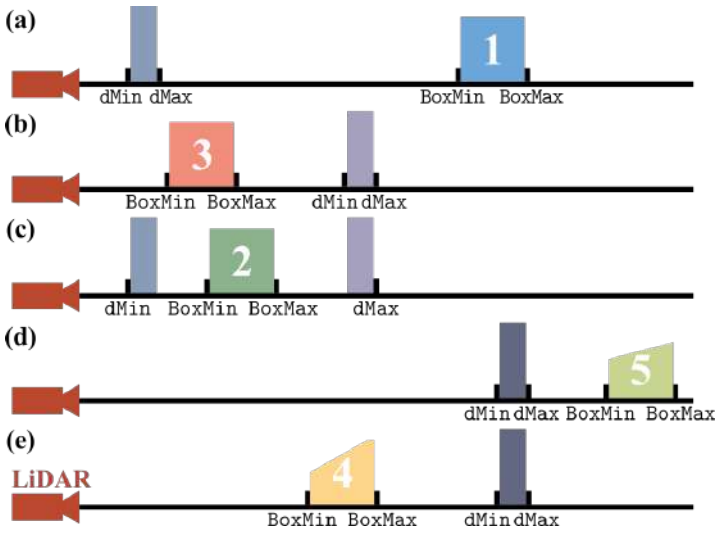


Figure 4.5: The relative position between the cells and objects in the environments, along the one-pixel direction of the depth image, is determined by comparing the depth range of the cell (represented by the minimum depth `BoxMin` and maximum depth `BoxMax`) with the depth range on the depth image (represented by the minimum depth (`dMin` and maximum depth `dMax`). Grids 1-3 completely locate inside the LiDAR's sensing area, while Grids 4-5 are partially inside.

as known since it is situated in front of all objects in the observed environment; Grid 1 is determined as unknown due to its location behind the objects. The occupancy state of Grid 2 remains undetermined since part of it lies in front of the objects while the other part lies behind. Regarding the cells with part of them inside the sensing area, Grid 5 is determined as unknown because it is located behind the objects. Though lying in front of the object, the occupancy state of Grid 4 remains undetermined owing to its location not completely inside.

Following the principles above, D-Map projects the cells that are either inside or intersected with the LiDAR's sensing area onto the current depth image rasterized from recent point clouds which represent the objects in the environment. The relative location between the cells and the objects is determined by comparing their depth values. The procedure of occupancy state determination on the depth image is described in Alg. 9 and explained below.

In consideration of consistency and computational efficiency, we project a cell to the depth image by its inscribed circle, as shown in Fig. 4.4 (b) and (c). The cell center, denoted as `C`, is first transformed into a spherical coordinate as (r, θ, ϕ) (Line 2). The depth range covered by the cell is described as its minimum depth `BoxMin` and maximum

depth `BoxMax` with respect to its center `C` and cell size `L` (Line 3). The boundary of the projected area on the depth image (i.e., queried pixels in Fig. 4.4(c)) is obtained as follows

$$\begin{aligned} \mathtt{x_L} &= \lfloor \frac{\theta - \arcsin(\frac{L}{2r})}{\psi_I} \rfloor, \ \mathtt{x_R} = \lceil \frac{\theta + \arcsin(\frac{L}{2r})}{\psi_I} \rceil \\ \mathtt{y_L} &= \lfloor \frac{\phi - \arcsin(\frac{L}{2r})}{\psi_I} \rfloor, \ \mathtt{y_R} = \lceil \frac{\phi + \arcsin(\frac{L}{2r})}{\psi_I} \rceil \end{aligned} \tag{4.3}$$

Then, we prepare the following information for determining the occupancy states of the cell. Firstly, the number of queried pixels covered by the projected area is counted in `ProjPixels` (Line 5). Secondly, the number of occupied pixels `dSum`, minimum depth `dMin`, and maximum depth `dMax` within the projected area is provided by querying the 2-D segment tree as described in Section 4.4.2 (Line 6). Thirdly, we define the observation completeness α of a cell to determine whether a cell is completely observed by the current depth image, which is calculated by dividing the number of occupied pixels `dSum` over the number of queried pixels `ProjPixels` on the depth image (Line 7). The computed α quantifies if the cell's projected area has been sufficiently observed by points in the current depth image. A large α indicates that a major part of the cell lies inside the sensing area, with most of the pixels in its projected area on the depth image actively occupied by LiDAR points. Only cells with large α have their occupancy state updated.

The occupancy state of a cell is acquired by comparing the depth range of the depth image (represented by the minimum depth `dMin` and maximum depth `dMax`) against the depth range of the cell (represented by minimum depth `BoxMin` and maximum depth `BoxMax`), as shown in Fig. 4.5 and detailed below. We first categorize the cells by comparing the observation completeness α with a completeness threshold ε. For completely observed cells (i.e., $\alpha > \varepsilon$ in Line 9), the occupancy state is determined as unknown if `dMax` of the depth image is smaller than `BoxMin` of the cell (Line 10∼12). This condition indicates that the cell is located behind the object in the environment, as shown in Fig. 4.5(a). When `dMin` of the depth image is greater than `BoxMax` of the cell, the occupancy state of the cell is determined as known (Line 13∼15), indicating that it is located in front of the objects as shown in Fig. 4.5(b). Otherwise, the occupancy state of the cell cannot be determined, as shown in Fig. 4.5(c). For cells that are not completely observed (i.e., $\alpha \leqslant \varepsilon$), the occupancy state is determined as unknown if `dMax` of the depth image is less than `BoxMin` of the cell (Line 20∼22), indicating that the cell lies

behind the objects as shown in Fig.4.5(d). If this condition is not met, the occupancy state of the cell is considered undetermined due to incomplete observation (Line 22∼24), as shown in Fig. 4.5(e).

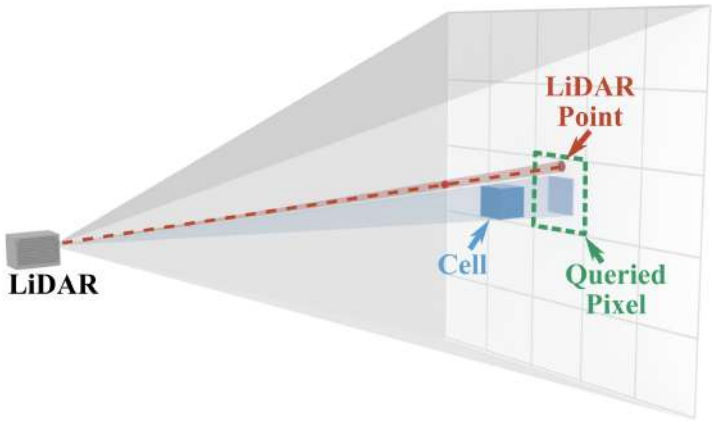


Figure 4.6: A special case when the pixel size is larger than the projected area of a cell. This happens when the depth image resolution is computed from LiDAR's angular resolution.

In some situations, the projected area of a cell may be smaller than the one-pixel area on the depth image, as shown in Fig. 4.6. This occurs when the depth image resolution ψ_I is computed from the LiDAR's angular resolution, as defined by (4.2). These cases are referred to as "single pixel cases". In such cases, recalling that a pixel only saves one LiDAR point with the smallest depth as presented in Section 4.4.1, we should further determine whether the LiDAR point actually traverses the cell by examining if the point lies within the cell's projected area. If the LiDAR point lies outside the projected area, the occupancy state of the cell remains unknown. Otherwise, we use a similar logic to Alg. 9 to determine the occupancy state.

4.4.4 Depth Image Resolution Analysis

The resolution of a depth image is a critical parameter that plays a crucial role in balancing the accuracy and efficiency of occupancy state determination. A low-resolution depth image is more convenient to query but at the cost of increased information loss from the original point clouds. Conversely, a high-resolution depth image provides better accuracy but requires a larger computational effort, leading to reduced efficiency. When considering a depth image $\mathcal{M}_I$ with a standard resolution of ψ_I determined by (4.2), it is possible to relax the resolution to $\psi = \gamma\psi_I$ by a relaxed factor γ to achieve

higher efficiency while sacrificing some accuracy. To evaluate the accuracy loss with the resolution relaxation factor γ, we derive a function $f(\gamma)$ that describes the retained accuracy from the original depth image $\mathcal{M}_I$ as a function of the factor γ.

We assume a LiDAR is employed to scan a large-scale open environment free of obstacles at a static pose. When determining the free space by projecting to the depth image $\mathcal{M}_I$ with the standard resolution of ψ_I, the resultant occupancy map should show complete free space in a 3D spherical domain, the volume of which is denoted as $\boldsymbol{V}_I$. However, when relaxing the depth image resolution to $\psi = \gamma\psi_I$ for a relaxed depth image $\mathcal{M}$, some free space is falsely determined as unknown when projecting to $\mathcal{M}$ due to the single pixel cases explained in Section 4.4.3. We denote the volume of this case as $\boldsymbol{V}$. An illustration of $\boldsymbol{V}_I$ and $\boldsymbol{V}$ is presented in Figure 4.7. The accuracy function $f(\gamma)$ is defined as the ratio of space correctly determined as free:

$$f(\gamma) = \frac{\boldsymbol{V}}{\boldsymbol{V}_I} \tag{4.4}$$

We made the following assumption to ease the theoretical analysis:

Assumption 1. *We assume that the LiDAR is an idealized representation, which assumes a horizontal FoV of* 360° *and a symmetric vertical FoV in the range of* $[-\alpha, \alpha]$ *with infinitely small angular resolution (e.g.,* $\psi_{\texttt{lidar}} \to 0$*) and a detection range* R*. The occupancy mapping resolution is* d*.*

Theorem 1. *The function* $f(\gamma)$ *is derived based on Assumption 1 as follows:*

$$f(\gamma) = \begin{cases} \frac{\gamma_0^3}{\gamma} + A(1 - \frac{\gamma_0}{\gamma})\frac{1}{\gamma^2} & \gamma > \gamma_0 \\ 1 & 0 < \gamma \leqslant \gamma_0 \end{cases} \tag{4.5}$$

where A *and* γ_0 *are constant factors given by*

$$A = \frac{3(1 - \cos(\alpha)) + \frac{12}{\pi}\sin(\alpha)}{1 - \frac{1}{2}(\sin(\alpha)\cos^2(\alpha) + (1 - \sin(\alpha))^2(2 + \sin(\alpha)))} \tag{4.6a}$$

$$\gamma_0 = \sqrt{3}\sin(\min\{\operatorname{atan}(\frac{1}{\sqrt{2}}) + \alpha, \frac{\pi}{2}\}). \tag{4.6b}$$

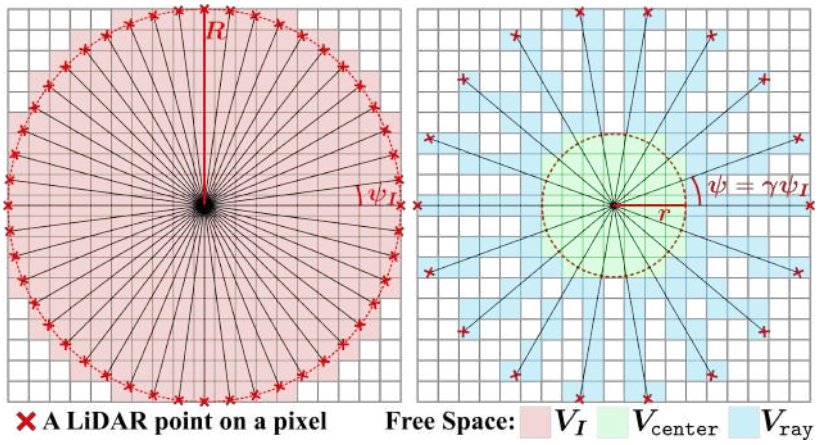


Figure 4.7: (a) Free space $\boldsymbol{V}_I$ determined by projecting to $\mathcal{M}_I$ with a resolution of ψ_I. (b) Free space $\boldsymbol{V}$ determined by projecting to$\mathcal{M}$ with a resolution of $\psi = \gamma\psi_I$, which is composed of $\boldsymbol{V}_{\texttt{center}}$ and $\boldsymbol{V}_{\texttt{ray}}$.

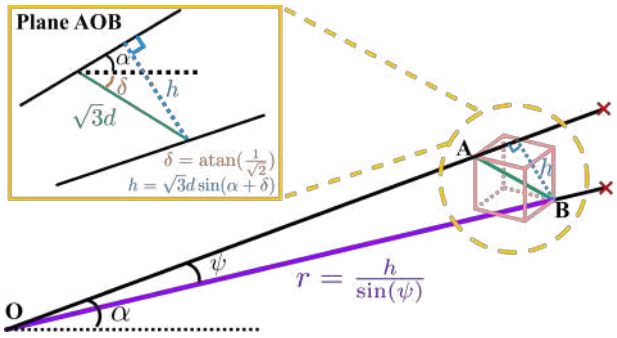


Figure 4.8: When casting rays from the sensor origin **O** to the LiDAR points on two neighbor pixels, there exist points **A** and **B** on the rays that the length of **AB** is large enough to contain a grid of size d. In this case, the length **AB** equals the diagonal of the grid, which is $\sqrt{3}d$, and the calculation of **OB** is conducted on plane **AOB**. The length of **OB** is the center radius r.

Proof. The volume of free space $\boldsymbol{V}$ is composed of two parts: center free space $\boldsymbol{V}_{\texttt{center}}$ and free space of rays $\boldsymbol{V}_{\texttt{ray}}$, as shown in Fig. 4.7. The radius r of the center sphere is obtained in consideration of vertical FoV $\Phi \in [-\alpha, \alpha]$ as follows:

$$r = \frac{\sqrt{3}\sin(\min\{\text{atan}(1/\sqrt{2}) + \alpha, \frac{\pi}{2}\})R}{\gamma} \tag{4.7}$$

An illustration to calculate the radius r is presented in Fig. 4.8. Let $\gamma_0 = \sqrt{3}\sin(\min\{\text{atan}(\frac{1}{\sqrt{2}}) + \alpha, \frac{\pi}{2}\})$, then $r = \frac{\gamma_0 R}{\gamma}$. The volume of center space $\boldsymbol{V}_{\texttt{center}}$ can be obtained in proportion to the entire free space $\boldsymbol{V}_{\boldsymbol{I}}$ as

$$\boldsymbol{V}_{\texttt{center}} = (\frac{r}{R})^3\boldsymbol{V}_I = (\frac{\gamma_0}{\gamma})^3\boldsymbol{V}_I \tag{4.8}$$

Since the volume of center space $\boldsymbol{V}_{\texttt{center}}$ is always smaller than the entire free space, γ_0/γ should be smaller than 1, leading to $\gamma > \gamma_0$. Otherwise, we have:

$$f(\gamma) = 1.0,\ 0 < \gamma \leqslant \gamma_0 \tag{4.9}$$

For the volume of free space of rays $\boldsymbol{V}_{\texttt{ray}}$, we consider the points $\mathcal{P} = \{p_{i,j} | i = 1, \ldots, N, j = 1, \ldots, \mathcal{M}_{\texttt{ray}}\}$ on the depth image $\mathcal{M}$ where N and M are the sizes of two dimension of the depth image. For each point $p_{i,j}$, the number of grids traversed by a ray from the origin to the point is the same as counting the grids intersected with the diagonal of a cuboid $\mathcal{C}$ as:

$$\begin{aligned} \mathcal{N}_{\texttt{grid}} =& a + b + c - \gcd(a, b) \\ & - \gcd(b, c) - \gcd(a, c) + \gcd(a, b, c) \end{aligned} \tag{4.10}$$

where a, b, c are the three dimensions of the cuboid. Suppose the point $p_{i,j}$ is projected to the depth image $\mathcal{M}$ with the spherical angle of (ϕ_i, θ_j), the size of the equivalent cuboid $\mathcal{C}$ is

$$\begin{gathered} a = \frac{R - r}{d} \cos(\phi_i) \cos(\theta_i),\ b = \frac{R - r}{d} \cos(\phi_i) \sin(\theta_i) \\ c = \frac{R - r}{d} \sin(\phi_i) \end{gathered} \tag{4.11}$$

Thus, the number of grids traversed by a ray from the origin to a point $p_{i,j}$ is obtained by approximation of (4.10) as:

$$\mathcal{N}_{\texttt{grid},i,j} = \frac{(R - r)}{d} \big(\mathrm{c}(\phi_i)(\mathrm{c}(\theta_j) + \mathrm{s}(\theta_j)) + \mathrm{s}(\phi_i)\big) \tag{4.12}$$

where $\mathrm{c}(\cdot)$ and $\mathrm{s}(\cdot)$ denote $\cos(\cdot)$ and $\sin(\cdot)$, respectively. As the volume of each grid is d^3, the total volume of these grids is derived as

$$\boldsymbol{V}_{\texttt{ray},i,j} = (1 - \frac{\gamma_0}{\gamma}) \big(\mathrm{c}(\phi_i)(\mathrm{c}(\theta_j) + \mathrm{s}(\theta_j)) + \mathrm{s}(\phi_i)\big) R d^2 \tag{4.13}$$

When summing the volume of free space traversed by each ray, we consider 1/8 of the points $\mathcal{P}$ for simplicity, which locates in an octant of free space $\boldsymbol{V}$ corresponding to the FoV angle of $[0, \pi/2] \times [0, \alpha]$. The depth image dimension corresponding to the

octant space is calculated as:

$$N \approx \frac{\alpha}{\gamma\psi_I}, \quad M \approx \frac{\pi}{2\gamma\psi_I} \tag{4.14}$$

The spherical coordinate (ϕ_i, θ_j) of $p_{i,j}$ can be obtained from the depth image resolution ψ as follows:

$$\phi_i = i\gamma\psi_I,\ \theta_j = j\gamma\psi_I \tag{4.15}$$

Summarizing the volume of free space covered by the rays from the origin to the point cloud $\mathcal{P}$ yields:

$$\begin{aligned} \boldsymbol{V}_{\texttt{ray}} &= 8\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\boldsymbol{V}_{\texttt{ray},i,j} \\ &= (4\pi(1-\cos(\alpha)) + 16\sin(\alpha))(1-\frac{\gamma_0}{\gamma})\frac{R^3}{\gamma^2} \end{aligned} \tag{4.16}$$

The volume of free space $\boldsymbol{V}_I$ is equivalent to the volume of LiDAR FoV, which can be obtained by the volume of a sphere cut by two sphere caps and two cones, which is

$$\begin{aligned} \boldsymbol{V}_I =& \frac{4\pi R^3}{3} - 2\cdot\frac{\pi R^3}{3}\sin(\alpha)\cos^2(\alpha) \\ & - 2\cdot\pi(R - R\sin(\alpha))^2(R - \frac{R - R\sin(\alpha)}{3}) \\ =& \frac{4\pi R^3}{3} - \frac{2\pi R^3}{3}\sin(\alpha)\cos^2(\alpha) \\ & - \frac{2\pi R^3}{3}(1-\sin(\alpha))^2(2+\sin(\alpha)) \end{aligned} \tag{4.17}$$

Finally, the accuracy function for the 3-Dimensional case is

$$\begin{aligned} f(\gamma) &= \frac{\boldsymbol{V}_{\texttt{center}} + \boldsymbol{V}_{\texttt{ray}}}{\boldsymbol{V}_I} \\ &= (\frac{\gamma_0}{\gamma})^3 + \frac{\boldsymbol{V}_{\texttt{ray}}}{\boldsymbol{V}_I} \\ &= (\frac{\gamma_0}{\gamma})^3 + A(1-\frac{\gamma_0}{\gamma})\frac{1}{\gamma^2},\ \gamma > \gamma_0 \end{aligned} \tag{4.18}$$

where A is a constant factor relative to LiDAR's vertical FoV range α

$$A = \frac{3(1-\mathrm{c}(\alpha)) + \frac{12}{\pi}\mathrm{s}(\alpha)}{1 - \frac{1}{2}(\mathrm{s}(\alpha)\mathrm{c}^2(\alpha) + (1-\mathrm{s}(\alpha))^2(2+\mathrm{s}(\alpha)))} \tag{4.19}$$

Summarizing (4.9) and (4.18), the accuracy function $f(\gamma)$ is provided as

$$f(\gamma) = \begin{cases} \frac{\gamma_0^3}{\gamma} + A(1 - \frac{\gamma_0}{\gamma})\frac{1}{\gamma^2} & \gamma > \gamma_0 \\ 1 & 0 < \gamma \leqslant \gamma_0 \end{cases} \tag{4.20}$$

□

According to Theorem 1, the relaxed factor γ can be determined by an expected accuracy value $\omega \in (0, 1]$ using the trigonometric form of Cardano's Formula [217], given as:

$$\gamma = \begin{cases} 2\sqrt{\frac{A}{3\omega}}\cos(\beta) & 0 < \omega < 1 \\ 1 & \omega = 1 \end{cases} \tag{4.21}$$

where β is given as

$$\beta = \frac{1}{3}\arccos\big(-\frac{A\gamma_0 - \gamma_0^3}{2\omega}(\frac{A}{3\omega})^{\frac{3}{2}}\big) \tag{4.22}$$

It is worth noting that for $\alpha \in [0, \pi/2]$, γ_0 always satisfies $\gamma_0 \geqslant 1$. Therefore, setting $\gamma = 1$ always results in $f(\gamma) = 1$, regardless of the value of α. Besides, it is noted the accuracy function $f(\gamma)$ attains a value of 1 for $\gamma \in (0, \gamma_0]$, as shown in equation (4.5). Therefore, in equation (4.21), we select $\gamma = 1$ as the solution for $\omega = 1$.

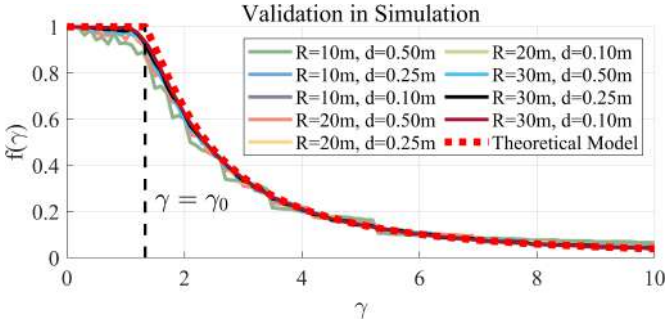


Figure 4.9: The simulation results for validation on accuracy function $f(\gamma)$.

The accuracy function $f(\gamma)$ is validated through a series of simulation experiments involving various LiDAR detection ranges and map resolutions. Specifically, given a detection range R and map resolution d, a batch of point clouds covering the free space within the detection range is generated on a sphere of radius R. To compute the volume $\boldsymbol{V}_I$, we count the number of cells in the map with resolution d that are identified as free by projecting the map to the original depth image $\mathcal{M}_I$ and multiplying the result

by the unit volume d^3. Similarly, we obtain the volume $\boldsymbol{V}$ by projecting the map onto the relaxed depth image $\mathcal{M}$. The accuracy function value $f(\gamma)$ is calculated using the definition in (4.4). In the simulations, we set the LiDAR's FoV Φ to $[-15°, 15°]$.

The results of the simulation experiments are presented in Fig. 4.9. The theoretical model fits the experimental curves well. However, slight errors arise when γ approaches γ_0 due to discretization in counting the number of cells to evaluate the volume of covered regions. These errors are evident in the curves of low resolution and short detection range (e.g., in the setup of $R = 10$m and $d = 0.5$m, which is the green solid line in Fig. 4.9).

4.5 Occupancy Mapping

In this section, we describe how to update and query the occupancy states using the map structure in D-Map.

4.5.1 Occupancy Map Structure

With an aim to optimize the balance between computation and memory efficiency, D-Map utilizes a hybrid map structure that leverages a hashing grid map to manage occupied space and an octree to handle unknown space.

4.5.1.1 Hashing Grid Map

As there are typically fewer occupied regions in the environment than free and unknown ones, we maintain the occupied space of the environment in a hashing grid map using the voxel hashing technique [117], which allows efficient update and query operations in a time complexity of $\boldsymbol{\mathcal{O}}(1)$. The hash key value for a given point $\boldsymbol{p} = [x, y, z]$ is computed by a hashing function `Hash`, defined as follows:

$$\begin{aligned} &\texttt{Hash}(\boldsymbol{p}) = (\texttt{P}^2 n_z + \texttt{P} n_y + n_x) \texttt{ mod Q} \\ &n_x = \lfloor \frac{x}{d} \rfloor,\ n_y = \lfloor \frac{y}{d} \rfloor,\ n_z = \lfloor \frac{z}{d} \rfloor, \end{aligned} \tag{4.23}$$

where d represents the resolution of the hashing grid map. `P` and `Q` are large prime numbers while `Q` also serves as the size of the hashing table. `mod` is the modulus calculation between two integers. The value of `P` and `Q` are carefully selected to minimize conflict probability [218], with `P` and `Q` set to 116101 and 201326611 in our work, respectively.

4.5.1.2 Octree

The tree-based map structure is a commonly used approach for occupancy mapping due to its high memory efficiency. Among various tree-based data structures, the octree [174] stands out as it outperforms other spatial data structures for dynamic updates [219]. Additionally, the spatial division on the octree naturally allows for determining the occupancy states of large cells during tree updates. Therefore, we utilize an octree to organize the unknown space.

In D-Map, a node on the octree contains the following elements:

- The point array `ChildNodes[8]` contains the address to its eight child nodes. The point array is empty if the node is a leaf node.
- The center `C` of the cell represented by the node.
- The size `L` of the cell represented by the node.

4.5.1.3 Initialization

To initialize the mapping procedure in D-Map, the occupancy states of the environment are considered to be entirely unknown. In the implementation, given an initial bounding box $\mathtt{C}_{\mathtt{bbx}}$ of the interested regions, the root node of the octree is initialized to represent an unknown cube $\mathtt{C}_{\mathtt{root}}$ whose the center `C` is allocated at the center of $\mathtt{C}_{\mathtt{bbx}}$. The size `L` of the root node is assigned as the longest side length of the bounding box $\mathtt{C}_{\mathtt{bbx}}$, and the point array `ChildNodes` for child nodes is initialized as empty. Besides, the hash table in the hashing grid map is initialized as an empty table. Notably, the initial bounding box does not restrict the mapping area, as both the octree and the hashing grid map allow for the growth of the mapping space on-the-fly [117, 122].

4.5.2 Occupancy Update

The occupancy updates in D-Map involve updating the hashing grid map and the octree, as described in Alg. 10. The point clouds are directly inserted into the hashing grid map using the hash function in (4.23) (Line 2) and rasterized into a depth image $\mathcal{M}$ (Line 3). A 2-D segment tree is constructed from the depth image $\mathcal{M}$ for the subsequent occupancy state determination as described in Section 4.4 (Line 4). The sensing area

Algorithm 10: Occupancy Update

Params: Map Resolution d, Initial Grid Size `E`,
LiDAR Detection Range R,
LiDAR FoV Φ and Θ
Input : Sensor Pose `T`, Point Cloud $\mathcal{P}$

```
Algorithm Start
    UpdateGridMap(P);
    M ←Rasterization(P);
    T ←Build2DSegTree(M);
    V ←SensingArea(T,R,Φ,Θ);
    UpdateOctree(RootNode,T,V,T);
Algorithm End
Function UpdateOctree(Node,T,V,T)
    L,C ←GetCenterAndSize(Node);
    if ! Intersected(T,C,V) then return;
    if L > E then
        S ← Undetermined;
    else
        S ←DetermineOccupancy(T,C,L,T);
    end
    switch S do
        case Unknown do
            return;
        end
        case Known do
            DeleteNode(Node);
        end
        case Undetermined do
            if L ⩽ d then
                DeleteNode(Node)
            else
                if IsLeaf(Node) then Split(Node);
                foreach Child in ChildNodes do
                    UpdateOctree(Child,T,V);
                end
            end
        end
    end
End Function
```

$\mathcal{V}$ for cell extraction is obtained from the sensor pose `T`, LiDAR detection range R and LiDAR FoV $\Phi \times \Theta$ (Line 5). D-Map employs an on-tree update strategy to update the occupancy states of octree nodes within the LiDAR sensing area $\mathcal{V}$ using a function named `UpdateOctree` (Line 6).

The function `UpdateOctree` is described in Lines 9∼33 and explained as follows. Starting from the root node of the octree, the corresponding cell center `C` and its size `L` are obtained (Line 9). The algorithm checks if the corresponding cell intersects with LiDAR sensing area $\mathcal{V}$, terminating further updates if there is no intersection (Line 10). To avoid meaningless occupancy state determination on a too-large cell that always returns undetermined, we directly split those cells larger than an initial cell size `E` for efficiency (Line 11∼13). Otherwise, the occupancy state `S` is determined by Alg. 9 (Line 14). The operation on the tree node is decided based on the occupancy state `S`: If the cell is determined as unknown, the node is kept on the tree, and no further operations are required (Line 17∼19). The node is removed from the tree if the cell is known (i.e., free or occupied) (Line 20∼22). If the occupancy state of the node cannot be determined, further updates are required (Line 23∼32). A special condition is considered when the cell size `L` achieves the map resolution d (Line 24∼26). In this case, at least one pixel with a depth no smaller than the cell's minimum depth `BoxMin` exists (otherwise, it is determined as unknown for `dMax` $<$ `BoxMin`, see Alg. 9). As a result, the cell can be determined as known since it must have been either hit or traversed by the corresponding point cloud on that pixel. Otherwise, if the cell size is greater than the map resolution, we split the node, if not have done so (Line 27), and performed updates on its eight children (Line 28∼30).

The recursive function `UpdateOctree` operates in a manner that visits cells from the largest to the smallest size. This approach allows the determination of occupancy states directly on large cells (if they are unknown or free), thus avoiding unnecessary updates on smaller cells. As a result, occupancy updates on the octree are performed concurrently with the tree traversal, resulting in a so-called "on-tree" update strategy. This strategy is distinct from the classical pipeline [96, 214], which updates cells at the constant map resolution d after ray-casting. Furthermore, D-Map continuously removes known cells from the environment's unknown space, making it a decremental mapping method. This property enhances the efficiency as the mapping process progresses.

4.5.3 Occupancy State Query

Since two data structures are used in D-Map, two steps are performed to obtain the occupancy state of a cell at map resolution. Firstly, the cell is determined as unknown if its corresponding node exists on the octree. Otherwise, the region is determined as known. Subsequently, the hashing grid map is queried to determine whether it is occupied. If not, the region is determined as free.

4.6 Time Complexity Analysis

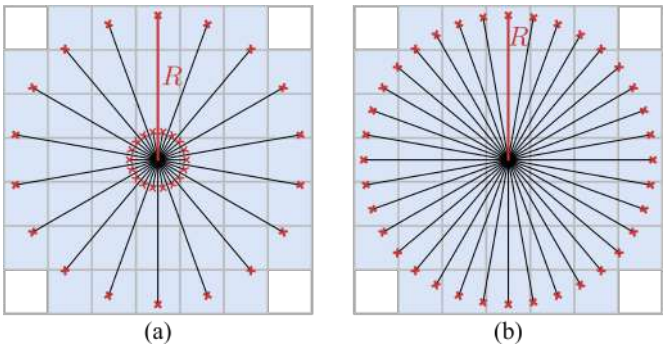


Figure 4.10: A 2-D illustration of the worst-case occupancy updates for (a) D-Map and (b) ray-casting-based methods. In the worst-case scenario for D-Map, two neighboring points appear at the minimum and maximum distance to the sensor origin, resulting in a serrated-shaped point cloud. In contrast, the worst-case scenario for ray-casting-based methods results in a spherical-shaped point cloud located at the maximum distance to the sensor origin.

In this section, we analyze the time complexity of updating and querying the occupancy states on D-Map. To facilitate further analysis of the benchmark experiments in Section 4.7, we additionally provide a time complexity analysis of the classical occupancy update pipeline, which involves ray-casting prior to map updates.

4.6.1 Occupancy State Update

The occupancy state updates in D-Map are composed of two parts: updating occupied cells on the hashing grid map and updating unknown cells on the octree. The time complexity for occupancy updates is mainly due to the on-tree update in the octree data structure, as the time of maintaining occupied points in a hashing grid map is constant at $\mathcal{O}(1)$. The on-tree update strategy involves searching for nodes within the LiDAR sensing area on the octree and determining their occupancy states. To analyze

the time complexity, we first consider the number of nodes to be visited on the octree, as presented in the following lemma:

Lemma 3. *Suppose the maximum side length of environment bounding box C_{bbx} on the octree is D, and map resolution is d. The number of nodes to be visited $\mathcal{S}_{\texttt{DMap}}$ on the octree for the on-tree update strategy in D-Map is bounded as:*

$$\mathcal{S}_{\texttt{DMap}} = \boldsymbol{\mathcal{O}}\big(n\frac{R}{d}\log\big(\frac{D}{d}\big)\big) \tag{4.24}$$

where n is the number of points projected to the depth image, and R is the detection range of the LiDAR sensor.

Proof. The on-tree update strategy searches the nodes inside or intersecting with the sensing area. To analyze the time complexity, we examine the worst-case scenario for updating the D-Map, which occurs when the octree splits the sensing area into the smallest size. This situation can arise when the point cloud has a serrated shape (as shown in Fig. 4.10(a)), leading to a depth image resembling a checkerboard where there are abrupt changes in depth values from a minimum to a maximum between neighboring pixels. In such situations, the number of leaf nodes visited in the D-Map is equivalent to the number of grids traversed by rays of points on the depth image. For each ray, the number of traversed grids is approximated as $\boldsymbol{\mathcal{O}}(R/d)$, as derived in Section 4.4.4, (4.10) and (4.12). Thus, the total number of leaf nodes in the worst case is given as:

$$\mathcal{S}_{\texttt{leaf}} = \boldsymbol{\mathcal{O}}\big(\frac{nR}{d}\big) \tag{4.25}$$

Next, we consider the other nodes visited along the path toward the leaf nodes. The number of these accompanying nodes $\mathcal{S}_{\texttt{acc}}$ is determined by the height of the octree, which is $\log(D/d)$, as follows:

$$\mathcal{S}_{\texttt{acc}} = \boldsymbol{\mathcal{O}}\big(\frac{nR}{d}\log\big(\frac{D}{d}\big)\big) \tag{4.26}$$

Therefore, the total number of visited nodes in the worst case of our on-tree update strategy is obtained as

$$\begin{aligned}\mathcal{S}_{\texttt{DMap}} &= \mathcal{S}_{\texttt{leaf}} + \mathcal{S}_{\texttt{acc}} \\ &= \mathcal{O}\big(\frac{nR}{d}\big) + \mathcal{O}\big(\frac{nR}{d}\log\big(\frac{D}{d}\big)\big) \\ &= \mathcal{O}\big(\frac{nR}{d}\log\big(\frac{D}{d}\big)\big)\end{aligned} \tag{4.27}$$

□

Then we derive the time complexity of occupancy state determination (see Alg. 9) that is frequently used in the update process.

Lemma 4. *The time complexity of occupancy state determination is* $\mathcal{O}(\log(\Phi_I/\psi_I)\log(\Theta_I/\psi_I))$*, where* Φ_I *and* Θ_I *are the FoV angles of the depth image and* ψ_I *is the depth image resolution.*

Proof. When considering the procedure of occupancy state determination, the time complexity is dominated by the range query on the 2-D segment tree. As explained in Section 4.4.2, the time complexity of a range query on a 2-D segment tree is $\mathcal{O}(\log N \log M)$ where N and M are the sizes of two dimensions. In our case, the 2-D segment tree is constructed from a depth image, with $N = \lceil \Phi_I/\psi_I \rceil$ and $M = \lceil \Theta_I/\psi_I \rceil$. Thus, the time complexity of occupancy state determination is $\mathcal{O}(\log(\Phi_I/\psi_I)\log(\Theta_I/\psi_I))$. □

Finally, the time complexity for occupancy updates on D-Map is given as follows:

Theorem 2. *The time complexity of occupancy updates in D-Map is* $\mathcal{O}\big(\frac{nR}{d}\log\big(\frac{D}{d}\big)\big)$*, where* n *is the number of points projected to the depth image,* d *is the map resolution, and* R *is the LiDAR detection range.*

Proof. In the on-tree update strategy, an occupancy state determination process is applied to each node that is visited. The time complexity of this process is given by Lemma 4. Additionally, the number of visited nodes is given by $\mathcal{S}_{\texttt{DMap}}$ in Lemma 3. Thus, the overall time complexity can be expressed as the product of these two factors:

$$\mathcal{O}\Big(\frac{nR}{d}\log\big(\frac{D}{d}\big)\log\big(\frac{\Phi_I}{\psi_I}\big)\log\big(\frac{\Theta_I}{\psi_I}\big)\Big) \tag{4.28}$$

In D-Map, the depth image resolution is bounded by (4.2) as:

$$\psi_I = \max\{\frac{d}{R}, \psi_{\texttt{lidar}}\} \geqslant \psi_{\texttt{lidar}} \tag{4.29}$$

Therefore, the time complexity in (4.28) can be written as:

$$\boldsymbol{\mathcal{O}}\Big(\frac{nR}{d}\log(\frac{D}{d})\log(\frac{\Phi_I}{\psi_{\texttt{lidar}}})\log(\frac{\Theta_I}{\psi_{\texttt{lidar}}})\Big) \tag{4.30}$$

Considering that Φ_I and Θ_I are bounded by constants (e.g., π) and $\psi_{\texttt{lidar}}$ is a constant parameter related to the LiDAR sensor, the time complexity can be further simplified as $\boldsymbol{\mathcal{O}}\big(\frac{nR}{d}\log(\frac{D}{d})\big)$. □

We provide the time complexity of the occupancy updates on a grid-based map and an octree-based map as follows:

Theorem 3. *The time complexity to update the occupancy states on a grid-based map is* $\boldsymbol{\mathcal{O}}(\frac{nR}{d})$ *and on an octree-based map is* $\boldsymbol{\mathcal{O}}(\frac{nR}{d}\log(\frac{D}{d}))$.

Proof. The occupancy updates involve ray casting to determine the grids intersected with the rays, followed by updating their occupancy states on the map. We start by considering the worst-case scenario where all points are at the detection range R (as shown in Fig. 4.10(b)). In such cases, the number of grids traversed by rays can be approximated as $\mathcal{S}_{\texttt{rc}} = \boldsymbol{\mathcal{O}}(\frac{nR}{d})$, as derived in Section 4.4.4, (4.10) and (4.12). The time complexity of map updates is $\boldsymbol{\mathcal{O}}(1)$ for a grid-based map and $\boldsymbol{\mathcal{O}}(\log(D/d))$ for an octree-based map. Therefore, the time complexity for occupancy updates on a grid-based map and an octree-based map is $\boldsymbol{\mathcal{O}}(\frac{nR}{d})$ and $\boldsymbol{\mathcal{O}}(\frac{nR}{d}\log(\frac{D}{d}))$, respectively. □

Remark 1. Although D-Map and the octree-based map have the same time complexity of $\boldsymbol{\mathcal{O}}(\frac{nR}{d}\log(\frac{D}{d}))$, their sources of time complexity are entirely different. Since D-Map updates the occupancy states concurrently with the tree traversal, as discussed in Section 4.5.2, the time complexity of D-Map depends solely on the number of visited nodes $\mathcal{S}_{\texttt{DMap}}$ during the on-tree update process, which contributes to $\boldsymbol{\mathcal{O}}(\frac{nR}{d}\log(\frac{D}{d}))$ directly. In contrast, occupancy updates on an octree-based map consist of two consecutive procedures: ray-casting, which contributes to $\boldsymbol{\mathcal{O}}(\frac{nR}{d})$, and map updates, which contribute to $\boldsymbol{\mathcal{O}}(\log(D/d))$, resulting in the same total time complexity of $\boldsymbol{\mathcal{O}}(\frac{nR}{d}\log(\frac{D}{d}))$.

Remark 2. The time complexity of D-Map is calculated without considering the removal of known cells, which could reduce the number of visited nodes $\mathcal{S}_{\texttt{DMap}}$ substantially. In contrast, existing ray-casting-based methods (either grid-based map or octree-based map) do not have such decremental properties.

Remark 3. We derive the time complexity of occupancy updates for our D-Map and ray-casting-based methods (i.e., the grid-based and octree-based maps) based on their respective worst-case scenario, as illustrated in Fig. 4.10. However, the occurring chances of their worst-case scenarios are different. Generally speaking, the worst-case scenario for ray-casting-based methods is more likely to occur in the real world (e.g., large open space) than for our D-Map.

4.6.2 Occupancy State Query

The time complexity of the occupancy state query in D-Map is provided as follows:

Theorem 4. *The time complexity of querying the occupancy state of a cell in D-Map is* $\mathcal{O}(\log(D/d))$.

Proof. The query process in D-Map contains a query on the octree and a query on the hashing grid map whose time complexity is $\mathcal{O}(\log(D/d))$ and $\mathcal{O}(1)$, respectively. Therefore, the time complexity of the occupancy state query is easily obtained as $\mathcal{O}(\log(D/d))$. □

4.7 Benchmark Results

In this section, exhaustive benchmark experiments are conducted on various datasets with different LiDARs to evaluate the computational efficiency, accuracy, and memory consumption against other state-of-the-art occupancy mapping approaches. Besides the time performance evaluation, we conduct an ablation study based on the time complexity to provide deeper insight into the efficiency of our approach.

Table 4.1: Details of Datasets for Benchmark Experiment

	Mapped Area (m^3)	Number of Scans	Average Point Number (per scan)
FR_079	$48 \times 37 \times 5$	66	89445.9
Freiburg	$293 \times 168 \times 29$	81	247817.1
Workshop	$40 \times 40 \times 8$	7340	23540.8
MainBuilding	$636 \times 251 \times 26$	1402	22124.9
Kitti_04	$551 \times 160 \times 37$	271	125265.2
Kitti_06	$616 \times 181 \times 47$	1101	122189.4
Kitti_07	$362 \times 347 \times 40$	1101	121225.5

4.7.1 Datasets

The experiments are conducted on three open datasets and one private dataset. The first public dataset is *Kitti* dataset which captured depth measurements by a Velodyne HDL-64E rotating 3D laser scanner at 10 Hz [220]. Considering the tremendous memory consumption associated with grid-based maps that will be utilized in our benchmark experiments, we chose sequences *Kitti_04*, *Kitti_06*, and *Kitti_07* to conduct the benchmark evaluation. The second public dataset is an outdoor sequence *MainBuilding* provided in FAST-LIO [130], which used a semi-solid state 3D LiDAR sensor Livox Avia to collect data at 10 Hz. The third public dataset, provided in the work Octomap [96], includes an indoor sequence *FR-079* and an outdoor sequence *Freiburg*. In addition, we evaluate the benefit of decremental property on a private dataset *Workshop*, where a cluttered indoor environment is completely scanned manually using 3D LiDAR Livox Avia at 10 Hz. The reconstruction result of *Workshop* dataset is available in [221]. It is noticed that the odometry estimation for occupancy mapping in sequence *Workshop* and *MainBuilding* was acquired by our LiDAR-inertial odometry framework FAST-LIO2 [219]. Table 4.1 provides further details on the aforementioned datasets.

4.7.2 Evaluation Setup

We compared D-Map with state-of-the-art occupancy mapping methods commonly used in robotics applications, including a grid-based method (Grid Map updated by 3DDDA [222]), a tree-based method (Octomap [96]), and the extended versions of Grid Map and Octomap using super rays and culling regions [214] (denoted with "SR&CR(Grid)"

and "SR&CR(Octo)"). These occupancy mapping methods were selected for their efficient updates and widespread usage in the field. For the Octomap, SR&CR(Grid), and SR&CR(Octo), we used their open-source implementations available on GitHub repositories[1][2]. For the grid map, we modify the open-source SR&CR(Grid) by disabling the super rays and culling regions and termed it as "GridMap". It is noticed that all grid-based maps employed voxel hashing to achieve better memory efficiency. The benchmark experiments are conducted at various map resolutions, ranging from a rough resolution of 1.0 m to a finer resolution of 1 cm for indoor sequences. Due to system memory limitations, the minimum map resolution for outdoor sequences is set to 5 cm.

The computation platform for evaluation is an Intel NUC computer with CPU Intel i7-10710U and 64 GB RAM. To facilitate evaluation on high-resolution maps in large-scale environments, a 64 GB swap space on Solid State Drive (SSD) is allocated to account for extensive memory usage.

D-Map uses a fixed set of parameters for all benchmark experiments. The completeness threshold ε in Alg. 9 is set to 0.8, while the initial cell size `E` in Alg. 10 is set to 5.0 m. The depth image resolution is determined using (4.2) without relaxation (i.e., $\gamma = 1$, which is not greater than γ_0 and leads to $f(\gamma) = 1$, as explained in Section 4.4.4). The LiDAR detection range R and LiDAR angular resolution $\psi_{\texttt{lidar}}$ are acquired from LiDARs' manual sheets, except for *Workshop* indoor sequence whose detection range is assigned as 60 m in consideration of the indoor environment.

Given the high accuracy and low false alarm rate of LiDAR sensors, the space that is hit by a LiDAR pulse can be reliably considered occupied, and that passed by the laser ray can be considered free. We use the following parameter setup for other occupancy mapping approaches in the benchmark experiment to ensure consistency with this assumption that our D-Map has utilized. We set the probabilities for hit and miss by a ray (i.e., for occupied and free space) to $P_{\texttt{occ}} = 0.9999$ and $P_{\texttt{free}} = 0.4999$, respectively, and the minimum and maximum clamping probabilities to $P_{\texttt{min}} = 0.499$ and $P_{\texttt{max}} = 0.9999$. Moreover, we set the initial occupancy probabilities of all cells to 0.5, which represents unknown cells without any update (or observation) from LiDAR points, and we regard the cells with occupancy probability values smaller than 0.5 (due to

[1] https://github.com/OctoMap/octomap
[2] https://github.com/PinocchioYS/SuperRay

subsequent LiDAR points update) as free and those with occupancy probability values larger than 0.5 as occupied. Finally, the methods SR&CR(Grid) and SR&CR(Octo) both require a threshold parameter k for the minimum number to generate super rays, which is set to the default value $k = 20$ as in [214].

4.7.3 Efficiency Evaluation and Analysis

4.7.3.1 Benchmark Results

We summarize the average update time of each occupancy mapping approach and report the results in Table 4.2. The update time for SR&CR(Grid) and SR&CR(Octo) includes the time required for preprocessing, which involves generating super rays and constructing culling regions, as well as the time for occupancy updates involving ray-casting and map updates. In contrast, for GridMap and Octomap, only the time for ray-casting and map updates is considered, as no preprocessing is required. As for our D-Map, we count the preprocessing time for depth image rasterization and 2-D segment tree construction as well as the occupancy update time for updating the octree and hashing grid map. As presented in Table 4.2, our D-Map consistently outperforms the other mapping approaches across various map resolutions in three sequences of the *Kitti* dataset. Particularly, at a map resolution of 5 cm in *Kitti_ 06*, D-Map achieves a remarkable speedup of 7.66 times and 4.78 times faster than GridMap and Octomap, respectively. However, in the *MainBuilding* datasets, GridMap shows the highest efficiency at different map resolutions, followed by our D-Map and SR&CR(Grid). The lower efficiency of D-Map is caused by the sparse clouds in *MainBuilding* sequence, which only has an average number of 22 k per scan, while *Kitti* datasets have over 120 k points per scan on average. This finding suggests that our method is less efficient in updating sparse point clouds since a sparse depth image rasterized from the sparse point clouds causes redundant occupancy state queries on those unobserved regions during the updating process. In contrast, ray-casting-based mapping approaches exhibit better performance due to their precise traversal of observed regions. However, this disadvantage of D-Map could be compensated by the decremental property as shown in the *Workshop* indoor sequence, where the scanning process frequently visited the previously explored areas with an aim to map the entire indoor environment exhaustively. This scanning process, which is often used in mapping applications, enables our D-Map to continuously remove

Table 4.2: Comparison of Update Time (ms) at Different Resolution

	Kitti_04					*Kitti_06*					*Kitti_07*				
Resolution (m)	1.0	0.5	0.25	0.1	0.05	1.0	0.5	0.25	0.1	0.05	1.0	0.5	0.25	0.1	0.05
GridMap	47.32	89.76	190.78	720.33	3636.54	50.22	97.70	203.56	1110.29	10505.13	37.04	68.64	140.28	536.51	3431.81
Octomap	398.52	637.52	964.52	2459.13	7623.91	437.43	664.70	1076.62	2629.29	6558.01	299.72	469.21	775.76	1777.65	3877.65
SR&CR(Grid)	43.08	129.51	298.76	1398.72	7360.95	48.86	116.96	322.30	2422.06	17977.50	30.17	72.91	180.42	995.12	6290.95
SR&CR(Octo)	44.47	125.31	389.37	2330.39	9287.51	56.49	161.41	554.35	2974.88	12361.56	30.05	85.18	307.54	1634.40	6153.84
Ours	**22.67**	**39.05**	**101.14**	**470.67**	**1905.32**	**22.71**	**36.08**	**85.91**	**367.08**	**1371.69**	**19.12**	**27.27**	**59.33**	**209.46**	**779.32**

	Workshop							*MainBuilding*				
Resolution (m)	1.0	0.5	0.25	0.1	0.05	0.025	0.01	1.0	0.5	0.25	0.1	0.05
GridMap	4.94	8.42	15.97	50.17	127.43	346.68	13702.461	**9.63**	**19.91**	**45.88**	**162.65**	**640.83**
Octomap	42.21	59.62	92.76	175.17	297.93	589.07	7568.64	64.82	106.43	199.05	529.66	1333.22
SR&CR(Grid)	**2.64**	7.35	18.65	72.68	244.95	1514.41	39624.96	10.86	27.65	72.61	315.12	1384.12
SR&CR(Octo)	**2.64**	9.29	29.96	162.54	717.84	3352.30	50174.95	12.61	44.12	156.80	908.08	3783.49
Ours	3.02	**4.45**	**9.26**	**20.10**	**42.12**	**88.16**	**915.20**	11.16	31.56	112.52	332.83	991.66

	FR_079							*Freibrug*				
Resolution (m)	1.0	0.5	0.25	0.1	0.05	0.025	0.01	1.0	0.5	0.25	0.1	0.05
GridMap	10.14	15.47	26.47	63.77	152.04	438.84	4082.14	61.41	109.99	230.90	1021.84	6005.56
Octomap	3.89	7.90	17.45	66.48	347.57	1514.97	7568.64	356.43	548.27	990.92	3357.36	10637.87
SR&CR(Grid)	**3.76**	7.91	16.34	53.34	237.25	816.58	4805.09	40.72	110.27	285.78	1522.21	7911.33
SR&CR(Octo)	3.87	**7.70**	18.28	68.21	375.14	1613.46	8402.84	43.68	112.59	354.07	2186.24	8709.99
Ours	10.09	11.29	**15.11**	**29.01**	**65.22**	**205.06**	**756.33**	**31.40**	**40.48**	**82.55**	**678.09**	**3599.96**

known space from the octree map, achieving higher efficiency than the other methods. Fig. 4.11 presents the update time consumption of different occupancy approaches at a 1 cm map resolution, clearly illustrating the decreasing update time of D-Map as the environment is explored.

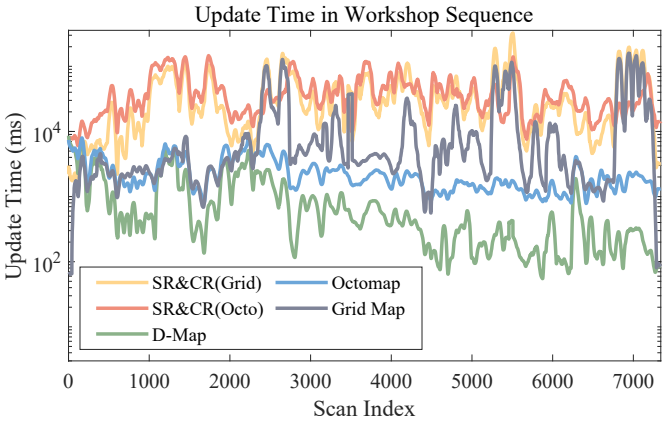


Figure 4.11: Update time in *Workshop* indoor sequence at a high map resolution of 1 cm.

D-Map demonstrates superior performance compared to other approaches at high map resolutions ($d \leqslant 0.25\,\mathrm{m}$) in the indoor sequence *FR_079*, as well as at various resolutions in the outdoor sequence *Freiburg*. However, at low resolutions ($d \geqslant 0.5\,\mathrm{m}$) in *FR_079* sequence, the performance is limited by the preprocessing time required for depth image rasterization (approximately 5.7 ms) and the processing time to insert occupied points into the hashing grid map (about 4.2 ms).

Across all benchmark experiments conducted in this study, we have observed that SR&CR(Grid) and SR&CR(Octo) exhibit low efficiency at high resolution compared to GridMap and Octomap, respectively, without super rays or culling regions. This phenomenon has been previously noted and discussed in [214], where it is attributed to the low grouping ratio of super rays and the need for the timely rebuilding of culling regions at high resolution. Additionally, we have also observed anomalous time performances when comparing the grid-based method (i.e., GridMap and SR&CR(Grid)) against the tree-based methods (i.e., Octomap and SR&CR(Octo)). Normally, grid-based maps are expected to offer superior update efficiency compared to tree-based maps. However, we have observed that grid-based methods are slower than tree-based methods at high

resolution (d =5 cm) in the sequences *Kitti_ 06* and *Kitti_ 07*. The inefficiency of grid-based methods can be attributed to their reliance on swap space on the SSD during the update process since their memory usage exceeds the capacity of RAM. As memory access on SSD is significantly slower than RAM, this reliance on swap space can lead to the observed slower update performance.

4.7.3.2 Efficiency Analysis

We present a comprehensive analysis of our superior efficiency performance based on the time complexity analysis in Section 4.6. As demonstrated in Section 4.6, the efficiency of D-Map is solely determined by the number of visited cells $\mathcal{S}_{\texttt{DMap}}$ due to its "on-tree" update strategy. However, GridMap, Octomap, SR&CR(Grid), and SR&CR(Octo), which follow the classical occupancy update pipeline using ray-casting, have an efficiency that depends on two factors: the number of cells $\mathcal{S}_{\texttt{rc}}$ traversed by rays and the time complexity for map updates, which is $\mathcal{O}(1)$ for grid-based methods and $\mathcal{O}(\log(D/d))$ for octree-based methods. Therefore, we conduct a comparison between the number of visited cells $\mathcal{S}_{\texttt{DMap}}$ in D-Map and the number of updated cells $\mathcal{S}_{\texttt{rc}}$ in the ray-casting-based methods, followed by a comparison of the corresponding occupancy update time (without counting the preprocessing time). Specifically, we opt to compare D-Map with SR&CR(Grid) and SR&CR(Octo), which incorporate super rays and culling region techniques to reduce the number of cells. The comparison is carried out on *FR_ 079* and *Freiburg* sequences, with the results presented in Fig. 4.12.

In comparison between $\mathcal{S}_{\texttt{DMap}}$ and $\mathcal{S}_{\texttt{rc}}$, we conduct an ablation study to investigate the performance of all methods (i.e., SR&CR(Grid), SR&CR(Octo), and D-Map) without any removal of cells. For our D-Map, we disable the removal of known cells from the octree. For SR&CR(Grid) and SR&CR(Octo), we disable the culling region technique and denote these modified methods as "SR(Grid)" and "SR(Octo)", respectively. As shown in Fig. 4.12, SR(Grid) exhibits the highest count of cells across all resolutions. followed by SR(Octo) and our D-Map without removal. The difference in the number of updated cells between SR(Grid) and SR(Octo) is attributed to the batching-based method in SR(Octo) that batches the uniform cells traversed by multiple rays to reduce the cells. The difference between the two methods becomes smaller at higher resolution as there are fewer uniform cells traversed by multiple rays. It is worth noticing that,

although batching is essential to tree-based methods to account for the cost of tree updates, it is unnecessary for grid-based maps, which have an efficient update in simple time complexity of $\mathcal{O}(1)$. As analyzed in Section 4.6, D-Map has a theoretical number of visited nodes $\mathcal{S}_{\texttt{DMap}} = \mathcal{O}(\frac{nR}{d}\log(\frac{D}{d}))$ in its worst case, which is larger than the theoretical number of traversed cells $\mathcal{S}_{\texttt{rc}} = \mathcal{O}(\frac{nR}{d})$ in the worst case of ray-casting-based methods. However, in this ablation study, D-Map without removal visits a comparable number of nodes to SR(Octo) because the worst case of D-Map rarely occurs. Specifically, D-Map without removal processes fewer nodes than SR(Octo) at low resolutions (i.e., $d \geqslant 0.25$m) in indoor sequence *FR_079* and at all resolutions except 5 cm in outdoor sequence *Freiburg*. When approaching high resolutions (e.g., $d < 0.25$m in indoor sequence and $d = 0.05$m in outdoor sequence), the number of visited nodes $\mathcal{S}_{\texttt{DMap}}$ on D-Map is closer to its worst case, which has a higher increasing rate on a logarithmic scale than $\mathcal{S}_{\texttt{rc}}$.

The removal of known cells and the culling region technique are then enabled in corresponding methods for further investigation. D-Map demonstrates the most significant reduction in the number of cells among all approaches when the removal technique is enabled, while the culling region technique only slightly lowers the number of updated cells in SR&CR(Grid) and SR&CR(Octo).

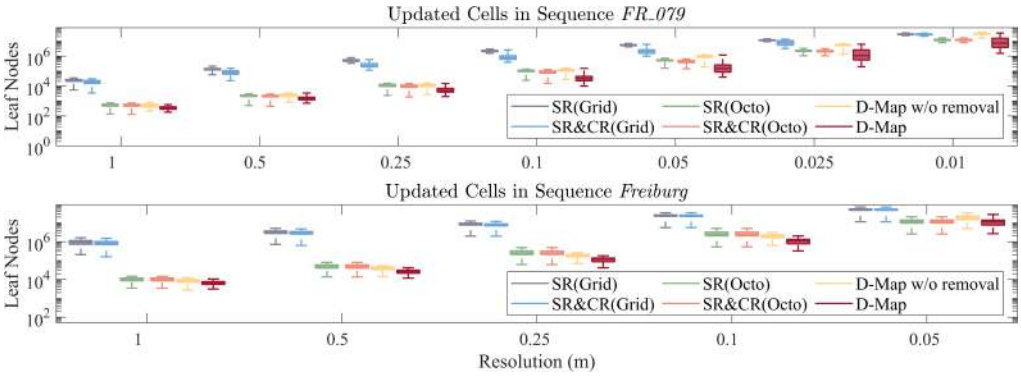


Figure 4.12: The number of cells to be updated in sequences *FR_079* and *Freiburg*.

Finally, we compare the time consumption for occupancy updates in both sequences, as shown in Fig. 4.13. When disabling the removal of known cells in D-Map and the culling region technique, the results reveal that D-Map without removal outperforms the others across all map resolutions in sequence *Freiburg*, as well as at low resolutions (i.e., $d > 5$cm) in sequence *FR_079*. At high resolutions (i.e., $d \leqslant 0.5$m) in sequence

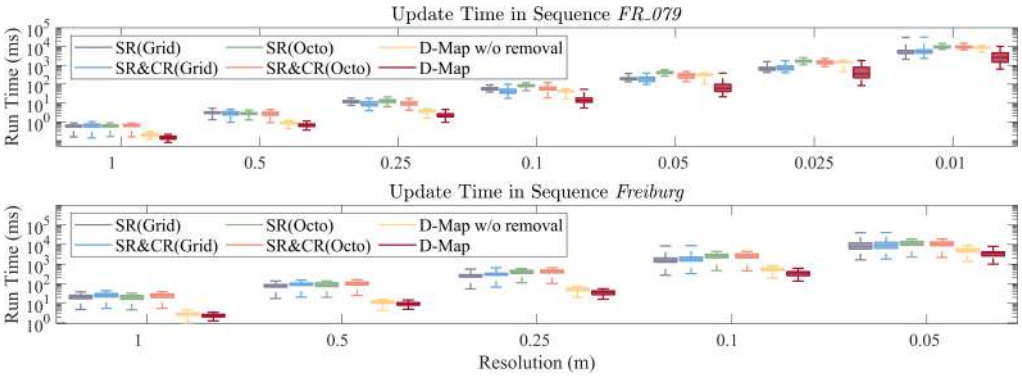


Figure 4.13: The update time in sequences *FR_079* and *Freiburg*.

FR_079, SR(Grid) achieves better performance than D-Map without removal due to a smaller number of cells to be updated. However, despite having a slightly larger number of visited nodes, D-Map without removal maintains its leading position over SR(Octo), which is hindered by the timely updates on the tree structure. When enabling the removal of known cells, D-Map demonstrates the highest efficiency across different map resolutions, owing to the fewest cells requiring to be updated.

4.7.4 Accuracy and Memory Evaluation

4.7.4.1 Accuracy Benchmark

We conduct accuracy benchmark experiments using the mapping results from Octomap [96] as ground truth. Specifically, we calculate the accuracy of D-Map by counting the cells with correct occupancy states over the total number of cells inside the mapping area and with the corresponding states. The benchmark results of unknown space and free space are presented in Table 4.3. It is noted that the accuracy of occupied space is very close to 100 % in all experiments and thus not presented in the table. This is due to our high confidence in occupied space based on the assumption of high accuracy and low false alarm rate of LiDAR sensors.

The overall accuracy performance of D-Map shows a slight accuracy loss compared to Octomap, as reported in Table 4.3. The inaccuracy mainly arises from the occupancy state determination module on the depth image. At low resolutions (i.e., $d \geqslant 0.5$m), the accuracy of unknown space is relatively lower due to the more severe discretization error introduced in (4.3) when determining the projected area on a lower-resolution

Table 4.3: Mapping Accuracy at Different Resolution

	Unknown Space						Free Space						
Resolution (m)	1.0	0.5	0.25	0.1	0.05	0.025	0.01	1.0	0.5	0.25	0.1	0.05	0.025	0.01
Kitti_04	97.33%	98.29%	99.67%	99.77%	99.46%	-	-	98.54%	98.49%	96.73%	94.61%	93.34%	-	-
Kitti_06	97.77%	98.62%	99.87%	99.84%	99.24%	-	-	98.80%	98.77%	97.21%	94.99%	94.53%		
Kitti_07	97.68%	99.50%	99.64%	99.87%	99.40%	-	-	98.40%	98.36%	96.91%	94.93%	94.24%		
FR_079	83.53%	96.46%	97.86%	98.96%	99.66%	99.17%	95.39%	95.06%	95.72%	96.55%	97.88%	95.14%	91.68%	93.56%
Freiburg	97.79%	98.93%	99.39%	99.91%	99.96%	-	-	98.45%	98.61%	98.80%	92.60%	83.28%	-	-
Workshop	98.68%	99.19%	99.61%	99.82%	99.87%	99.95%	99.99%	95.11%	96.74%	98.08%	98.80%	99.01%	98.90%	99.00%
MainBuilding	99.51%	99.71%	99.77%	99.58%	99.80%	-	-	97.65%	97.56%	97.08%	95.10%	87.76%	-	-

depth image. The results also show a decreasing accuracy of free space when the map resolution grows smaller. This decreasing accuracy is introduced by projecting a cell by its circumsphere on the depth image. When a ray crosses a large cell only at its corner but not intersects with its circumsphere, this large cell is incorrectly decided as not intersecting with any rays without further splitting and query. Thus, the small cells at the corner space are falsely determined as unknown. With a higher map resolution, more small free cells in the corner space are determined as unknown, leading to a decreasing accuracy. To improve the accuracy of free space for such cases, it is recommended to use a smaller initial cell size `E` to avoid missing the cell corner. However, a trade-off must be considered as a smaller `E` might miss the opportunity to update on large cells, leading to slower occupancy updates.

Notably, since we utilize Octomap as ground truth, the discretization error in Octomap also contributes to the accuracy loss in our benchmark experiment. Besides, despite the quantization error, Octomap has been widely used in real-world applications. Therefore, the accuracy achieved in our experiment, as reported in Table 4.3, is adequate for various practical scenarios, as demonstrated in Section 4.8.

4.7.4.2 Memory Consumption

We evaluated the memory consumption of each mapping approach and reported the results in Table 4.4. Given that SR&CR(Grid) and SR&CR(Octo) utilize the same map structure as GridMap and Octomap with equivalent mapping results, we only compare the memory consumption of D-Map against GridMap and Octomap. As shown in Table 4.4, Octomap exhibits the lowest memory consumption in most experiments, while GridMap consumes the highest. The memory consumption of D-Map follows that of Octomap closely. The excess memory consumption of D-Map mainly arises from using hashing grid map for maintaining occupied space. When high-resolution maps are built in large-scale environments (e.g., 5 cm in *Kitti* datasets), our D-Map exhibits better memory efficiency than Octomap due to its decremental property as well as the proper combination of grid-based and tree-based structures for different occupancy states.

Table 4.4: Comparison of Memory Consumption (MB) at Different Resolution

Kitti_ 04						*Kitti_ 06*						*Kitti_ 07*					
Resolution (m)	1.0	0.5	0.25	0.1	0.05	1.0	0.5	0.25	0.1	0.05	1.0	0.5	0.25	0.1	0.05	1.0	0.5	0.25	0.1	0.05
Grid Map	13.00	97.11	751.00	6964.45	33012.05	25.32	194.32	1532.03	14586.55	110589.78	22.49	105.35	894.36	7616.39	56745.76					
Octomap	**6.49**	**32.79**	**181.97**	**2212.85**	14594.72	**10.87**	**57.29**	**340.08**	**4587.48**	31780.61	**8.66**	**44.24**	**247.52**	**2858.18**	18505.91					
Ours	18.19	75.05	282.98	2377.75	**11990.52**	26.36	135.79	620.75	4726.50	**24082.73**	20.80	89.68	402.93	3417.51	**15315.92**					
Workshop											*MainBuilding*									
Resolution (m)	1.0	0.5	0.25	0.1	0.05	0.025	0.01				1.0	0.5	0.25	0.1	0.05					
Grid Map	0.18	1.32	6.50	102.35	826.43	6719.52	109724.66				22.31	106.70	817.38	7328.23	31258.35					
Octomap	**0.11**	**0.52**	**2.81**	**27.39**	**149.20**	**803.47**	**10619.32**				**5.83**	**33.28**	261.84	3359.45	**17566.79**					
Ours	0.31	1.80	8.53	72.56	338.94	1516.09	11453.62				7.90	33.45	**187.56**	**3038.41**	26168.12					
FR_ 079											*Freiburg*									
Resolution (m)	1.0	0.5	0.25	0.1	0.05	0.025	0.01				1.0	0.5	0.25	0.1	0.05					
Grid Map	0.16	0.71	5.36	49.99	391.92	1921.08	26432.80				11.60	56.30	432.26	6594.09	30490.11					
Octomap	**0.08**	**0.34**	**1.64**	**14.32**	**99.17**	797.95	9991.86				**4.23**	**21.61**	**132.60**	2012.67	14453.00					
Ours	0.25	1.02	4.38	26.89	120.96	**600.16**	**3289.48**				11.00	45.67	189.19	**1721.62**	**14026.21**					

4.8 Real-world Applications

We demonstrate the high efficiency of D-Map in two applications that require real-time high-resolution occupancy mapping on a high-resolution LiDAR.

4.8.1 Interactive Guidance for High-resolution Real-time 3D Mapping

With the recent emergence of 3D applications such as metaverse [223, 224], virtual and augmented reality [225], and physical simulators [221, 226], the demand for accurate and detailed 3D reconstructions of real-world environments has increased. The accuracy and completeness of such reconstructions depend heavily on the quality of the data collection process. To overcome this challenge, we have developed an interactive guidance system that leverages our D-Map to achieve high-resolution real-time 3D mapping. The system offers users information on the explored and unexplored areas, along with suggestions for the next mapping region. By utilizing this information, users can avoid rescanning the same areas repeatedly or skipping any unscanned areas, thereby improving their overall work efficiency. A video demonstrating the use of the system is available on YouTube: youtu.be/m5QQPbkYYnA?t=251.

4.8.1.1 Experiment Setup

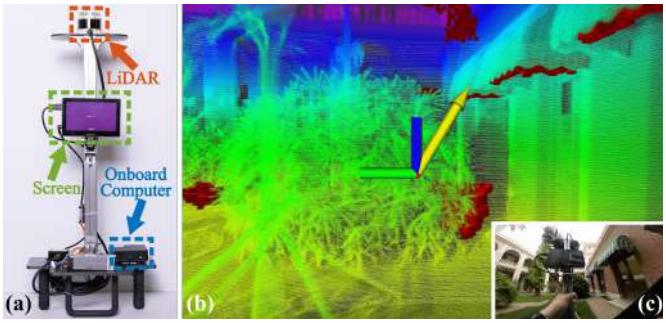


Figure 4.14: (a) Hardware setup of the handheld device for high-resolution 3D mapping, including an onboard computer (blue dashed block), a high-resolution LiDAR (orange dashed block), and a screen for online visualization (green dashed block). (b) A screenshot of the online visualization for interactively guiding the mapping process. The red cubes are the frontiers that users need to eliminate by scanning. The yellow arrow indicates the direction to the next suggested frontier for scanning. The depth measurements are accumulated and visualized by height value on the screen. (c) The first-person view of the user to conduct 3D mapping using our handheld device.

A handheld device is designed to conduct high-resolution 3D mapping, as shown in Fig. 4.14(a). The handheld device is equipped with an Intel NUC onboard computer with an i9-8550U CPU and 64 GB memory, and an OS1-128 LiDAR integrated with an IMU. The OS1-128 LiDAR can output $2,621,440$ high-precision point clouds per second with a maximum detection range of 120 m and a $360^\circ \times 45^\circ$ field of view. The frame rate of the OS1-128 LiDAR is set to 10 Hz.

We develop an interactive guidance system that consists of three modules: localization, mapping, and guidance. The localization module provides 6 DoF sensor pose estimation using our previous work FAST-LIO2 [219]. Additionally, the point cloud acquired at each frame is compensated for motion distortion in FAST-LIO2 and transformed from the LiDAR frame to the world frame. The mapping module leverages our D-Map to update the occupancy map in real-time using the estimated LiDAR pose and the currently registered point cloud. Finally, the guidance module visualizes the frontiers, which correspond to the unknown space adjacent to the free space in the occupancy map, and suggests a direction to the user for complete scanning, as shown in Fig. 4.14(b).

The experiment aims to reconstruct an area measuring 43 m×19 m×9 m located in Main Building, University of Hong Kong. We use a map resolution of 10 cm for occupancy mapping. The relax factor for D-Map is set to $\gamma = 1$ for full accuracy. In addition, we set the completeness threshold ε in Alg. 9 to 0.8, and the initial cell size `E` to 1.6 m.

4.8.1.2 Results

Table 4.5: Occupancy Mapping Performance in Interactive Guidance System

	Average Update Time	Processed Scan	Completion Rate[1]
D-Map	36.50 ms	9980	99.46%
SR&CR(Grid)	348.58 ms	2587	25.78%
SR&CR(Octo)	721.79 ms	1252	12.48%

[1] The completion rate is calculated as the ratio of the processed scan over the total scan. In this experiment, LiDAR generated 10034 scans of point clouds in total.

As Fig. 4.16(a) illustrates, the building reconstruction was completed with high fidelity. The mapping area was thoroughly scanned, except for an inaccessible classroom

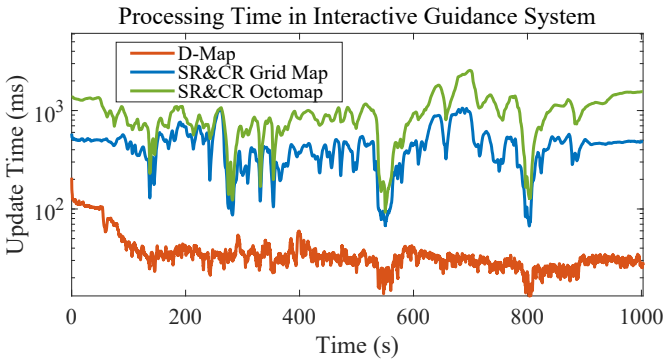


Figure 4.15: Comparison of the update time among our D-Map, SR&CR(Grid), and SR&CR(Octo). The update time of D-Map is acquired online in the interactive guidance system during the mapping process. The update time of the SR&CR(Grid) and SR&CR(Octo) is obtained by processing the recorded point clouds offline on the same computation platform.

located in the middle of the map.

We conducted a performance evaluation of our D-Map against the super ray and culling region-based methods (i.e., SR&CR(Grid) and SR&CR(Octo)) on the recorded LiDAR data file that contains 10,034 scans of LiDAR points in total. As shown in Fig. 4.15 and Table 4.5, D-Map achieved an average update time of only 36.50 ms to update the occupancy map, which is about 9.6 times faster than SR&CR(Grid) and 19.8 times faster than SR&CR(Octo). In addition, D-Map successfully processed the LiDAR data in real-time, except for the initial 54 s when the device was stationary. Consequently, D-Map processed 99.46 % of the total scan, generating a high-resolution occupancy map with high fidelity. In contrast, SR&CR(Grid) and SR&CR(Octo) failed to process in real-time, processing only 25.78 % and 12.48 % of the total scan, respectively. As a result, they missed a significant amount of environment information required for interactive guidance.

4.8.2 Autonomous UAV Exploration

Unmanned aerial vehicles (UAVs) are becoming increasingly popular for autonomous exploration and scanning of real-world environments due to their unrestricted flight view and accessibility to hard-to-reach locations, such as caves [227] and ancient remains [228]. However, the limited onboard computing power of UAVs presents a higher demand for efficient mapping modules compared to handheld devices. To address this demand, we have embedded our D-Map into a LiDAR-based UAV system, enabling it to

autonomously explore complex scenes with higher-resolution mapping. This integration enables UAVs to achieve real-time mapping and guidance, thereby providing high-fidelity results even in challenging environments.

4.8.2.1 Hardware System Setup

The UAV hardware platform includes an OS1-128 LiDAR, a Nora+ flight controller, and an onboard computer NUC 12 Pro Kit (i7-1260P CPU, maximum 4.70 GHz, 12-core, 32 GB RAM), as shown in Fig. 4.18(c). The extrinsic parameters between the LiDAR and the IMU on the flight controller are calibrated by LI-Init [229]. Additionally, we installed an action camera (DJI action 2) on the UAV to provide first-person view (FPV) images for better visualization. The small size of the UAV (40 cm×40 cm×21 cm), combined with its large detection range and high scanning density, makes it well-suited for exploration and scanning tasks in complex scenes.

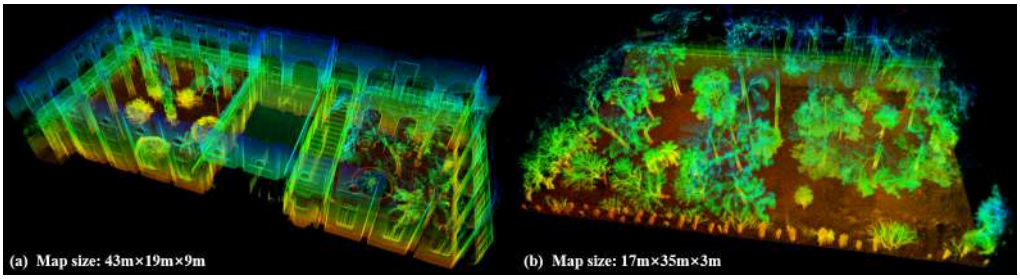


Figure 4.16: The high-fidelity point cloud map reconstructed from data collected by a high-resolution LiDAR. (a) Main Building in the University of Hong Kong, collected by a handheld device. (b) A forest in Hong Kong, autonomously collected by a UAV platform.

4.8.2.2 Software System Implementation

Multiple modules operate concurrently on the onboard computer to accomplish autonomous exploration tasks. The localization module employs FAST-LIO2 [219] to estimate the UAV's pose using data from the LiDAR and the IMU. This module generates undistorted point clouds, which are fed into the mapping module of the exploration planner. The mapping module utilizes D-Map for real-time occupancy mapping in place of the traditional ray-casting-based methods. The exploration planner, FUEL [12], reads the information in the mapping module and calculates efficient, collision-free trajectories

for exploration. These trajectories are tracked using an on-manifold model predictive controller [230].

In this experiment, we used the same parameters for D-Map as in Section 4.8.1. The maximum detection range of the LiDAR was set to 15 m to ease the computation load in the exploration planner.

4.8.2.3 Results

The real-flight autonomous exploration and scanning experiments were conducted in two complex scenarios: an abandoned fortress site built about 100 years ago (20.5 m×16 m×6 m) and a natural forest (17 m×35 m×3 m). The entire exploration process is available on youtu.be/m5QQPbkYYnA?t=78. We successfully carried out three flight experiments in each scenario. The UAV completed high-precision scanning in these complex environments, and the average flight time for autonomous exploration and scanning by the UAV was 123 s and 163 s, respectively. The acquired point clouds are shown in Fig. 4.18(a) and Fig. 4.16(b), exhibiting high precision and completeness.

We compared the running times of the mapping module employing D-Map and a uniform grid map utilizing ray-casting, which was the original method used in FUEL [12], on the same computing platform. As depicted in Fig. 4.17, D-Map substantially reduced the time required for the mapping module, thereby demonstrating its effectiveness and efficiency for real-time occupancy mapping on massive streaming point clouds of over two million per second.

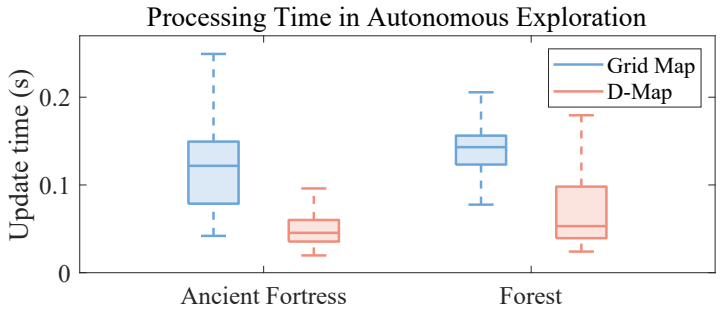


Figure 4.17: The comparison of processing time between a ray-casting-based grid map and our D-Map.

Figure 4.18: Our proposed framework D-Map served as a real-time high-resolution occupancy mapping module for an autonomous UAV exploration task in an ancient fortress. (a) The high-fidelity point cloud collected by UAV. (b) A bird-view of the scene. (c) The aerial platform carried a 128-channel LiDAR (OS1-128) to conduct the exploration task. The accompanying video of this paper is available on Youtube: youtu.be/m5QQPbkYYnA.

4.9 Extensions

4.9.1 Occupancy Mapping in Large-scale Environment

Table 4.6: Comparison of Update Time (ms) at Different Resolutions on a Large-scale Dataset

	ford_1				*ford_2*			
Resolution [m]	1.0	0.5	0.25	0.15	1.0	0.5	0.25	0.15
GridMap	21.0	42.7	153.4	×[1]	18.3	40.5	133.9	×
Octomap	186.8	305.5	570.9	1312.6	176.5	289.3	533.1	966.0
SR&CR(Grid)	24.7	64.1	233.2	×	22.9	58.9	208.2	×
SR&CR(Octo)	34.8	105.4	373.1	1197.9	30.2	103.6	391.8	×
Ours	**19.1**	**41.1**	**121.0**	**277.9**	**17.2**	**36.5**	**111.3**	**278.3**

[1] "×" denotes that this method failed due to exceeding memory limitation (i.e., 64 GB RAM + 64 GB swap memory).

To evaluate the performance of our D-Map in large-scale occupancy mapping, we conduct an experiment using two sequences from Ford Multi-AV Seasonal Dataset [231]. These two sequences, originally namely "2017-10-26-V2-Log1" and "2017-10-26-V2-Log2", are referred to as *ford_1* and *ford_2*, respectively. The mapping areas for these sequences measure 8090 m×11 494 m×96 m and 8107 m×11 659 m×103 m. We compare our D-Map with four benchmark methods, as discussed in Section 4.7. The corresponding results are presented in Table 4.6. The findings indicate that D-Map consistently outperforms the other methods across all resolutions in the context of large-scale applications. Notably, D-Map exhibits superior memory efficiency, enabling it to handle a resolution of 0.15 m within such expansive mapping areas. In contrast, grid-based maps (i.e., GridMap and SR&CR(Grid)) fail to meet the memory limitations imposed by the device.

4.9.2 Map Region Sliding for D-Map

In the context of occupancy mapping in even larger-scale environments, the issue of memory consumption becomes a significant concern. Therefore, we introduce a map region sliding technique in D-Map to address this challenge. This approach allows the removal of distant occupancy information from the current vehicle position. Fig. 4.19 illustrates the concept of map region sliding, wherein the mapping region of D-Map

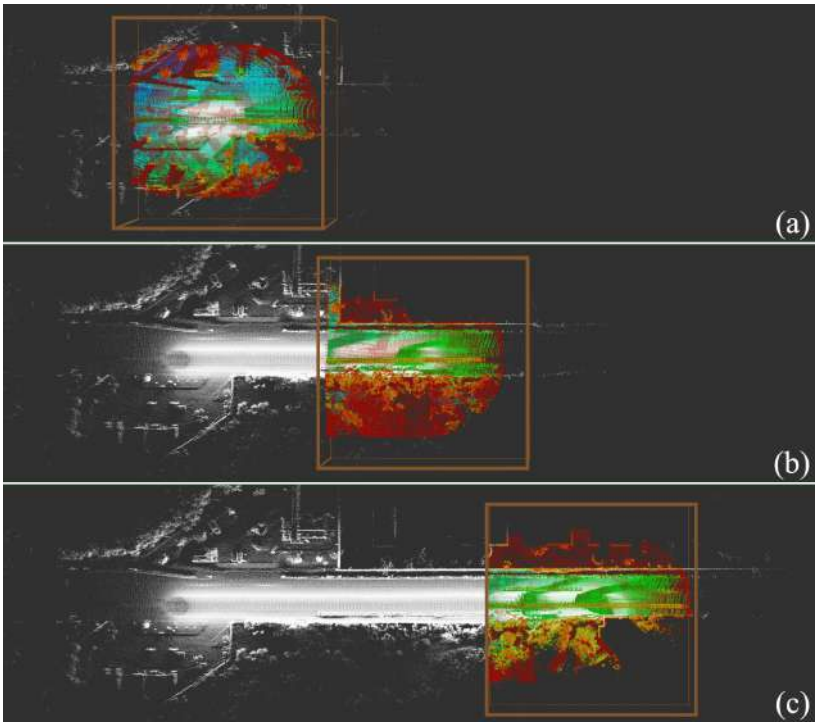


Figure 4.19: A demonstration of D-Map with map region sliding. Figures (a), (b), and (c) showcase the mapping process as the vehicle moves from left to right, with the mapping region sliding accordingly. The white point clouds represent the accumulated historical point clouds. The colored space within the mapping region of D-Map represents the occupancy information. An axis-aligned bounding box is employed to provide a visual representation of the mapping region, outlined by orange lines.

slides alongside the vehicle. The sliding process adjusts the mapping region when the vehicle has moved beyond a distance threshold and removes information outside of the mapping region from the D-Map. By dynamically updating the mapping region, D-Map optimizes memory usage and only retains the relevant occupancy information. This mechanism enables our method to handle large-scale environments while mitigating the impact of memory constraints.

4.9.3 Extension to Range Sensors with Measurement Noise

In D-Map, the on-tree update strategy leverages the high precision of depth measurements from LiDAR sensors to remove known cells on the octree directly. We further extend D-Map to range sensors with higher measurement noise (e.g., depth cameras) by incorporating occupancy probabilities. At each update, the occupancy probabilities on the correspondent nodes decrease for cells determined as known, while the occupancy probabilities in the corresponding cells of point clouds in the hashing grid map increase. The removal of D-Map is disabled to obtain correct occupancy probabilities. When querying the map, D-Map summarizes the occupancy probabilities from the hybrid map structure to determine occupancy states. This adaptation of occupancy probabilities appropriately handles measurement noise similar to other existing occupancy mapping approaches while retaining the superior efficiency of D-Map. We conduct several experiments to validate the performance of D-Map when incorporating occupancy probabilities.

4.9.3.1 Qualitative Evaluation

We conducted a qualitative comparison of the mapping results obtained from the original D-Map and D-Map with occupancy probability against those obtained from Octomap. To evaluate our approach, we used the *pioneer_slam3* sequence in the TUM dataset [232], which was captured by a Kinect depth camera mounted on a Pioneer robot. The original D-Map used the same parameters as those described in Section 4.7. For D-Map with occupancy probability and Octomap, the hit and miss probabilities by a ray for occupied and free space are set to 0.7 and 0.4, respectively, while the occupancy threshold for determining a cell as occupied is set to 0.9. The mapping results are presented in Fig. 4.20. The results of the original D-Map and D-Map with

occupancy probability (Fig. 4.20(a) and (b), respectively) highlight the effective handling of measurement noise from the depth camera. Furthermore, the mapping result obtained by D-Map with occupancy probability (Fig. 4.20(b)) exhibits few discrepancies from that produced by Octomap (Fig. 4.20(c)), indicating our accurate mapping performance.

4.9.3.2 Efficiency

We conduct benchmark experiments to evaluate our efficiency on depth cameras. We compare the update time of our original D-Map and D-Map with occupancy probability against the four mapping methods in Section 4.7, using four sequences in the TUM dataset [232]: *pioneer_360*, *pioneer_slam*, *pioneer_slam2*, and *pioneer_slam3*. The results are presented in Table 4.7, where the original D-Map and D-Map with occupancy probability are referred to as "Ours" and "Ours(prob)" respectively. Super Ray and Culling Region-based Grid Map (i.e., SR&CR(Grid)) performs the best at resolutions of 1.0 m and 0.5 m. However, at high resolutions (i.e., $d < 0.5$m), our original D-Map and D-Map with occupancy probability have superior performance. On average, the original D-Map and D-Map with occupancy probability achieve approximately 9.1 and 5.3 times faster than the fastest competing method (i.e., GridMap) at the resolution of 5 cm, respectively. The update time of D-Map with occupancy probability is comparable to that of the original D-Map at low resolutions (i.e., $d \geqslant 0.25$m). However, at high resolutions (i.e., $d \leqslant 0.1$m), D-Map with occupancy probability is slower than the original D-Map since a large number of cells are kept on the octree to maintain occupancy probabilities.

4.9.3.3 Accuracy

In addition to the previous experiments, we conduct simulation experiments to evaluate the accuracy of D-Map with occupancy probability. The simulation environment has a dimension of 30 m×20 m×4 m, as depicted in Fig. 4.21(a). We evaluate the Area Under the Receiver Operating Characteristic (AUROC) curves of D-Map with occupancy probability, Grid Map, and Octomap by removing different fractions of observed data, with results provided in Fig. 4.21(b). The results show that our D-Map with occupancy probability has a similar mapping performance as Octomap and Grid Map.

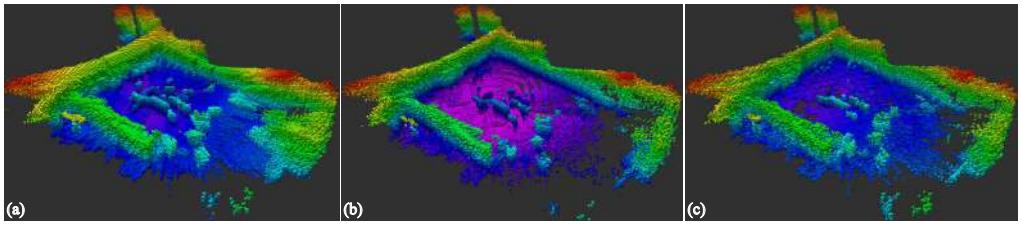


Figure 4.20: The mapping results of the sequence *pioneer_slam3* in TUM dataset. (a) The original D-Map (b) D-Map with occupancy probability. (c) Octomap

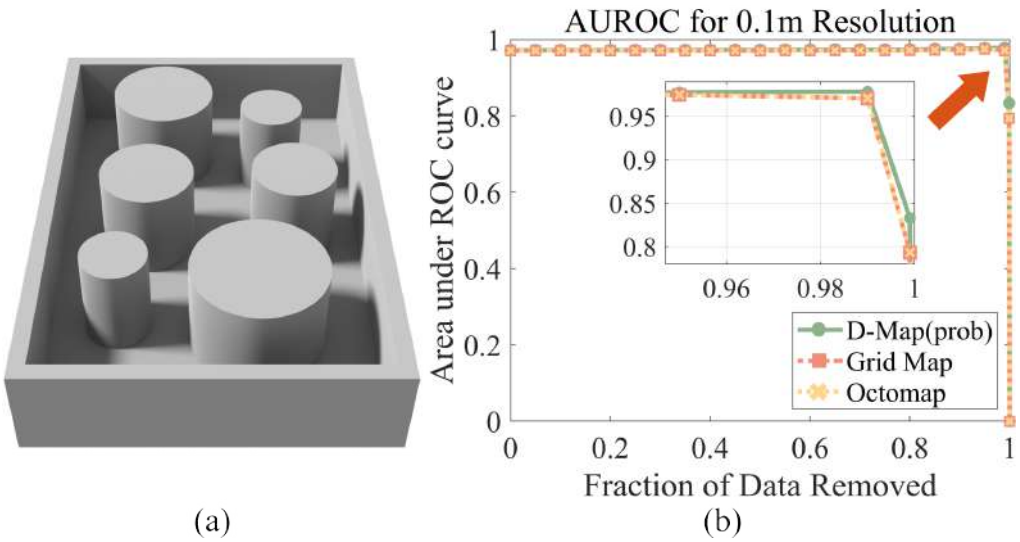


Figure 4.21: (a) The simulation environment for accuracy evaluation. (b) The AUROC curves of D-Map with occupancy probability, Grid Map, and Octomap for different fractions of removed data.

Table 4.7: Benchmark Comparison of Update Time (ms) at Different Resolution on TUM dataset

	pioneer_360					*pioneer_slam*					*pioneer_slam2*					*pioneer_slam3*				
Resolution [m]	1.0	0.5	0.25	0.1	0.05	1.0	0.5	0.25	0.1	0.05	1.0	0.5	0.25	0.1	0.05	1.0	0.5	0.25	0.1	0.05
GridMap	29.7	48.3	85.1	195.8	383.2	30.4	47.1	83.1	186.7	377.9	32.7	51.8	90.9	211.5	408.0	28.5	43.2	76.4	173.0	334.3
Octomap	93.3	154.0	273.3	611.9	1138.7	91.3	148.9	262.8	582.1	1135.2	106.5	179.4	318.4	699.7	1237.6	83.7	138.5	246.5	542.2	1001.4
SR&CR(Grid)	6.8	**10.2**	34.7	180.5	482.9	**7.0**	11.0	35.2	169.8	472.7	**7.5**	11.8	38.3	202.7	532.5	6.4	9.5	31.2	159.4	436.6
SR&CR(Octo)	**6.7**	10.3	31.1	192.7	485.9	7.1	**10.5**	31.3	184.7	514.7	7.6	**11.3**	35.2	213.1	508.0	**6.2**	**9.2**	28.0	172.4	464.6
Ours(prob)	26.3	25.7	**27.5**	37.6	73.9	29.0	27.7	**29.5**	38.5	70.3	28.6	29.2	**30.2**	39.6	70.8	25.3	26.1	**27.3**	36.0	66.5
Ours	27.0	26.8	27.8	**31.5**	**41.2**	30.5	28.9	29.9	**33.2**	**40.9**	30.3	30.4	30.7	**34.5**	**43.2**	26.8	27.3	27.7	**31.0**	**38.9**

4.9.4 Comparison against SuperEight

SuperEight [123] is a mapping system designed for efficient occupancy mapping featuring adaptive resolution. Similar to D-Map, this approach leverages depth image rasterization to accelerate occupancy updates. In this section, we conduct a comparative analysis between our proposed D-Map method and SuperEight using the Parkland sequence from the Newer College Dataset [233]. The LiDAR sensor used in the Parkland sequence is the Ouster OS1 with 64 channels, capable of generating 2,621,440 points per second. For our experimental evaluation, we utilized the publicly available implementation of SuperEight[3]. We present the experiment results in Table 4.8 which detailed the time consumption for updates at different resolution. The results show that the update performance of our D-Map surpasses that of SuperEight at high resolutions ($d \geqslant 0.25$m). Notably, SuperEight experienced a significant performance failure at the resolution of 0.05 meters, as the processing time for the complete sequence exceeded 12 hours. Conversely, at lower resolutions (d $<$ 0.25m), D-Map demonstrates slightly slower performance compared to SuperEight, primarily attributed to the preprocessing duration required for depth image rasterization and segment tree construction.

Table 4.8: Update Time (ms) on Parkland Sequence of Newer College Dataset

Resolution [m]	1.0	0.5	0.25	0.1	0.05
SuperEight	**5.60**	**20.41**	83.96	1004.90	-
Ours	15.12	26.99	**54.49**	**186.67**	**413.76**

"-" denotes that this method failed due to extremely long processing time for the entire sequence (over 12 hours).

In addition, we present the detailed update time at resolution of 0.25 m and 0.1 m, as shown in Figure 4.22. The figure depicts that, although D-Map initially exhibits a higher update time compared to SuperEight, its inherent decremental property rapidly reduces the update time, surpassing SuperEight and continuing to decrease throughout the entire mapping process. Moreover, the update time trend in D-Map remains more consistent across the two resolutions, whereas SuperEight encounters peaks in update time at the resolution of 0.1 m.

[3] https://supereight2.readthedocs.io/en/stable/

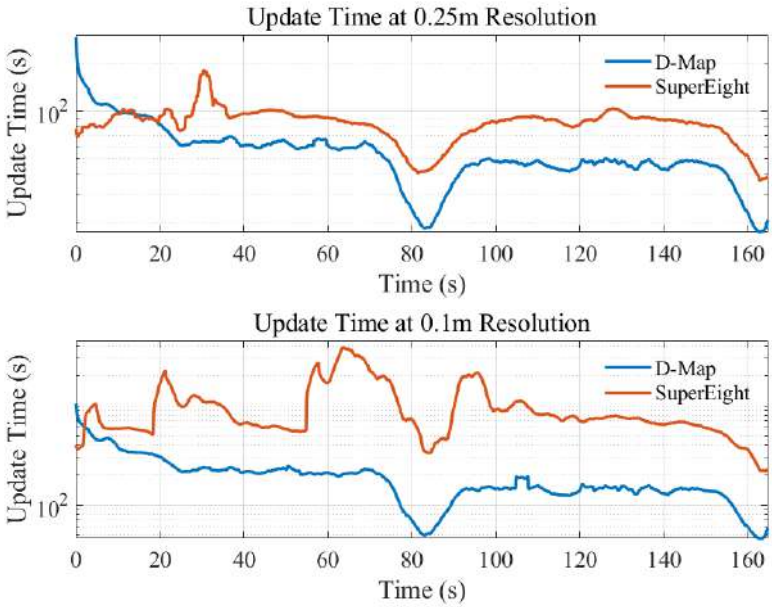


Figure 4.22: The detailed update time at resolution of 0.25 m and 0.1 m on Parkland Sequence of Newer College Dataset.

4.10 Discussion

In this section, we first discuss occupancy mapping using a depth image in terms of efficiency and accuracy, followed by the discussion on parallel processing over D-Map.

4.10.1 Occupancy Mapping on Depth Image

4.10.1.1 Efficiency

ray-casting is an indispensable component in the existing occupancy mapping framework for occupancy updates. However, the computational demands of ray-casting increase with the number of point clouds and the longer detection range of high-resolution LiDARs, which makes it unsuitable for computationally limited robotic applications. To alleviate the increasing computation load in occupancy mapping, we propose an alternative approach that leverages a depth image created from point clouds to determine occupancy states. Moreover, we design a novel on-tree occupancy update strategy that exploits the hierarchical structure of the octree to determine the occupancy state of larger cells, avoiding the need for timely updates on smaller cells as required by ray-casting-based methods. Additionally, we remove known cells from the octree, reducing the map size and further lowering the update cost. The significant reduction in both the

number of cells to be updated and the cost of map update renders substantially high efficiency in D-Map, as verified in the benchmark experiments in Section 4.7.

4.10.1.2 Accuracy

The accuracy of D-Map can be affected by two primary factors: depth map rasterization and occupancy state determination, which are both related to the depth map resolution.

In the rasterization process, using a low-resolution depth image might result in the loss of depth measurements since a pixel only keeps one point with the smallest depth, as described in Section 4.4.1. In the occupancy state determination process, D-Map determines occupancy states in spherical coordinates on the depth image rather than Cartesian coordinates. Several design choices affect the accuracy of D-Map, including 1) projecting cells onto a depth image using their circumsphere radius, which can result in information loss at the corners of the cells; 2) discretizing the projected area into pixels on a depth image, which can result in a distorted shape in 3D space; and 3) using an occupancy state determination method that allows for early determination of large regions if the observation completeness α surpasses a threshold ε, which may cause errors in the unobserved regions.

Despite a slight decrease in accuracy, D-Map provides comparable accuracy with the existing mapping approaches while achieving higher efficiency thanks to the comprehensive analysis and appropriate selection of depth image resolution.

4.10.2 Parallel Processing over D-Map

Although the primary focus during the design of D-Map does not prioritize parallel processing, certain aspects of the update process in D-Map lend themselves well to parallel implementation. Specifically, 1) The projection of a point cloud onto a depth image can be parallelized at the pixel level. 2) The update process of the hashing grid map naturally lends itself to parallel processing, allowing concurrent updates to different grid cells. However, it is worth noting that the octree structure employed in D-Map poses challenges for parallelization. Parallelizing the octree's operations is not straightforward

due to its hierarchical nature. As an alternative, we suggest utilizing a B-tree [234] to exploit parallel processing on efficient management of 3D data within D-Map.

4.11 Conclusion

This paper proposes a novel framework for occupancy mapping termed D-Map, which aims to provide efficient occupancy updates for high-resolution LiDAR sensors. Our proposed framework consists of three key techniques. Firstly, a method has been proposed to determine the occupancy state of a cell at arbitrary size through depth image projection. Secondly, a hybrid map structure has been developed with an efficient on-tree update strategy. Thirdly, a removal strategy has been introduced, which utilizes the low false alarm rate of LiDARs to remove known cells from the map. These techniques work in conjunction to reduce the number of cells that need updating and lower the cost of map updates, resulting in a significant improvement in efficiency.

To validate our proposed framework, we provide theoretical analyses of its accuracy and efficiency and conduct extensive benchmark experiments on various LiDAR datasets. The results show that D-Map substantially improves efficiency against other state-of-the-art mapping methods while maintaining comparable accuracy and high memory efficiency. Two real-world applications were demonstrated to showcase the effectiveness and efficiency of D-Map for high-resolution LiDAR-based applications.

Chapter 5

Conclusion and Future Work

5.1 Conclusions

This thesis primarily focuses on addressing the efficiency challenges in LiDAR mapping. To achieve higher efficiency in simultaneous localization and mapping (SLAM) and occupancy mapping, this thesis proposed new map representations and structures that leverage the strong sensing abilities of LiDARs. This, in turn, enhanced the autonomous ability of robots to conduct complex tasks in challenging environments.

Chapter 2 presented ikd-Tree, a data structure designed to manage sequential point clouds. The ikd-Tree supports various incremental functions, including point-wise and box-wise insertion, deletion, and re-insertion, that are suitable for robotic applications. The balance requirement of k-d trees is addressed through a double-thread re-balancing mechanism that actively monitors the balance status of (sub-)trees and partially rebuilds those that are unbalanced. The guarantee of tree balance ensures consistently high efficiency in incremental updates and nearest neighbor search, which is validated through theoretical time complexity analysis and benchmark comparison against static k-d trees.

Chapter 3 introduced a novel LiDAR-inertial odometry framework, FAST-LIO2, that exhibited high efficiency, accuracy, and robustness. FAST-LIO2 takes advantage of the powerful ikd-Tree to manage a point cloud map with efficient incremental updates and nearest neighbor search. The high efficiency of ikd-Tree enables FAST-LIO2 to directly register raw points into the map without feature extraction. This direct approach endows FAST-LIO2 with higher accuracy and robustness than existing methods as subtle

features of surrounding environments can be fully exploited. Furthermore, as no hand-engineered feature extraction is required, FAST-LIO2 is applicable to LiDAR sensors with different types of scanning patterns, making it a popular framework for providing accurate state estimation for robots. Exhaustive benchmark experiments validated the superior performance of FAST-LIO2 against state-of-the-art methods. We deployed FAST-LIO2 in both handheld and aerial platforms to demonstrate its effectiveness in providing accurate localization in agile motions and large-scale scenarios.

Chapter 4 proposed D-Map, an efficient occupancy mapping framework for high-resolution LiDAR sensors. D-Map avoids inefficient ray casting by projecting on a depth image to determine occupancy states. An on-tree update strategy is designed in collaboration with the projection-based occupancy state determination method to reduce redundant visits to cells, improving efficiency in occupancy updates. Furthermore, D-Map takes advantage of the low false alarm rate of LiDAR sensors to directly remove known cells from the map. This approach endows D-Map with a decremental property, leading to further improved efficiency in both memory and computation due to the decreasing map size. The design of D-Map was validated through rigorous theoretical analysis in both accuracy and efficiency, accompanied by extensive benchmark experiments against existing occupancy grid mapping methods. The results demonstrated that D-Map achieved significant improvements in both computational and memory efficiency while maintaining comparable mapping accuracy. This thesis also demonstrated deployments of D-Map in real-world applications to support occupancy grid mapping for high-resolution LiDAR sensors in real-time.

5.2 Discussion of Limitations

This thesis proposed different designs of map representations and structures aimed at enhancing the efficiency of LiDAR SLAM and occupancy mapping. However, while these designs demonstrated excellent performance, we acknowledge two limitations in our methods.

Firstly, the proposed mapping approaches lack the ability to actively correct previous maps, which is critical in maintaining a consistent map when faced with accumulated odometry drift and dynamic, long-term scenarios. Although ikd-Tree in Chapter 2

supports various types of incremental functions that allow for map correction, FAST-LIO2 assumes a perfect point cloud map without considering potential inconsistencies and inaccuracies. Since FAST-LIO2 is a front-end framework without loop closure, accumulated errors in odometry can lead to an inconsistent contaminated map, which deteriorates localization accuracy. Similarly, in Chapter 4, D-Map assumes perfect pose estimation and static environments, which limits its ability to correct inaccuracies in mapping, particularly in dynamic scenarios.

Secondly, the proposed mapping approaches were designed for serial processing without careful consideration of their possible extension to parallel processing in the future. Although query functions in Chapter 2 can be easily paralleled, the incremental updates are limited to a single-thread manner due to the binary tree structure and the need for re-balancing. Therefore, FAST-LIO2 in Chapter 3 follows a serial processing manner in updating the point cloud map, while the nearest neighbor search is done in a parallel manner. Similarly, in Chapter 4, D-Map utilizes an octree to allow efficient on-tree updates, in which the tree structure also limits the extension to parallel processing.

5.3 Future Work

Motivated by the aforementioned limitations, this thesis proposes three potential directions for future research in LiDAR mapping.

5.3.1 Consistent LiDAR Mapping

Previous research has sufficiently investigated the inconsistency of LiDAR mapping resulting from inaccurate pose estimation through loop closure detection [235–238] and LiDAR bundle adjustment [46, 65, 195]. These methods leverage a strong human prior in the geometric structures and rigorous mathematical derivation to guarantee convergence in accuracy. Another branch of research has been working on learning-based approaches, which aims to extract deep features that are difficult for humans to understand to facilitate feature matching and place recognition [239–242]. There is still great potential to exploit the combination of these two branches of research. For instance, inspired by

the idea of Gaussian Splatting [243], we could easily capture the geometric features using mathematical representations and leverage learning-based methods to train suitable parameters within the representation and exploit an effective matching algorithm.

Achieving consistent LiDAR mapping also requires the identification of static and dynamic objects, with the former used for scene reconstruction and the latter for object tracking and prediction. Research in detecting dynamic objects has explored explicit designs for pattern detection [244, 245] as well as learning-based methods [246–248]. However, these approaches face significant challenges when adapting to various types of LiDAR sensors, either due to the need for handcrafted parameter tuning or limited generality. Nevertheless, considering that the dynamic nature of objects is inherent to their physical properties (e.g., a car is movable, whether moving or not), we believe that learning-based methods will emerge as the dominant paradigm in the future. Deep features, such as semantic information, can be learned and encoded by training on large datasets, enabling the robust detection of dynamic objects across different LiDAR sensor types.

While the previous discussion focuses on the spatial aspect of consistent mapping, long-term consistency is also crucial to address temporal changes in the environment, such as seasonal variations [231]. With temporal changes, even if the pose estimation is correct, the robot may encounter sensor measurements that are inconsistent with the historical map due to real-world scenario differences. In such situations, the mapping system should have the ability to determine whether to correct the information in the map by assessing the uncertainty in sensor measurements, pose estimation, and the map itself.

5.3.2 Efficient Mapping on Heterogeneous Platform

A significant portion of existing research in LiDAR mapping has primarily focused on CPU platforms, which employ serial computing methods to process sensor measurements. However, as LiDAR sensors produce increasingly large amounts of data, computation times increase linearly or even polynomially. This poses a challenge for mobile robots with limited computational power as they may struggle to process the growing

amount of LiDAR data efficiently. To address this issue, researchers have developed various simplification techniques, such as voxel down-sampling and feature extraction, that trade off algorithm efficiency on low-power mobile robots with accuracy and robustness. However, these approaches are only partial solutions to the problem at hand and do not address the growing gap between the demands of processing increasing amount of LiDAR measurements and the limited computational capabilities of CPUs.

The clock speed of CPUs has been limited to around 4 GHz for a long time. From the 1950's to the 2000's, the clock speed doubled every 1-2 years, following the prediction of Moore's Law [249]. However, this trend has ceased to hold true in recent years due to Dennard Scaling [250], which states that the power density of transistors remains constant as their size decreases. As transistors have now reached the nanometer level, manufacturers face significant challenges in striking a balance between size, power consumption, and heat generation. Consequently, it is time to seek change and explore new computational structures for the future of mapping.

Inspired by the insights of renowned scientists, we turn our attention to heterogeneous computational structures. Herb Sutter, the author of "The Free Lunch is Over" which explained the transition from single-core CPUs to multi-core CPUs, stated in his 2012 article "Welcome to the Jungle" [251] that "Hence, a single compute-intensive application will need to harness different kinds of cores, in immense numbers, to get its job done". Similarly, Andrew Davison noted in [24] that future mapping must incorporate parallel and heterogeneous computation, predicting it to be the dominant paradigm for practical systems. Therefore, we believe that, moving the heavy computational pressure in LiDAR mapping from CPUs to parallel, heterogeneous, specialized processing units would offer significant potential to meet the rapidly increasing demands for computational resources.

In general, Emerging components of a heterogeneous computing structure could include computation units for both serial processing (CPUs) and parallel processing (IPUs), as well as specialized units for graphics processing (GPUs), tensor computation (TPUs), and neural network inference (NPUs). To illustrate initial designs for LiDAR mapping on heterogeneous computation units, we use NVIDIA Jetson modules as an example, which are embedded computation systems specialized for edge computing in

robotics [252]. The Jetson platforms include an ARM architecture CPU and a GPU. In this context, the mapping module is ideally implemented fully using the GPU in parallel, allowing for concurrent access, updates, and queries to the map. Furthermore, the parallel computation ability facilitates the summarization of information from the mapping module (e.g., parallel reduction sum of residuals). The serial computation unit leverages the summarized results to make decisions while managing the data communication among the mapping module and other serial modules.

5.3.3 Multi-modal Collaborative Mapping

Multi-modal mapping involves integrating visual sensors, such as RGB cameras and infrared cameras, with LiDAR sensors to achieve a more comprehensive understanding of the environment. While LiDAR sensors excel at reconstructing the geometry of the environment, extracting semantic information directly from depth measurements can be challenging. Visual sensors, on the other hand, are well-suited for capturing semantic information. However, when investigating state-of-the-art learning-based methods, the majority of these approaches concentrate on analyzing semantic information from a single image input or a series of historical image inputs. A more effective solution could be to combine implicit features from image encoders with accurate 3D geometry representations from LiDARs. The fundamental principle underlying this approach to multi-modal mapping revolves around creating a comprehensive map representation that effectively accommodates information from all modalities.

The information provided by a single agent is often limited due to constraints on payload, travel distance, and traversability. To overcome these limitations, we propose collaborative mapping, which integrates information from heterogeneous agents (e.g., UAVs, legged robots) equipped with different sensors to build a more efficient and comprehensive mapping system. Previous research in multi-agent robotic systems [16, 253, 254] has demonstrated the advantages of decentralized architectures. Similarly, collaborative mapping should leverage the same ethos by managing representations of the environment through decentralized mapping modules and refining maps through communication among agents. The decentralized architecture also allows for distributed computation, which extracts deep features from raw sensor measurements, reducing the computational load of map synthesis in other agents and facilitating post-processing in

cloud computation platforms. Designing a decentralized and distributed collaborative mapping system requires determining the level of abstraction to be computed in each agent and shared via communication, considering task difficulties as well as available computational and memory resources.

References

[1] E. van Henten, C. Montenegro, M. Popovic, S. Vougioukas, A. Daniel, and G. Han. "Embracing Robotics and Intelligent Machine Systems for Smart Agricultural Applications [From the Guest Editors]". In: *IEEE Robotics & Automation Magazine* 30.4 (2023), pp. 8–112.

[2] J. Weyler, T. Läbe, J. Behley, and C. Stachniss. "Panoptic Segmentation With Partial Annotations for Agricultural Robots". In: *IEEE Robotics and Automation Letters* 9.2 (2024), pp. 1660–1667.

[3] W. Tabib, K. Goel, J. Yao, C. Boirum, and N. Michael. "Autonomous cave surveying with an aerial robot". In: *IEEE Transactions on Robotics* 38.2 (2021), pp. 1016–1032.

[4] D. Baril, S.-P. Deschênes, O. Gamache, M. Vaidis, D. LaRocque, J. Laconte, V. Kubelka, P. Giguère, and F. Pomerleau. "Kilometer-scale autonomous navigation in subarctic forests: challenges and lessons learned". In: *arXiv preprint arXiv:2111.13981* (2021).

[5] M. Chiou, G.-T. Epsimos, G. Nikolaou, P. Pappas, G. Petousakis, S. Mühl, and R. Stolkin. "Robot-assisted nuclear disaster response: Report and insights from a field exercise". In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2022, pp. 4545–4552.

[6] *Mars Helicopter*. science.nasa.gov/mission/mars-2020-perseverance/ingenuity-mars-helicopter. [Online; accessed 25-May-2024].

[7] P. Arm, G. Waibel, J. Preisig, T. Tuna, R. Zhou, V. Bickel, G. Ligeza, T. Miki, F. Kehl, H. Kolvenbach, et al. "Scientific exploration of challenging planetary analog environments with a team of legged robots". In: *Science robotics* 8.80 (2023), eade9548.

[8] C. Campos, R. Elvira, J. J. G. Rodrıéguez, J. M. Montiel, and J. D. Tardós. "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam". In: *IEEE Transactions on Robotics* 37.6 (2021), pp. 1874–1890.

[9] R. Mur-Artal and J. D. Tardós. "Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras". In: *IEEE transactions on robotics* 33.5 (2017), pp. 1255–1262.

[10] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos. "ORB-SLAM: a versatile and accurate monocular SLAM system". In: *IEEE transactions on robotics* 31.5 (2015), pp. 1147–1163.

[11] T. Qin, P. Li, and S. Shen. "Vins-mono: A robust and versatile monocular visual-inertial state estimator". In: *IEEE Transactions on Robotics* 34.4 (2018), pp. 1004–1020.

[12] B. Zhou, Y. Zhang, X. Chen, and S. Shen. "FUEL: Fast UAV exploration using incremental frontier structure and hierarchical planning". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 779–786.

[13] B. Zhou, H. Xu, and S. Shen. "Racer: Rapid collaborative exploration with a decentralized multi-uav system". In: *IEEE Transactions on Robotics* (2023).

[14] X. Zhou, Z. Wang, H. Ye, C. Xu, and F. Gao. "Ego-planner: An esdf-free gradient-based local planner for quadrotors". In: *IEEE Robotics and Automation Letters* 6.2 (2020), pp. 478–485.

[15] A. Rosinol, A. Violette, M. Abate, N. Hughes, Y. Chang, J. Shi, A. Gupta, and L. Carlone. "Kimera: From SLAM to spatial perception with 3D dynamic scene graphs". In: *The International Journal of Robotics Research* 40.12-14 (2021), pp. 1510–1546.

[16] Y. Tian, Y. Chang, F. H. Arias, C. Nieto-Granda, J. P. How, and L. Carlone. "Kimera-multi: Robust, distributed, dense metric-semantic slam for multi-robot systems". In: *IEEE Transactions on Robotics* 38.4 (2022).

[17] A. Asgharivaskasi and N. Atanasov. "Semantic OcTree Mapping and Shannon Mutual Information Computation for Robot Exploration". In: *IEEE Transactions on Robotics* 39.3 (2023), pp. 1910–1928. DOI: 10.1109/TRO.2023.3245986.

[18] S. Peng, K. Genova, C. Jiang, A. Tagliasacchi, M. Pollefeys, T. Funkhouser, et al. "Openscene: 3d scene understanding with open vocabularies". In: *Proceedings of*

the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023, pp. 815–824.

[19] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, et al. "Stanley: The robot that won the DARPA Grand Challenge". In: *Journal of field Robotics* 23.9 (2006), pp. 661–692.

[20] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al. "Autonomous driving in urban environments: Boss and the urban challenge". In: *Journal of Field Robotics* 25.8 (2008), pp. 425–466.

[21] Z. Liu, F. Zhang, and X. Hong. "Low-cost retina-like robotic lidars based on incommensurable scanning". In: *IEEE/ASME Transactions on Mechatronics* 27.1 (2021), pp. 58–68.

[22] Y. Li and J. Ibanez-Guzman. "Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems". In: *IEEE Signal Processing Magazine* 37.4 (2020), pp. 50–61.

[23] *Livox MID-360*. https://www.livoxtech.com/mid-360. [Online; accessed 26-May-2024].

[24] A. J. Davison. "FutureMapping: The computational structure of spatial AI systems". In: *arXiv preprint arXiv:1803.11288* (2018).

[25] H. D. Whyte. "Simultaneous localisation and mapping (SLAM): Part I the essential algorithms". In: *Robotics and Automation Magazine* (2006).

[26] D. Durrant-Whyte Hugh and Rye and E. Nebot. "Localization of Autonomous Guided Vehicles". In: *Robotics Research*. London: Springer London, 1996, pp. 613–625.

[27] J. A. Castellanos, J. M. Martıénez, J. Neira, and J. D. Tardós. "Experiments in multisensor mobile robot localization and map building". In: *IFAC Proceedings Volumes* 31.3 (1998), pp. 369–374.

[28] N. Ayache and O. D. Faugeras. "Building, registrating, and fusing noisy visual maps". In: *The International Journal of Robotics Research* 7.6 (1988), pp. 45–65.

[29] J. A. Castellanos, J. D. Tardós, and G. Schmidt. "Building a global map of the environment of a mobile robot: The importance of correlations". In: *Proceedings*

of International Conference on Robotics and Automation. Vol. 2. IEEE. 1997, pp. 1053–1059.

[30] J. Guivant, E. Nebot, and S. Baiker. "Localization and map building using laser range sensors in outdoor applications". In: *Journal of robotic systems* 17.10 (2000), pp. 565–583.

[31] J. J. Leonard and H. J. S. Feder. "A computationally efficient method for large-scale concurrent mapping and localization". In: *Robotics Research: The Ninth International Symposium*. Springer. 2000, pp. 169–176.

[32] K. S. Chong and L. Kleeman. "Feature-based mapping in real, large scale environments using an ultrasonic array". In: *The International Journal of Robotics Research* 18.1 (1999), pp. 3–19.

[33] J. Zhang and S. Singh. "LOAM: Lidar Odometry and Mapping in Real-time." In: *Robotics: Science and Systems*. Vol. 2. 9. 2014.

[34] J. A. Hesch, F. M. Mirzaei, G. L. Mariottini, and S. I. Roumeliotis. "A Laser-aided Inertial Navigation System (L-INS) for human localization in unknown indoor environments". In: *2010 IEEE International Conference on Robotics and Automation*. IEEE. 2010, pp. 5376–5382.

[35] Z. Cheng, D. Liu, Y. Yang, T. Ling, X. Chen, L. Zhang, J. Bai, Y. Shen, L. Miao, and W. Huang. "Practical phase unwrapping of interferometric fringes based on unscented Kalman filter technique". In: *Optics express* 23.25 (2015), pp. 32337–32349.

[36] W. Zhen, S. Zeng, and S. Soberer. "Robust localization and localizability estimation with a rotating laser scanner". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 6240–6245.

[37] X. Zuo, P. Geneva, W. Lee, Y. Liu, and G. Huang. "LIC-Fusion: LiDAR-Inertial-Camera Odometry". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 5848–5854. DOI: 10.1109/IROS40897.2019.8967746.

[38] C. Qin, H. Ye, C. E. Pranata, J. Han, S. Zhang, and M. Liu. "LINS: A Lidar-Inertial State Estimator for Robust and Efficient Navigation". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 8899–8906.

[39] P. Geneva, K. Eckenhoff, Y. Yang, and G. Huang. "LIPS: Lidar-inertial 3d plane slam". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 123–130.

[40] H. Ye, Y. Chen, and M. Liu. "Tightly coupled 3d lidar inertial odometry and mapping". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 3144–3150.

[41] F. Moosmann and C. Stiller. "Velodyne slam". In: *2011 ieee intelligent vehicles symposium (iv)*. IEEE. 2011, pp. 393–398.

[42] J. Zhang and S. Singh. "Low-drift and real-time lidar odometry and mapping". In: *Autonomous Robots* 41 (2017), pp. 401–416.

[43] T. Shan and B. Englot. "Lego-loam: Lightweight and ground-optimized lidar odometry and mapping on variable terrain". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 4758–4765.

[44] L. Zhou, D. Koppel, and M. Kaess. "LiDAR SLAM with plane adjustment for indoor environment". In: *IEEE Robotics and Automation Letters* 6.4 (2021), pp. 7073–7080.

[45] L. Zhou, S. Wang, and M. Kaess. "π-LSAM: LiDAR smoothing and mapping with planes". In: *2021 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2021, pp. 5751–5757.

[46] Z. Liu and F. Zhang. "Balm: Bundle adjustment for lidar mapping". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3184–3191.

[47] L. Zhou, G. Huang, Y. Mao, J. Yu, S. Wang, and M. Kaess. "PLC-LiSLAM: LiDAR SLAM With Planes, Lines, and Cylinders". In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 7163–7170.

[48] C. Chen, H. Wu, Y. Ma, J. Lv, L. Li, and Y. Liu. "LiDAR-Inertial SLAM with Efficiently Extracted Planes". In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2023, pp. 1497–1504.

[49] K. Li, M. Li, and U. D. Hanebeck. "Towards High-Performance Solid-State-LiDAR-Inertial Odometry and Mapping". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 5167–5174. DOI: 10.1109/LRA.2021.3070251.

[50] J. Lin and F. Zhang. "Loam livox: A fast, robust, high-precision LiDAR odometry and mapping package for LiDARs of small FoV". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 3126–3131.

[51] H. Pfister, M. Zwicker, J. Van Baar, and M. Gross. "Surfels: Surface elements as rendering primitives". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 335–342.

[52] T. Weise, T. Wismer, B. Leibe, and L. Van Gool. "Online loop closure for real-time interactive 3D scanning". In: *Computer Vision and Image Understanding* 115.5 (2011), pp. 635–648.

[53] M. Keller, D. Lefloch, M. Lambers, S. Izadi, T. Weyrich, and A. Kolb. "Real-time 3d reconstruction in dynamic scenes using point-based fusion". In: *2013 International Conference on 3D Vision-3DV 2013*. IEEE. 2013, pp. 1–8.

[54] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. "Real-time 3D reconstruction at scale using voxel hashing". In: *ACM Transactions on Graphics (ToG)* 32.6 (2013), pp. 1–11.

[55] R. F. Salas-Moreno, B. Glocken, P. H. Kelly, and A. J. Davison. "Dense planar SLAM". In: *2014 IEEE international symposium on mixed and augmented reality (ISMAR)*. IEEE. 2014, pp. 157–164.

[56] T. Whelan, M. Kaess, H. Johannsson, M. Fallon, J. J. Leonard, and J. McDonald. "Real-time large-scale dense RGB-D SLAM with volumetric fusion". In: *The International Journal of Robotics Research* 34.4-5 (2015), pp. 598–626.

[57] T. Whelan, S. Leutenegger, R. F. Salas-Moreno, B. Glocker, and A. J. Davison. "ElasticFusion: Dense SLAM without a pose graph." In: *Robotics: science and systems*. Vol. 11. Rome, Italy. 2015, p. 3.

[58] J. Behley and C. Stachniss. "Efficient Surfel-Based SLAM using 3D Laser Range Data in Urban Environments." In: *Robotics: Science and Systems*. Vol. 2018. 2018, p. 59.

[59] D. Droeschel, M. Schwarz, and S. Behnke. "Continuous mapping and localization for autonomous navigation in rough terrain using a 3D laser scanner". In: *Robotics and Autonomous Systems* 88 (2017), pp. 104–115.

[60] J. Quenzel and S. Behnke. "Real-time multi-adaptive-resolution-surfel 6D LiDAR odometry using continuous-time trajectory optimization". In: *2021 IEEE/RSJ*

International Conference on Intelligent Robots and Systems (IROS). IEEE. 2021, pp. 5499–5506.

[61] C. Park, S. Kim, P. Moghadam, C. Fookes, and S. Sridharan. "Probabilistic surfel fusion for dense LiDAR mapping". In: *Proceedings of the IEEE International Conference on Computer Vision Workshops*. 2017, pp. 2418–2426.

[62] C. Park, P. Moghadam, J. L. Williams, S. Kim, S. Sridharan, and C. Fookes. "Elasticity meets continuous-time: Map-centric dense 3D LiDAR SLAM". In: *IEEE Transactions on Robotics* 38.2 (2021), pp. 978–997.

[63] C. Yuan, W. Xu, X. Liu, X. Hong, and F. Zhang. "Efficient and probabilistic adaptive voxel mapping for accurate online lidar odometry". In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 8518–8525.

[64] C. Wu, Y. You, Y. Yuan, X. Kong, Y. Zhang, Q. Li, and K. Zhao. "VoxelMap++: Mergeable Voxel Mapping Method for Online LiDAR (-Inertial) Odometry". In: *IEEE Robotics and Automation Letters* 9.1 (2023), pp. 427–434.

[65] Z. Liu, X. Liu, and F. Zhang. "Efficient and consistent bundle adjustment on lidar point clouds". In: *IEEE Transactions on Robotics* (2023).

[66] X. Chen, I. Vizzo, T. Läbe, J. Behley, and C. Stachniss. "Range image-based LiDAR localization for autonomous vehicles". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 5802–5808.

[67] S. Pütz, T. Wiemann, and J. Hertzberg. "The mesh tools package–introducing annotated 3d triangle maps in ros". In: *Robotics and Autonomous Systems* 138 (2021), p. 103688.

[68] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. "Kinectfusion: Real-time dense surface mapping and tracking". In: *2011 10th IEEE international symposium on mixed and augmented reality*. Ieee. 2011, pp. 127–136.

[69] W. E. Lorensen and H. E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *Seminal graphics: pioneering efforts that shaped the field*. 1998, pp. 347–353.

[70] J. Chen, D. Bautembach, and S. Izadi. "Scalable real-time volumetric surface reconstruction." In: *ACM Trans. Graph.* 32.4 (2013), pp. 113–1.

[71] O. Kähler, V. Prisacariu, J. Valentin, and D. Murray. "Hierarchical voxel block hashing for efficient integration of depth images". In: *IEEE Robotics and Automation Letters* 1.1 (2015), pp. 192–197.

[72] E. Vespa, N. Nikolov, M. Grimm, L. Nardi, P. H. Kelly, and S. Leutenegger. "Efficient octree-based volumetric SLAM supporting signed-distance and occupancy mapping". In: *IEEE Robotics and Automation Letters* 3.2 (2018), pp. 1144–1151.

[73] O. Kähler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. Torr, and D. Murray. "Very high frame rate volumetric integration of depth images on mobile devices". In: *IEEE transactions on visualization and computer graphics* 21.11 (2015), pp. 1241–1250.

[74] M. Klingensmith, I. Dryanovski, S. S. Srinivasa, and J. Xiao. "Chisel: Real Time Large Scale 3D Reconstruction Onboard a Mobile Device using Spatially Hashed Signed Distance Fields." In: *Robotics: science and systems*. Vol. 4. 1. Citeseer. 2015.

[75] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto. "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 1366–1373.

[76] M. Kazhdan, M. Bolitho, and H. Hoppe. "Poisson surface reconstruction". In: *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7. 4. 2006.

[77] I. Vizzo, X. Chen, N. Chebrolu, J. Behley, and C. Stachniss. "Poisson surface reconstruction for LiDAR odometry and mapping". In: *2021 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2021, pp. 5624–5630.

[78] J. Lin, C. Yuan, Y. Cai, H. Li, Y. Ren, Y. Zou, X. Hong, and F. Zhang. "Immesh: An immediate lidar localization and meshing framework". In: *IEEE Transactions on Robotics* (2023).

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

[80] M. Magnusson, A. Lilienthal, and T. Duckett. "Scan registration for autonomous mining vehicles using 3D-NDT". In: *Journal of Field Robotics* 24.10 (2007), pp. 803–827.

[81] M. Magnusson. "The three-dimensional normal-distributions transform: an efficient representation for registration, surface analysis, and loop detection". PhD thesis. Örebro universitet, 2009.

[82] S. Zhao, Z. Fang, H. Li, and S. Scherer. "A robust laser-inertial odometry and mapping method for large-scale highway environments". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2019, pp. 1285–1292.

[83] M. Yokozuka, K. Koide, S. Oishi, and A. Banno. "LiTAMIN2: Ultra light LiDAR-based SLAM using geometric approximation applied with KL-divergence". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 11619–11625.

[84] M. Magnusson, A. Nuchter, C. Lorken, A. J. Lilienthal, and J. Hertzberg. "Evaluation of 3D registration reliability and speed-A comparison of ICP and NDT". In: *2009 IEEE International Conference on Robotics and Automation*. IEEE. 2009, pp. 3907–3912.

[85] R. W. Wolcott and R. M. Eustice. "Fast LIDAR localization using multiresolution Gaussian mixture maps". In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, pp. 2814–2821.

[86] R. W. Wolcott and R. M. Eustice. "Robust LIDAR localization using multiresolution Gaussian mixture maps for autonomous driving". In: *The International Journal of Robotics Research* 36.3 (2017), pp. 292–319.

[87] B. Eckart, K. Kim, A. Troccoli, A. Kelly, and J. Kautz. "Accelerated generative models for 3D point cloud data". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 5497–5505.

[88] S. Srivastava and N. Michael. "Approximate continuous belief distributions for precise autonomous inspection". In: *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE. 2016, pp. 74–80.

[89] X. Chen, A. Milioto, E. Palazzolo, P. Giguere, J. Behley, and C. Stachniss. "Suma++: Efficient lidar-based semantic slam". In: *2019 IEEE/RSJ International*

Conference on Intelligent Robots and Systems (IROS). IEEE. 2019, pp. 4530–4537.

[90] A. Milioto, I. Vizzo, J. Behley, and C. Stachniss. "Rangenet++: Fast and accurate lidar semantic segmentation". In: *2019 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2019, pp. 4213–4220.

[91] C. Bai, T. Xiao, Y. Chen, H. Wang, F. Zhang, and X. Gao. "Faster-LIO: Lightweight tightly coupled LiDAR-inertial odometry using parallel sparse incremental voxels". In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 4861–4868.

[92] J. L. Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[93] C. L. Jackins and S. L. Tanimoto. "Oct-trees and their use in representing three-dimensional objects". In: *Computer Graphics and Image Processing* 14.3 (1980), pp. 249–270.

[94] J. L. Vermeulen, A. Hillebrand, and R. Geraerts. "A comparative study of k-nearest neighbour techniques in crowd simulation". In: *Computer Animation and Virtual Worlds* 28.3-4 (2017), e1775.

[95] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.

[96] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. "OctoMap: An efficient probabilistic 3D mapping framework based on octrees". In: *Autonomous robots* 34.3 (2013), pp. 189–206.

[97] A. Elfes. "Sonar-based real-world mapping and navigation". In: *IEEE Journal on Robotics and Automation* 3.3 (1987), pp. 249–265.

[98] H. P. Moravec. "Sensor fusion in certainty grids for mobile robots". In: *AI magazine* 9.2 (1988), pp. 61–61.

[99] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto. "Voxblox: Incremental 3D Euclidean Signed Distance Fields for on-board MAV planning". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 1366–1373. DOI: 10.1109/IROS.2017.8202315.

[100] L. Han, F. Gao, B. Zhou, and S. Shen. "FIESTA: Fast Incremental Euclidean Distance Fields for Online Motion Planning of Aerial Robots". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 4423–4430. DOI: 10.1109/IROS40897.2019.8968199.

[101] S. T. O'Callaghan and F. T. Ramos. "Gaussian process occupancy maps". In: *The International Journal of Robotics Research* 31.1 (2012), pp. 42–62.

[102] M. Veeck and W. Veeck. "Learning polyline maps from range scan data acquired with mobile robots". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*. Vol. 2. IEEE. 2004, pp. 1065–1070.

[103] M. Paskin and S. Thrun. "Robotic mapping with polygonal random fields". In: *arXiv preprint arXiv:1207.1399* (2012).

[104] S. Kim and J. Kim. "GPmap: A unified framework for robotic mapping based on sparse Gaussian processes". In: *Field and service robotics*. Springer. 2015, pp. 319–332.

[105] J. Wang and B. Englot. "Fast, accurate gaussian process occupancy maps via test-data octrees and nested bayesian fusion". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2016, pp. 1003–1010.

[106] K. Doherty, T. Shan, J. Wang, and B. Englot. "Learning-aided 3-D occupancy mapping with Bayesian generalized kernel inference". In: *IEEE Transactions on Robotics* 35.4 (2019), pp. 953–966.

[107] J. P. Saarinen, H. Andreasson, T. Stoyanov, and A. J. Lilienthal. "3D normal distributions transform occupancy maps: An efficient representation for mapping in dynamic environments". In: *The International Journal of Robotics Research* 32.14 (2013), pp. 1627–1644.

[108] A.-A. Agha-Mohammadi, E. Heiden, K. Hausman, and G. Sukhatme. "Confidence-rich grid mapping". In: *The International Journal of Robotics Research* 38.12-13 (2019), pp. 1352–1374.

[109] F. Ramos and L. Ott. "Hilbert maps: Scalable continuous occupancy mapping with stochastic gradient descent". In: *The International Journal of Robotics Research* 35.14 (2016), pp. 1717–1730.

[110] K. Doherty, J. Wang, and B. Englot. "Probabilistic map fusion for fast, incremental occupancy mapping with 3d hilbert maps". In: *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2016, pp. 1011–1018.

[111] V. Guizilini and F. Ramos. "Large-scale 3d scene reconstruction with hilbert maps". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2016, pp. 3247–3254.

[112] W. Zhi, L. Ott, R. Senanayake, and F. Ramos. "Continuous occupancy map fusion with fast bayesian hilbert maps". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 4111–4117.

[113] S. Srivastava and N. Michael. "Efficient, multifidelity perceptual representations via hierarchical gaussian mixture models". In: *IEEE Transactions on Robotics* 35.1 (2018), pp. 248–260.

[114] C. O'Meadhra, W. Tabib, and N. Michael. "Variable resolution occupancy mapping using gaussian mixture models". In: *IEEE Robotics and Automation Letters* 4.2 (2018), pp. 2015–2022.

[115] P. Fankhauser and M. Hutter. "A universal grid map library: Implementation and use case for rough terrain navigation". In: *Robot Operating System (ROS) The Complete Reference (Volume 1)* (2016), pp. 99–120.

[116] Y. Ren, Y. Cai, F. Zhu, S. Liang, and F. Zhang. "ROG-map: An efficient robocentric occupancy grid map for large-scene and high-resolution LiDAR-based motion planning". In: *arXiv preprint arXiv:2302.14819* (2023).

[117] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. "Real-time 3D reconstruction at scale using voxel hashing". In: *ACM Transactions on Graphics (ToG)* 32.6 (2013), pp. 1–11.

[118] C. Ericson. *Real-time collision detection*. Crc Press, 2004.

[119] G. K. Kraetzschmar, G. P. Gassull, and K. Uhl. "Probabilistic quadtrees for variable-resolution mapping of large environments". In: *IFAC Proceedings Volumes* 37.8 (2004), pp. 675–680.

[120] K. M. Wurm, D. Hennes, D. Holz, R. B. Rusu, C. Stachniss, K. Konolige, and W. Burgard. "Hierarchies of octrees for efficient 3d mapping". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2011, pp. 4249–4255.

[121] E. Einhorn, C. Schröter, and H.-M. Gross. "Finding the adequate resolution for grid mapping-cell sizes locally adapting on-the-fly". In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 1843–1848.

[122] D. Duberg and P. Jensfelt. "UFOMap: An efficient probabilistic 3D mapping framework that embraces the unknown". In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 6411–6418.

[123] N. Funk, J. Tarrio, S. Papatheodorou, M. Popović, P. F. Alcantarilla, and S. Leutenegger. "Multi-resolution 3D mapping with explicit free space representation for fast and accurate mobile robot motion planning". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3553–3560.

[124] Y. Cai, F. Kong, Y. Ren, F. Zhu, J. Lin, and F. Zhang. "Occupancy Grid Mapping Without Ray-Casting for High-Resolution LiDAR Sensors". In: *IEEE Transactions on Robotics* (2023).

[125] A. Guttman. "R-trees: A dynamic index structure for spatial searching". In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 1984, pp. 47–57.

[126] P. Z. X. Li, S. Karaman, and V. Sze. "GMMap: Memory-Efficient Continuous Occupancy Map Using Gaussian Mixture Model". In: *IEEE Transactions on Robotics* (2024).

[127] J. H. Friedman, J. L. Bentley, and R. A. Finkel. *An algorithm for finding best matches in logarithmic time*. Department of Computer Science, Stanford University, 1975.

[128] A. Nuchter, K. Lingemann, and J. Hertzberg. "Cached kd tree search for ICP algorithms". In: *Sixth International Conference on 3-D Digital Imaging and Modeling (3DIM 2007)*. IEEE. 2007, pp. 419–426.

[129] A. Segal, D. Haehnel, and S. Thrun. "Generalized-icp." In: *Robotics: science and systems*. Vol. 2. 4. Seattle, WA. 2009, p. 435.

[130] W. Xu and F. Zhang. "Fast-lio: A fast, robust lidar-inertial odometry package by tightly-coupled iterated kalman filter". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3317–3324.

[131] J. Ichnowski and R. Alterovitz. "Fast nearest neighbor search in SE (3) for sampling-based motion planning". In: *Algorithmic Foundations of Robotics XI*. Springer, 2015, pp. 197–214.

[132] F. Gao and S. Shen. "Online quadrotor trajectory generation and autonomous navigation on point clouds". In: *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE. 2016, pp. 139–146.

[133] B. T. Lopez and J. P. How. "Aggressive 3-D collision avoidance for high-speed navigation." In: *ICRA*. 2017, pp. 5759–5765.

[134] P. R. Florence, J. Carter, J. Ware, and R. Tedrake. "Nanomap: Fast, uncertainty-aware proximity queries with lazy search over local 3d data". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 7631–7638.

[135] F. Gao, W. Wu, W. Gao, and S. Shen. "Flying on point clouds: Online trajectory generation and autonomous navigation for quadrotors in cluttered environments". In: *Journal of Field Robotics* 36.4 (2019), pp. 710–733.

[136] J. Ji, Z. Wang, Y. Wang, C. Xu, and F. Gao. "Mapless-planner: A robust and fast planning framework for aggressive autonomous flight without map fusion". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 6315–6321.

[137] R. B. Rusu and S. Cousins. "3d is here: Point cloud library (pcl)". In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 1–4.

[138] R. Bayer. "Symmetric binary B-trees: Data structure and maintenance algorithms". In: *Acta informatica* 1.4 (1972), pp. 290–306.

[139] C. R. Aragon and R. Seidel. "Randomized search trees". In: *FOCS*. Vol. 30. 1989, pp. 540–545.

[140] D. D. Sleator and R. E. Tarjan. "Self-adjusting binary search trees". In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 652–686.

[141] W. Hunt, W. R. Mark, and G. Stoll. "Fast kd-tree construction with an adaptive error-bounded heuristic". In: *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2006, pp. 81–88.

[142] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek. "Experiences with streaming construction of SAH KD-trees". In: *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2006, pp. 89–94.

[143] M. Shevtsov, A. Soupikov, and A. Kapustin. "Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes". In: *Computer Graphics Forum* 26.3 (2007), pp. 395–404. ISSN: 0167-7055. DOI: 10.1111/j.1467-8659.2007.01062.x.

[144] K. Zhou, Q. Hou, R. Wang, and B. Guo. "Real-time KD-tree construction on graphics hardware". In: *ACM Transactions on Graphics* 27.5 (2008), pp. 1–11. ISSN: 0730-0301. DOI: 10.1145/1409060.1409079.

[145] J. L. Bentley and J. B. Saxe. "Decomposable searching problems I: Static-to-dynamic transformation". In: *J. algorithms* 1.4 (1980), pp. 301–358.

[146] I. Galperin and R. L. Rivest. "Scapegoat Trees." In: *SODA*. Vol. 93. 1993, pp. 165–174.

[147] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. "Bkd-tree: A dynamic scalable kd-tree". In: *International Symposium on Spatial and Temporal Databases*. Springer. 2003, pp. 46–65.

[148] J. T. Robinson. "The KDB-tree: a search structure for large multidimensional dynamic indexes". In: *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 1981, pp. 10–18.

[149] M. Muja and D. G. Lowe. "Fast approximate nearest neighbors with automatic algorithm configuration." In: *VISAPP (1)* 2.331-340 (2009), p. 2.

[150] M. Muja and D. Lowe. "Flann-fast library for approximate nearest neighbors user manual". In: *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* (2009).

[151] P. Chanzy, L. Devroye, and C. Zamora-Cura. "Analysis of range search for random kd trees". In: *Acta informatica* 37.4-5 (2001), pp. 355–383.

[152] Z. Liu, F. Zhang, and X. Hong. "Low-cost Retina-like Robotic Lidars Based on Incommensurable Scanning". In: *IEEE/ASME Transactions on Mechatronics* (2021), pp. 1–1. DOI: 10.1109/TMECH.2021.3058173.

[153] C. Forster, Z. Zhang, M. Gassner, M. Werlberger, and D. Scaramuzza. "SVO: Semidirect visual odometry for monocular and multicamera systems". In: *IEEE Transactions on Robotics* 33.2 (2016), pp. 249–265.

[154] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza. "On-Manifold Preintegration for Real-Time Visual–Inertial Odometry". In: *IEEE Transactions on Robotics* 33.1 (2016), pp. 1–21.

[155] C. Campos, R. Elvira, J. J. G. Rodrıéguez, J. M. Montiel, and J. D. Tardós. "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM". In: *IEEE Transactions on Robotics* (2021).

[156] R. Newcombe. "Dense visual SLAM". PhD thesis. Imperial College London, 2012.

[157] M. Meilland, C. Barat, and A. Comport. "3d high dynamic range dense visual slam and its application to real-time object re-lighting". In: *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. IEEE. 2013, pp. 143–152.

[158] M. Bloesch, J. Czarnowski, R. Clark, S. Leutenegger, and A. J. Davison. "CodeSLAM—learning a compact, optimisable representation for dense visual SLAM". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2560–2568.

[159] C. Kerl, J. Sturm, and D. Cremers. "Dense visual SLAM for RGB-D cameras". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2013, pp. 2100–2106.

[160] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, et al. "Towards fully autonomous driving: Systems and algorithms". In: *2011 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2011, pp. 163–168.

[161] S. Liu, M. Watterson, K. Mohta, K. Sun, S. Bhattacharya, C. J. Taylor, and V. Kumar. "Planning dynamically feasible trajectories for quadrotors using safe flight corridors in 3-d complex environments". In: *IEEE Robotics and Automation Letters* 2.3 (2017), pp. 1688–1695.

[162] D. Wang, C. Watkins, and H. Xie. "MEMS Mirrors for LiDAR: A review". In: *Micromachines* 11.5 (2020), p. 456.

[163] K. Li, M. Li, and U. D. Hanebeck. "Towards high-performance solid-state-lidar-inertial odometry and mapping". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 5167–5174.

[164] J. Lin and F. Zhang. "A fast, complete, point cloud based loop closure for lidar odometry and mapping". In: *arXiv preprint arXiv:1909.11811* (2019).

[165] H. Wang, C. Wang, and L. Xie. "Lightweight 3-D Localization and Mapping for Solid-State LiDAR". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 1801–1807.

[166] C. Forster, M. Pizzoli, and D. Scaramuzza. "SVO: Fast semi-direct monocular visual odometry". In: *2014 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2014, pp. 15–22.

[167] G. C. Sharp, S. W. Lee, and D. K. Wehe. "ICP registration using invariant features". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.1 (2002), pp. 90–102.

[168] K.-L. Low. "Linear least-squares optimization for point-to-plane icp surface registration". In: *Chapel Hill, University of North Carolina* 4.10 (2004), pp. 1–3.

[169] T. Shan, B. Englot, D. Meyers, W. Wang, C. Ratti, and D. Rus. "LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 5135–5142. DOI: 10.1109/IROS45743.2020.9341176.

[170] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert. "iSAM2: Incremental smoothing and mapping using the Bayes tree". In: *The International Journal of Robotics Research* 31.2 (2012), pp. 216–235.

[171] A. Tagliabue, J. Tordesillas, X. Cai, A. Santamaria-Navarro, J. P. How, L. Carlone, and A.-a. Agha-mohammadi. "LION: Lidar-Inertial observability-aware navigator for Vision-Denied environments". In: *International Symposium on Experimental Robotics*. Springer. 2020, pp. 380–390.

[172] K. Koide, M. Yokozuka, S. Oishi, and A. Banno. "Voxelized gicp for fast and accurate 3d point cloud registration". In: *EasyChair Preprint* 2703 (2020).

[173] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. "The R*-tree: An efficient and robust access method for points and rectangles". In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 1990, pp. 322–331.

[174] D. Meagher. "Geometric modeling using octree encoding". In: *Computer graphics and image processing* 19.2 (1982), pp. 129–147.

[175] J. H. Friedman, J. L. Bentley, and R. A. Finkel. "an algorithm for finding best matches in logarithmic expected time". In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), pp. 209–226.

[176] J. Elseberg, S. Magnenat, R. Siegwart, and A. Nüchter. "Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration". In: *Journal of Software Engineering for Robotics* 3.1 (2012), pp. 2–12.

[177] S. Arya and D. Mount. "ANN: library for approximate nearest neighbor searching". In: *Proceedings of IEEE CGC Workshop on Computational Geometry, Providence, RI*. 1998.

[178] M. H. Overmars. *The design of dynamic data structures*. Vol. 156. Springer Science & Business Media, 1987.

[179] J. L. Blanco and P. K. Rai. *nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees*. https://github.com/jlblancoc/nanoflann(v1.3.2). 2014.

[180] C. Hertzberg, R. Wagner, U. Frese, and L. Schröder. "Integrating generic sensor fusion algorithms with sound state representations through encapsulation of manifolds". In: *Information Fusion* 14.1 (2013), pp. 57–77.

[181] D. He, W. Xu, and F. Zhang. "Symbolic representation and toolkit development of iterated error-state extended kalman filters on manifolds". In: *IEEE Transactions on Industrial Electronics* (2023).

[182] Z. Yan, L. Sun, T. Krajnik, and Y. Ruichek. "EU Long-term Dataset with Multiple Sensors for Autonomous Driving". In: *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020.

[183] W. Wen, Y. Zhou, G. Zhang, S. Fahandezh-Saadi, X. Bai, W. Zhan, M. Tomizuka, and L.-T. Hsu. "Urbanloco: a full sensor suite dataset for mapping and localization in urban scenes". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 2310–2316.

[184] N. Carlevaris-Bianco, A. K. Ushani, and R. M. Eustice. "University of Michigan North Campus long-term vision and lidar dataset". In: *The International Journal of Robotics Research* 35.9 (2016), pp. 1023–1035.

[185] G. Barend, L. Bruno, L. Mateusz, W. Adam, K. Menelaos, and F. Vissarion. *Boost Geometry Library*. https://www.boost.org/doc/libs/1_65_1/libs/geometry/. Sept. 2017.

[186] R. B. Rusu and S. Cousins. "3d is here: Point cloud library (pcl)". In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 1–4.

[187] G. Lu, W. Xu, and F. Zhang. "On-manifold model predictive control for trajectory tracking on robotic systems". In: *IEEE Transactions on Industrial Electronics* 70.9 (2022), pp. 9192–9202.

[188] F. Kong, W. Xu, Y. Cai, and F. Zhang. "Avoiding dynamic small obstacles with onboard sensing and computation on aerial robots". In: *IEEE Robotics and Automation Letters* 6.4 (2021), pp. 7869–7876.

[189] J. Lin, C. Zheng, W. Xu, and F. Zhang. "R2LIVE: A Robust, Real-time, LiDAR-Inertial-Visual tightly-coupled state Estimator and mapping". In: *IEEE Robotics and Automation Letters* 6.4 (2021), pp. 7469–7476.

[190] Y. Ren, F. Zhu, W. Liu, Z. Wang, Y. Lin, F. Gao, and F. Zhang. "Bubble planner: Planning high-speed smooth quadrotor trajectories using receding corridors". In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2022, pp. 6332–6339.

[191] Y. Ren, S. Liang, F. Zhu, G. Lu, and F. Zhang. "Online Whole-Body Motion Planning for Quadrotor using Multi-Resolution Search". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, pp. 1594–1600. DOI: 10.1109/ICRA48891.2023.10160767.

[192] F. Zhu, Y. Ren, F. Kong, H. Wu, S. Liang, N. Chen, W. Xu, and F. Zhang. "Swarm-LIO: Decentralized Swarm LiDAR-inertial Odometry". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, pp. 3254–3260. DOI: 10.1109/ICRA48891.2023.10161355.

[193] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman, et al. "A perception-driven autonomous urban vehicle". In: *Journal of Field Robotics* 25.10 (2008), pp. 727–774.

[194] Y. Li, L. Ma, Z. Zhong, F. Liu, M. A. Chapman, D. Cao, and J. Li. "Deep learning for lidar point clouds in autonomous driving: A review". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.8 (2020), pp. 3412–3432.

[195] X. Liu, Z. Liu, F. Kong, and F. Zhang. "Large-Scale LiDAR Consistent Mapping Using Hierarchical LiDAR Bundle Adjustment". In: *IEEE Robotics and Automation Letters* 8.3 (2023), pp. 1523–1530. DOI: 10.1109/LRA.2023.3238902.

[196] S. Liu, M. Watterson, S. Tang, and V. Kumar. "High speed navigation for quadrotors with limited onboard sensing". In: *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2016, pp. 1484–1491.

[197] J. Tordesillas, B. T. Lopez, and J. P. How. "Faster: Fast and safe trajectory planner for flights in unknown environments". In: *2019 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2019, pp. 1934–1940.

[198] Z. Zhang, T. Henderson, S. Karaman, and V. Sze. "FSMI: Fast computation of Shannon mutual information for information-theoretic mapping". In: *The International Journal of Robotics Research* 39.9 (2020), pp. 1155–1177.

[199] A. Elfes. "Robot navigation: Integrating perception, environmental constraints and task execution within a probabilistic framework". In: *Reasoning with Uncertainty in Robotics*. Springer Berlin Heidelberg, 1996, pp. 91–130. DOI: 10.1007/bfb0013955.

[200] *Livox Avia User Manual*. https://www.livoxtech.com/avia. Livox Technology Company Limited. Oct. 2020.

[201] S. Kim and J. Kim. "Building occupancy maps with a mixture of Gaussian processes". In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 4756–4761.

[202] T. Duong, M. Yip, and N. Atanasov. "Autonomous Navigation in Unknown Environments With Sparse Bayesian Kernel-Based Occupancy Mapping". In: *IEEE Transactions on Robotics* 38.6 (2022), pp. 3694–3712. DOI: 10.1109/TRO.2022.3177950.

[203] L. Mescheder, M. Oechsle, M. Niemeyer, S. Nowozin, and A. Geiger. "Occupancy networks: Learning 3d reconstruction in function space". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 4460–4470.

[204] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. "Nerf: Representing scenes as neural radiance fields for view synthesis". In: *Communications of the ACM* 65.1 (2021), pp. 99–106.

[205] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove. "Deepsdf: Learning continuous signed distance functions for shape representation". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 165–174.

[206] A. Rosinol, M. Abate, Y. Chang, and L. Carlone. "Kimera: an Open-Source Library for Real-Time Metric-Semantic Localization and Mapping". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 1689–1696. DOI: 10.1109/ICRA40945.2020.9196885.

[207] Z. Zhu, S. Peng, V. Larsson, W. Xu, H. Bao, Z. Cui, M. R. Oswald, and M. Pollefeys. "Nice-slam: Neural implicit scalable encoding for slam". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12786–12796.

[208] M. C. Martin and H. P. Moravec. *Robot evidence grids*. Carnegie Mellon University, the Robotics Institute, 1996.

[209] Y. Roth-Tabak and R. Jain. "Building an environment model using depth information". In: *Computer* 22.6 (1989), pp. 85–90.

[210] G. M. Hunter and K. Steiglitz. "Operations on images using quad trees". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2 (1979), pp. 145–153.

[211] P. Payeur, P. Hébert, D. Laurendeau, and C. M. Gosselin. "Probabilistic octree modeling of a 3d dynamic environment". In: *Proceedings of International Conference on Robotics and Automation*. Vol. 2. IEEE. 1997, pp. 1289–1296.

[212] M. Yguel, O. Aycard, and C. Laugier. "Update policy of dense maps: Efficient algorithms and sparse representation". In: *Field and service robotics*. Springer. 2008, pp. 23–33.

[213] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. "Ray tracing animated scenes using coherent grid traversal". In: *ACM SIGGRAPH 2006 Papers*. 2006, pp. 485–493.

[214] Y. Kwon, D. Kim, I. An, and S.-e. Yoon. "Super rays and culling region for real-time updates on grid-based occupancy maps". In: *IEEE Transactions on Robotics* 35.2 (2019), pp. 482–497.

[215] J. L. Bentley and D. Wood. "An optimal worst case algorithm for reporting intersections of rectangles". In: *IEEE Transactions on Computers* 29.07 (1980), pp. 571–577.

[216] V. K. Vaishnavi. "Computing point enclosures". In: *IEEE Transactions on Computers* 31.01 (1982), pp. 22–29.

[217] M. R. Spiegel, S. Lipschutz, and J. Liu. *Schaum's Outlines: Mathematical Handbook of Formulas and Tables*. Vol. 2. McGraw-Hill, 2009.

[218] M. Teschner, B. Heidelberger, M. Müller, D. Pomerantes, and M. H. Gross. "Optimized spatial hashing for collision detection of deformable objects." In: *Vmv*. Vol. 3. 2003, pp. 47–54.

[219] W. Xu, Y. Cai, D. He, J. Lin, and F. Zhang. "FAST-LIO2: Fast Direct LiDAR-Inertial Odometry". In: *IEEE Transactions on Robotics* 38.4 (2022), pp. 2053–2073. DOI: 10.1109/TRO.2022.3141876.

[220] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. "Vision meets robotics: The kitti dataset". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1231–1237.

[221] F. Kong, X. Liu, B. Tang, J. Lin, Y. Ren, Y. Cai, F. Zhu, N. Chen, and F. Zhang. "MARSIM: A light-weight point-realistic simulator for LiDAR-based UAVs". In: *IEEE Robotics and Automation Letters* 8.5 (2023), pp. 2954–2961.

[222] J. Amanatides and A. Woo. "A fast voxel traversal algorithm for ray tracing." In: *Eurographics*. Vol. 87. 3. 1987, pp. 3–10.

[223] S. Mystakidis. "Metaverse". In: *Encyclopedia* 2.1 (2022), pp. 486–497.

[224] Y. Wang, Z. Su, N. Zhang, R. Xing, D. Liu, T. H. Luan, and X. Shen. "A survey on metaverse: Fundamentals, security, and privacy". In: *IEEE Communications Surveys & Tutorials* (2022).

[225] P. Cipresso, I. A. C. Giglioli, M. A. Raya, and G. Riva. "The past, present, and future of virtual and augmented reality research: a network and cluster analysis of the literature". In: *Frontiers in psychology* 8 (2018), p. 2086.

[226] S. Shah, D. Dey, C. Lovett, and A. Kapoor. "Airsim: High-fidelity visual and physical simulation for autonomous vehicles". In: *Field and Service Robotics: Results of the 11th International Conference*. Springer. 2018, pp. 621–635.

[227] W. Tabib, K. Goel, J. Yao, C. Boirum, and N. Michael. "Autonomous cave surveying with an aerial robot". In: *IEEE Transactions on Robotics* 38.2 (2021), pp. 1016–1032.

[228] K. Themistocleous, C. Mettas, E. Evagorou, and D. Hadjimitsis. "The use of UAVs and photogrammetry for the documentation of cultural heritage monuments: the case study of the churches in Cyprus". In: *Earth resources and environmental remote sensing/GIS applications X*. Vol. 11156. SPIE. 2019, pp. 85–96.

[229] F. Zhu, Y. Ren, and F. Zhang. "Robust real-time lidar-inertial initialization". In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2022, pp. 3948–3955.

[230] G. Lu, W. Xu, and F. Zhang. "On-Manifold Model Predictive Control for Trajectory Tracking on Robotic Systems". In: *IEEE Transactions on Industrial Electronics* (2022), pp. 1–10. DOI: 10.1109/TIE.2022.3212397.

[231] S. Agarwal, A. Vora, G. Pandey, W. Williams, H. Kourous, and J. McBride. "Ford multi-AV seasonal dataset". In: *The International Journal of Robotics Research* 39.12 (2020), pp. 1367–1376.

[232] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. "A Benchmark for the Evaluation of RGB-D SLAM Systems". In: *Proc. of the International Conference on Intelligent Robot Systems (IROS)*. Oct. 2012.

[233] M. Ramezani, Y. Wang, M. Camurri, D. Wisth, M. Mattamala, and M. Fallon. "The newer college dataset: Handheld lidar, inertial and vision with ground truth". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 4353–4360.

[234] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens. "Engineering a high-performance gpu b-tree". In: *Proceedings of the 24th symposium on principles and practice of parallel programming*. 2019, pp. 145–157.

[235] G. Kim and A. Kim. "Scan context: Egocentric spatial descriptor for place recognition within 3d point cloud map". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 4802–4809.

[236] G. Kim, S. Choi, and A. Kim. "Scan context++: Structural place recognition robust to rotation and lateral variations in urban environments". In: *IEEE Transactions on Robotics* 38.3 (2021), pp. 1856–1874.

[237] C. Yuan, J. Lin, Z. Zou, X. Hong, and F. Zhang. "Std: Stable triangle descriptor for 3d place recognition". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 1897–1903.

[238] C. Yuan, J. Lin, Z. Liu, H. Wei, X. Hong, and F. Zhang. "BTC: A Binary and Triangle Combined Descriptor for 3D Place Recognition". In: *IEEE Transactions on Robotics* (2024).

[239] J. Komorowski. "Minkloc3d: Point cloud based large-scale place recognition". In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2021, pp. 1790–1799.

[240] J. Komorowski. "Improving point cloud based place recognition with ranking-based loss and large batch training". In: *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE. 2022, pp. 3699–3705.

[241] K. Vidanapathirana, M. Ramezani, P. Moghadam, S. Sridharan, and C. Fookes. "LoGG3D-Net: Locally guided global descriptor learning for 3D place recognition". In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, pp. 2215–2221.

[242] J. Knights, S. Hausler, S. Sridharan, C. Fookes, and P. Moghadam. "GeoAdapt: Self-Supervised Test-Time Adaptation in LiDAR Place Recognition Using Geometric Priors". In: *IEEE Robotics and Automation Letters* (2023).

[243] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis. "3D Gaussian Splatting for Real-Time Radiance Field Rendering". In: *ACM Transactions on Graphics* 42.4 (July 2023). URL: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

[244] L. Schmid, O. Andersson, A. Sulser, P. Pfreundschuh, and R. Siegwart. "Dynablox: Real-time detection of diverse dynamic objects in complex environments". In: *IEEE Robotics and Automation Letters* (2023).

[245] H. Wu, Y. Li, W. Xu, F. Kong, and F. Zhang. "Moving event detection from LiDAR point streams". In: *Nature Communications* 15.1 (2024), p. 345.

[246] M. Lu, H. Chen, and P. Lu. "Perception and avoidance of multiple small fast moving objects for quadrotors with only low-cost RGBD camera". In: *IEEE Robotics and Automation Letters* 7.4 (2022), pp. 11657–11664.

[247] T. Khurana, P. Hu, D. Held, and D. Ramanan. "Point Cloud Forecasting as a Proxy for 4D Occupancy Forecasting". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2023.

[248] B. Mersch, X. Chen, I. Vizzo, L. Nunes, J. Behley, and C. Stachniss. "Receding Moving Object Segmentation in 3D LiDAR Data Using Sparse 4D Convolutions". In: *IEEE Robotics and Automation Letters (RA-L)* 7.3 (2022), pp. 7503–7510.

[249] G. E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965), pp. 114–117.

[250] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of solid-state circuits* 9.5 (1974), pp. 256–268.

[251] *Welcome to the Jungle*. https://herbsutter.com/welcome-to-the-jungle/. [Online; accessed 12-June-2024].

[252] *Jetson Modules*. https://developer.nvidia.com/embedded/jetson-modules. [Online; accessed 13-June-2024].

[253] F. Zhu, Y. Ren, F. Kong, H. Wu, S. Liang, N. Chen, W. Xu, and F. Zhang. "Swarm-lio: Decentralized swarm lidar-inertial odometry". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 3254–3260.

[254] H. Xu, Y. Zhang, B. Zhou, L. Wang, X. Yao, G. Meng, and S. Shen. "Omni-swarm: A decentralized omnidirectional visual–inertial–uwb state estimation system for aerial swarms". In: *Ieee transactions on robotics* 38.6 (2022), pp. 3374–3394.