IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

LIO-HKDT: Fast and Accurate LiDAR-Inertial Odometry with Hash K-D Tree

Yuexin Mu, Ao Ren✉, Duo Liu✉, Murong Wang, Zihao Zhang, Haojie Lu, Yujuan Tan, Kan Zhong

***Abstract*—LiDAR-inertial odometry(LIO) has been widely applied in intelligent robotics and autonomous driving, providing high-precision and low-latency ego-motion estimation. However, the massive point clouds generated by LiDAR introduce intensive data processing demands, making k-nearest neighbor(KNN) search and map update a critical bottleneck that limits the real-time performance of the LIO system. This letter proposes a novel data structure, the Hash K-D Tree(hkd-Tree), which uses hashed voxel indices as keys and local k-d tree as values. It combines the localized search advantages of voxel-based methods with the efficient search capability of k-d tree, enabling fast KNN search and point cloud insertion. To further improve the performance of the hkd-Tree, we propose a voxel distribution mechanism and buffered update strategy, where each new point is assigned to neighboring voxels within the search radius and inserted into local k-d tree via parallel batch updates. We develop a LiDAR-inertial odometry system, LIO-HKDT, based on the proposed hkd-Tree. Extensive experiments demonstrate that the hkd-Tree enables highly efficient point cloud search and insertion. LIO-HKDT achieves comparable accuracy to state-of-the-art LIO systems while significantly improving runtime efficiency.**

***Index Terms*—SLAM, Localization, Range Sensing**

I. Introduction

LIDAR-INERTIAL odometry(LIO) estimates ego-motion by fusing LiDAR and IMU data, which provides high accuracy and robustness[1], [2]. It has been widely applied in fields such as intelligent robotics and autonomous driving[3]. A critical step in LIO systems is to align the current LiDAR scan with the global map to obtain the pose transformation with respect to the global coordinate frame[4]. This process requires frequent large-scale K-Nearest Neighbor (KNN) searches to construct residuals for pose estimation, which have become a major bottleneck for real-time performance[5], [6]. Meanwhile, newly generated point clouds must be incrementally integrated into the existing data structure to support continuous KNN searches. Therefore, a data structure that enables efficient KNN searches and incremental updates is essential for improving the overall performance of LIO systems.

To accelerate KNN searches, various spatial indexing structures have been proposed. Among them, the octree partitions 3D space into voxels and supports radius-based searches, but its efficiency is limited by the resolution, making it hard to balance accuracy and speed[7]. The k-d tree is a classic binary partitioning structure that achieves high search efficiency in low-dimensional and uniformly distributed point cloud scenarios[8]. It has been widely integrated into libraries such as PCL and FLANN[9], [10]. Nevertheless, LIO systems operate on continuously generated large-scale point clouds, which require frequent incremental updates and nearest neighbor searches[11]. Standard k-d tree incur considerable computational overhead during such updates, as they often necessitate reconstruction or rebalancing of the tree structure.

To address this, data structures such as the ikd-Tree[11] and i-Octree[12] have been proposed. These methods extend the standard k-d tree and octree by introducing mechanisms for dynamic insertion and deletion of point clouds, thereby improving the efficiency of KNN search and incremental updates. In particular, FAST-LIO2[13], which is based on the ikd-tree, effectively mitigates the issues of structural reconstruction and rebalancing. However, despite the high search efficiency of k-d tree, the ikd-tree still maintains a local map containing a large number of points, resulting in significant computational overhead for KNN search. FASTER-LIO[14] introduces the incremental voxel(iVox) structure, which manages sparse voxels using a hash table. By hashing search points to locate their corresponding voxels and traversing neighboring voxels to retrieve nearby points. However, in dense point cloud scenarios, brute-force traversal of all points within neighboring voxels leads to high computational overhead, which severely affects the real-time performance of the system.

In this paper, we propose the Hash K-D Tree(hkd-Tree) to address the above problem. We deem that the key to further improving the efficiency of KNN searches and incremental updates for point clouds lies in integrating the local search advantages of voxel-based methods with the spatial search efficiency of k-d tree. At the same time, it is essential to avoid the brute-force traversal of all points in neighboring voxels inherent to voxel methods, as well as the inefficiencies in search and update performance caused by building a k-d tree over the entire local map. To achieve this, we introduce a novel data structure, the hkd-Tree, which is designed as a sparse hash table where voxel hash values serve as keys and local k-d tree serve as values. The hkd-Tree simultaneously preserves voxel-level locality and enables efficient KNN searches within each voxel, eliminating the need for a global tree structure and greatly improving update and search efficiency.

To exploit the fixed-radius nature of KNN searches in LIO systems, we design a voxel distribution mechanism in the hkd-Tree. During insertion, each point is stored not only in its

Manuscript received: August, 8, 2025; Revised October, 21, 2025; Accepted December, 13, 2025. This paper was recommended for publication by Editor S. Behnk upon evaluation of the reviewers' comments. This work was supported by the Fundamental Research Funds for the Central Universities(Nos. 2024CDJGF-003, 2024CDJGF-019), by a research project of the supervisor's research group (Nos. 0225005401117), and by the National Natural Science Foundation of China(Nos. 62402070, 62572085). (Corresponding authors: Duo Liu and Ao Ren.)

The authors are with Chongqing University, Chongqing, China, and also with the Jialingjiang Laboratory, Chongqing, China (e-mail: yuexinmu@stu.cqu.edu.cn, liuduo@cqu.edu.cn, ren.ao@cqu.edu.cn).

Digital Object Identifier (DOI): see top of this page.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

primary voxel but also distributed to neighboring voxels within the search radius. This allows KNN searches to be resolved using only the local k-d tree of the current voxel, eliminating cross-voxel traversal and significantly improving search efficiency. In addition, we introduce a buffered update strategy: instead of performing immediate point-wise insertions into each local k-d tree, points assigned to a local k-d tree are first accumulated in its corresponding buffer during the insertion phase. A dedicated thread then asynchronously batches the buffered points into their respective local k-d tree, preventing frequent and inefficient single-point updates. This strategy leverages the temporal gap between incremental updates and KNN searches (e.g., preprocessing of new point cloud), further enhancing the efficiency of point cloud insertion.

Our contributions can be summarized as follows:

- We propose a novel data structure, hkd-Tree, which performs KNN searches by accessing only the local k-d tree within a single voxel, significantly improving the real-time performance of LIO systems.
- We propose a voxel distribution mechanism and a buffered update strategy, which distribute points to neighboring local k-d tree and update them in parallel, further improving insertion and search efficiency.
- We develop a LIO system, LIO-HKDT, based on the hkd-Tree. Extensive experimental results show that it achieves accuracy comparable to state-of-the-art methods while significantly improving runtime efficiency.

II. Related Works

LiDAR-inertial odometry(LIO) systems rely on accurate point cloud registration to estimate the relative pose between consecutive LiDAR scans[15]. The widely used Iterative Closest Point(ICP) algorithm[16] estimates relative transformations by iteratively minimizing the distances between corresponding points. Since registration frequently involves K-Nearest Neighbor(KNN) search, introducing efficient data structures can significantly accelerate this process[5]. To better position our work within the existing literature, we review representative spatial indexing structures relevant to this study.

The R-tree[17] is a classic data partitioning structure that supports KNN searches by clustering spatially adjacent data into potentially overlapping axis-aligned bounding boxes. Its improved variant, the R*-tree[18], introduces insertion based on the minimum-overlap criterion and enforces reinsertion during node splits to optimize search performance. Other commonly used spatial partitioning structures include the octree[7] and the k-dimensional tree(k-d tree)[8], the latter being a binary tree structure. Due to the high efficiency of k-d tree in KNN search[19], [20], they have been widely adopted in mainstream LIO methods such as LOAM, LEGO-LOAM, and their variants[21], [22], [23]. For example, LIO-SAM[24] builds a k-d tree on the point clouds of several keyframes and performs nearest neighbor searches using FLANN[25]. However, point cloud data in LIO systems are continuously and rapidly generated. Rebuilding the entire k-d tree from scratch to incorporate new frames is highly inefficient and time-consuming[11]. As a result, mainstream approaches often choose to update the k-d tree at low frequency[21], [22], [26], or construct local tree structures only for new data[27], [28]. While these strategies alleviate computational costs, they also limit the real-time performance and scalability of the system to some extent.

To address the challenge of continuously generated point cloud data in real-world applications, researchers have proposed various incremental spatial partitioning structures. Cai et al. introduced the ikd-Tree[11], which extends the traditional k-d tree by incorporating an incremental update mechanism. It also employs a multi-threaded self-balancing strategy, effectively reducing the time overhead of updates and searches, and improving the real-time performance of LIO systems. Zhu et al. proposed the i-Octree[12], which enhances the traditional octree to support dynamic insertion, deletion, and downsampling operations. Overall, incremental k-d tree and octrees maintain efficient nearest neighbor search while substantially reducing the cost of map updates. However, LiDAR sensors can generate tens of thousands of points per second. Even with incremental update strategies, it remains challenging to perform efficient KNN searches on local maps that contain a large number of points.

In contrast to tree-based spatial partitioning of point clouds, Faster-LIO proposes an incremental voxel structure (iVox)[14], which employs a hash table to manage sparse voxels. By locating the voxel of the search point and traversing its neighboring voxels, KNN searches are performed within a local region, which yields higher efficiency compared to traditional k-d tree-based methods. However, its neighbor search requires traversing all points in the neighboring voxels, which can incur significant overhead in dense point clouds. Moreover, the voxel resolution and the number of neighboring voxels to be queried must be carefully balanced between accuracy and efficiency.

III. Method

A. System Overview

We now introduce our proposed LIO-HKDT system, and its pipeline is illustrated in Fig. 1. Sensor inputs are first processed by the preprocessing module, followed by the state estimation module, which performs pose estimation by matching the new point cloud with the existing map using KNN search results. After obtaining the pose, the system forwards the latest point cloud to the incremental update module for map maintenance and then proceeds to the next frame. When inserting new points, the incremental update module first employs the voxel distribution module to propagate each point to neighboring voxels within the search radius. The buffered update module then aggregates the points assigned to each voxel into the corresponding point buffers and performs parallel batch insertions into the associated local k-d tree using multi-threading. With the voxel distribution mechanism, KNN searches are performed within a small local k-d tree. This effectively combines the local search efficiency of voxel-based methods with the high search efficiency of k-d tree, resulting in significantly improved search performance. Meanwhile, the buffered update module avoids the inefficiency of frequent single-point updates to local k-d tree, further improving the efficiency of incremental update.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

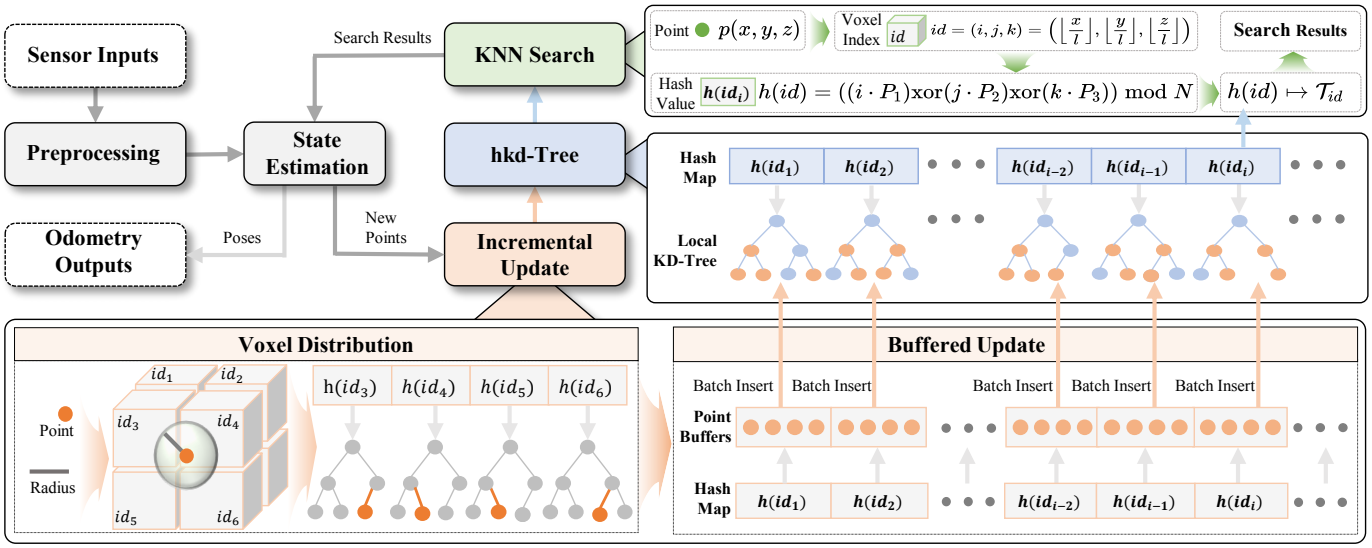


Fig. 1. System overview of LIO-HKDT. The hkd-Tree is implemented as a hash table, where each key is the hash value of a voxel index, and each value is the corresponding local k-d tree (blue box). Point cloud insertion consists of two stages: voxel distribution and buffered update (orange box). Point cloud search is performed by directly retrieving the local k-d tree using the hash value computed from the voxel index of the search point (green box).

B. Data Structure of hkd-Tree

The hkd-Tree is a hash table composed of spatial voxel indices and local k-d tree. Each key is the hash value of a fixed-size voxel index in 3D space, and each value is a lightweight local k-d tree(implemented with nanoflann[29]) maintained within the corresponding voxel. Point cloud insertion and search are performed efficiently by indexing directly into the local k-d tree via the hash value. This design retains the localized search advantages of voxel-based methods, avoids the overhead of maintaining a large number of points in a single k-d tree, and fully exploits the high search efficiency of k-d tree.

Given an arbitrary point $\mathbf{p} = (x, y, z) \in \mathbb{R}^3$ and a voxel resolution l, the corresponding voxel index $id = (i, j, k) \in \mathbb{Z}^3$ is computed as:

$$id = (i, j, k) = \left(\left\lfloor \frac{x}{l} \right\rfloor, \left\lfloor \frac{y}{l} \right\rfloor, \left\lfloor \frac{z}{l} \right\rfloor \right). \tag{1}$$

The hash value $h(id)$ of the voxel index is then computed using a hash function [30]:

$$h(id) = ((i \cdot P_1) \text{xor} (j \cdot P_2) \text{xor} (k \cdot P_3)) \bmod N, \tag{2}$$

where P_1, P_2, and P_3 are large prime numbers, and N is the size of the hash table.

The overall structure of the hkd-Tree can thus be represented as a mapping:

$$\mathcal{H} = h(id) \mapsto \mathcal{T}_{id}, \tag{3}$$

where $\mathcal{T}_{id}$ denotes the local k-d tree associated with voxel index id.

In practical scenarios, point clouds are sparsely distributed in space, and only a subset of voxels are populated in LIO systems. Therefore, the hkd-Tree maintains only those voxels that contain points, along with their corresponding local k-d tree. To limit memory usage, the system adopts an LRU (Least Recently Used) cache eviction mechanism [14]. Once the number of active voxels exceeds a predefined threshold, voxels that have not been accessed for a long time are automatically released. Benefiting from the $O(1)$ insertion and deletion complexity of hash tables, this process can be performed efficiently.

C. Incremental Update

We leverage two key properties of LIO systems: the fixed-radius constraint of KNN searches and the temporal gap between point cloud insertion and subsequent search (e.g., due to IMU integration or motion distortion correction). Based on these observations, we design a two-stage incremental update strategy—voxel distribution and buffered update—which significantly improves the KNN search and point insertion efficiency of the hkd-Tree.

1) Voxel Distribution: In LIO systems, KNN searches are typically constrained by a fixed radius. To ensure that each search can be completed within a single local k-d tree, we introduce a voxel distribution mechanism during the point cloud insertion stage. Specifically, let the KNN search radius be r, and let the set of points to be inserted be $\mathcal{P}_{\text{add}} = \mathbf{p}_i \in \mathbb{R}^3$. For each point $\mathbf{p}_i$, the voxel index id_i is computed using Eq. (1). We then define the neighboring voxel set as:

$$\mathcal{N}_i = \left\{ id \in \mathbb{Z}^3 \,\middle|\, \|id \cdot l - \mathbf{p}_i\|_2 \leq r \right\}, \tag{4}$$

which includes all voxel indices whose centers lie within a ball of radius r centered at $\mathbf{p}_i$. The point $\mathbf{p}_i$ is then inserted into the local k-d tree of each voxel $id \in \mathcal{N}_i$:

$$\forall id \in \mathcal{N}_i, \quad \mathbf{p}_i \in \mathcal{T}_{id}. \tag{5}$$

This mechanism ensures that radius-based KNN searches can be performed by accessing only the local k-d tree of the

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

voxel containing the search point, without scanning neighboring voxels. As a result, it significantly improves search efficiency by avoiding brute-force traversal of all nearby points.

2) Buffered Update: To further improve point cloud insertion efficiency, we design a buffered update strategy that performs multi-threaded batch insertion, avoiding the inefficiency of frequent single-point insertions into local k-d tree. Specifically, all incoming points are first processed through voxel distribution, establishing a mapping between each point and its set of associated voxels. The points are then temporarily buffered in the corresponding voxel containers. Once buffering is complete, a dedicated thread performs a batch insert operation into the corresponding local k-d tree. Note that the batch in "batch insert" corresponds to the number of buffered points assigned to a single local KD-tree, and its size is determined by the voxel distribution stage rather than being a tunable hyperparameter.

D. KNN Search

In LIO systems, KNN search is a critical step for point cloud registration, aiming to efficiently retrieve neighboring points within a specified search radius from the local map. Due to the voxel distribution mechanism applied during the insertion stage of the hkd-Tree, all points within the fixed search radius are pre-distributed to the voxel containing the search point. This eliminates the need for brute-force traversal of neighboring voxels during the search stage—KNN search can be completed by accessing only the local k-d tree within that voxel. Given a search point $\mathbf{q} \in \mathbb{R}^3$, its voxel index id_q is first computed using Eq.(1), followed by the hash key $h(id_q)$ using Eq.(2). The corresponding local k-d tree $\mathcal{T}_{id_q}$ is retrieved from the hkd-Tree hash table $\mathcal{H}$, and a KNN search is performed within it to obtain the neighbors of $\mathbf{q}$ within the search radius. Since both the hash computation for $\mathbf{q}$ and the local search within $\mathcal{T}_{id_q}$ are highly efficient, the hkd-Tree enables fast point cloud search.

E. Complexity of hkd-Tree

The computational complexity consists of two parts: incremental update and KNN search. Assuming the number of points to be inserted is denoted as n and the number of points in the local k-d tree as m, and the corresponding size of the local k-d tree as s. The incremental update mainly involves voxel distribution and buffered update. The voxel distribution has a time complexity of $O(n)$, as each point is processed once. The buffered update stage, which inserts m points into a k-d tree of size s, leads to a time complexity of $O(mlog(s))$. Therefore, the overall time complexity of incremental update is $O(nmlog(s))$. In the KNN search stage, the hkd-Tree locates the corresponding local k-d tree via hash indexing and performs the nearest neighbor search within it, resulting in a time complexity of $O(log(s))$.

IV. Experiments

This section presents experiments evaluating the hkd-Tree in terms of point insertion and nearest neighbor search efficiency. We first conduct comparative tests on randomly generated data to assess its performance against state-of-the-art methods. We then integrate hkd-Tree into a LIO system, termed LIO-HKDT, and evaluate its effect on real-time performance and pose estimation accuracy in real-world scenarios. To better isolate the contribution of the buffered update strategy to overall system performance, we implemented a serial version of hkd-Tree. In this version, insertion and search operations are executed sequentially without multi-threading. This version is denoted as hkd-Tree(Serial) and LIO-HKDT(Serial) in the subsequent comparative experiments. All experiments are conducted on a desktop equipped with an AMD R9-5950X CPU (3.4 GHz).

A. Randomized Data Experiments

To quantitatively evaluate the performance of hkd-Tree in point cloud insertion, KNN search and peak memory usage, we conduct comparisons with three state-of-the-art dynamic data structures: ikd-Tree[11], iVox(Linear)[14], and i-Octree[12]. These methods represent state-of-the-art solutions for efficient point cloud management and have been widely adopted in real-time LiDAR SLAM systems. We first generate 100,000 initial points within cubic spaces of side lengths 30m, 20m, and 10m to construct various data structures. Then, we perform 100 iterations, where each iteration inserts 1,000 newly generated points and conducts 1,000 radius-based KNN searches with a search radius of $r = 5m$ and $k = 5$ nearest neighbors. For each iteration, we record the point insertion time, KNN search time, their sum as the total computational cost, and the peak memory usage throughout the process. This allows for a comprehensive evaluation of the performance of different methods under varying point cloud densities.

As shown in Fig. 2, the proposed hkd-Tree consistently outperforms other methods, achieving the lowest total computational cost across all scenarios. Specifically, under identical spatial conditions, as point cloud density increases, both hkd-Tree and hkd-Tree(Serial) maintain stable insertion and search efficiency, while other methods show gradual degradation. In particular, iVox requires traversing all points in neighboring voxels during search, leading to a significant increase in search cost under high-density conditions. In terms of insertion performance, hkd-Tree employs a buffered update strategy that performs asynchronous insertions, fully exploiting the parallelism between insertion and search in LIO systems to achieve superior insertion efficiency. Although iVox achieves relatively fast insertion by simply placing points into corresponding voxels, its overall performance is still constrained by its brute-force search strategy.

Table I summarizes the average insertion time, search time, and peak memory usage of different methods under various point cloud densities. The results indicate that the proposed hkd-Tree only introduce minor extra memory usage. Nevertheless, given the typical memory capacity (4–64 GB) of modern computing platforms, this increase is considered reasonable. More importantly, hkd-Tree consistently achieves the highest insertion and search efficiency across all densities, demonstrating that it strikes a good balance between memory usage and computational performance.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

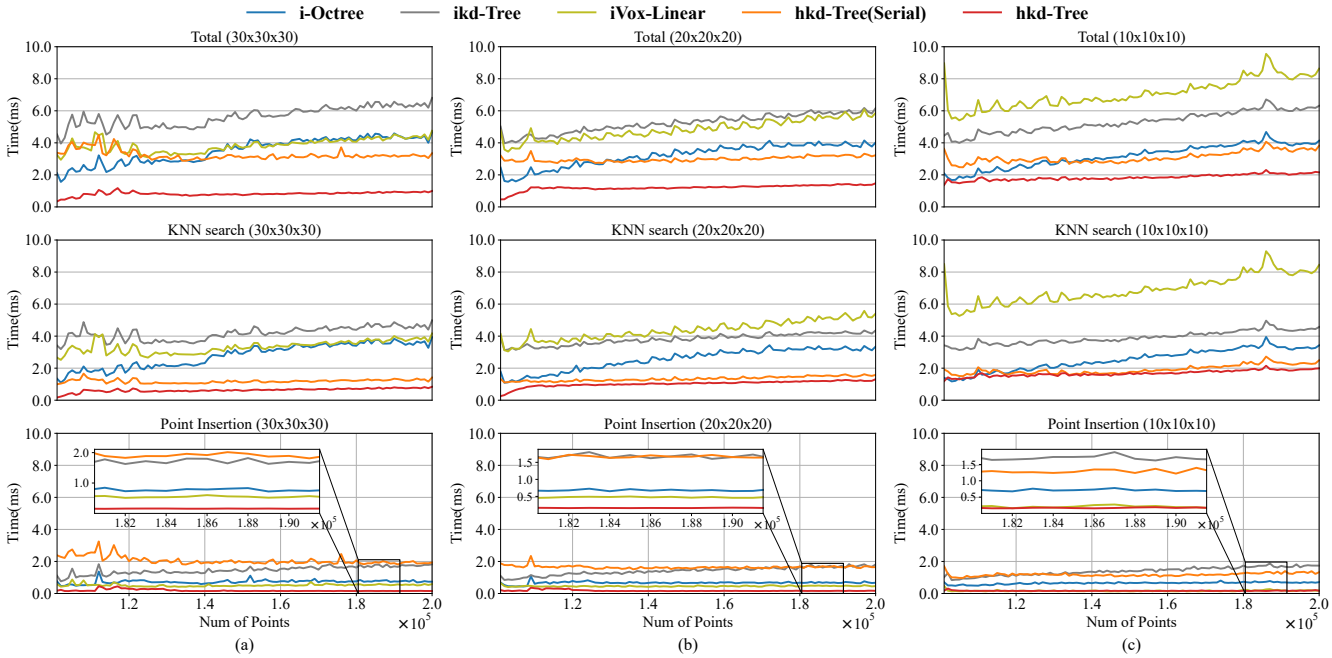


Fig. 2. Experimental results of point cloud insertion and KNN search time under different point cloud densities. "Total," "KNN Search," and "Point Insertion" represent the total time, KNN search time, and point insertion time, respectively. Subfigures (a), (b), and (c) correspond to cubic spaces with side lengths of 30 m, 20 m, and 10 m, respectively.

TABLE I
COMPARISON OF AVERAGE TIME AND PEAK MEMORY USAGE

		ikd-Tree	iVox-Linear	i-Octree	hkd-Tree(Serial)	hkd-Tree
Cubic Space (30×30×30)	Ins.(ms)[1]	1.46	0.61	0.73	2.06	**0.19**
	KNN(ms)[2]	4.18	3.45	2.79	1.19	**0.63**
	Mem.(MB)[3]	26.82	18.72	**11.79**	21.69	21.18
Cubic Space (20×20×20)	Ins.(ms)	1.41	0.46	0.67	1.65	**0.18**
	KNN(ms)	3.81	4.49	2.55	1.34	**1.03**
	Mem.(MB)	26.80	12.58	**11.88**	14.85	15.56
Cubic Space (10×10×10)	Ins.(ms)	1.40	0.18	0.65	1.18	**0.17**
	KNN(ms)	3.91	6.92	2.53	1.92	**1.68**
	Mem.(MB)	26.80	**8.92**	11.80	11.16	12.56

[1] Ins. denotes the average point insertion time.
[2] KNN. denotes the average KNN search time.
[3] Mem. denotes the peak memory usage.
The notation $(30 \times 30 \times 30)$, $(20 \times 20 \times 20)$, and $(10 \times 10 \times 10)$ represent cubic spaces with side lengths of 30 m, 20 m, and 10 m, respectively.

TABLE II
DETAILS OF ALL DATASET SEQUENCES

	Duration[1]	Distance[2]	Points[3]	Avg Points[4]	Name
nclt_1	111:46	4.01	2.30×10^9	3.43×10^4	20120115
nclt_2	43:17	1.86	9.29×10^8	3.60×10^4	20120429
nclt_3	84:32	3.13	1.72×10^9	3.38×10^4	20120511
nclt_4	55:10	1.62	1.17×10^9	3.54×10^4	20120615
nclt_5	17:02	0.26	3.12×10^8	3.07×10^4	20130110
m2dgr_1	3:01	0.14	8.11×10^7	4.71×10^4	gate01
m2dgr_2	5:36	0.29	1.51×10^8	4.68×10^4	gate02
m2dgr_3	4:56	0.25	1.33×10^8	4.73×10^4	gate03
liosam_1	9:11	0.66	6.77×10^7	1.24×10^4	park
liosam_2	5:58	0.46	6.62×10^7	1.87×10^4	garden
liosam_3	16:26	1.44	1.89×10^8	1.92×10^4	campus
utbm_1	16:59	5.03	3.99×10^8	3.97×10^4	20180713
utbm_2	15:59	4.99	3.87×10^8	4.06×10^4	20180717
utbm_3	16:39	5.00	4.27×10^8	4.30×10^4	20180718
utbm_4	15:26	4.98	3.97×10^8	4.31×10^4	20180719
utbm_5	16:45	4.99	4.22×10^8	4.21×10^4	20180720
utbm_6	14:55	4.99	3.55×10^8	3.99×10^4	20190418
utbm_7	11:59	5.11	2.72×10^8	3.78×10^4	20190418(round)

[1] *Duration* (min:s) is the recording duration of the sequence.
[2] *Distance* (km) is the traveled path length of the sequence.
[3] *Points* are the total number of valid points in the dataset.
[4] *Avg Points* are the average number of valid points per scan.

B. Real-world Data Experiments

To further evaluate the effectiveness of hkd-Tree in a practical LIO system, we build LIO-HKDT based on FASTER-LIO. The main modification involves replacing the original point cloud insertion and KNN search modules with the proposed hkd-Tree, while keeping all other components and parameter settings unchanged. We compare LIO-HKDT against several representative LiDAR-inertial odometry methods, including FAST-LIO2[13] based on ikd-Tree, FASTER-LIO[14] based on iVox(Linear), an extended version of FASTER-LIO integrated with i-Octree (denoted as FASTER-LIO(i-Octree)), as well as the static k-d tree–based methods LILI-OM[31], LIO-SAM[24], and DLIO[32]. Experiments are conducted on four public datasets: UTBM[33], M2DGR[34], NCLT[35], and the LIOSAM dataset (from LIO-SAM[24]), with detailed information provided in TableII. These datasets cover diverse environments, including urban roads, university campuses, and parks, enabling a comprehensive evaluation of system robustness and generalization in complex real-world scenarios. They are widely adopted in LIO research and serve as standard benchmarks for assessing modern LIO performance. Some sequences cannot be evaluated due to missing data required by certain methods (e.g., IMU orientation quaternions), denoted as "–" in the results. If a method fails or drifts excessively, making the result invalid, it is marked as "×".

1) Processing Time Evaluation: We first measure the time consumed by different data structures for KNN search, incremental updates (new point insertion), and their total runtime

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

TABLE III
THE COMPARISON OF AVERAGE TIME CONSUMPTION PER SCAN ON KNN SEARCH, INCREMENTAL UPDATE AND TOTAL TIME

	Total[ms]					KNN Search[1] [ms]					Incremental Update[2] [ms]				
	ikd-Tree	iVox-Linear	i-Octree	hkd-Tree (Serial)	hkd-Tree	ikd-Tree	iVox-Linear	i-Octree	hkd-Tree (Serial)	hkd-Tree	ikd-Tree	iVox-Linear	i-Octree	hkd-Tree (Serial)	hkd-Tree
nclt_1	3.66	3.43	2.95	2.82	**1.79**	2.46	2.84	2.11	1.44	**1.40**	1.20	0.59	0.84	1.38	**0.40**
nclt_2	3.91	3.53	2.95	2.83	**1.87**	2.59	2.90	2.09	1.48	**1.44**	1.32	0.63	0.86	1.35	**0.43**
nclt_3	3.70	3.25	2.80	2.74	**1.75**	2.43	2.66	1.99	1.40	**1.35**	1.27	0.59	0.81	1.34	**0.40**
nclt_4	4.04	3.45	2.91	2.78	**1.83**	2.64	2.80	2.07	1.46	**1.40**	1.40	0.65	0.84	1.32	**0.43**
nclt_5	3.46	2.41	2.47	2.21	**1.50**	2.17	1.93	1.78	1.21	**1.13**	1.29	0.47	0.69	1.00	**0.37**
m2dgr_1	5.74	6.10	3.95	4.19	**2.87**	3.36	5.20	2.67	2.39	**2.29**	2.38	0.90	1.28	1.80	**0.57**
m2dgr_2	5.12	5.06	3.86	3.79	**2.84**	3.13	4.28	2.58	2.18	**2.17**	2.00	0.79	1.28	1.61	**0.67**
m2dgr_3	5.49	5.16	3.90	3.79	**3.01**	3.34	4.37	2.61	**2.18**	2.33	2.14	0.79	1.28	1.61	**0.68**
liosam_1	3.12	1.85	2.01	1.30	**0.89**	1.66	1.44	1.42	**0.70**	**0.70**	1.46	0.40	0.59	0.60	**0.19**
liosam_2	2.65	1.75	1.50	1.30	**1.00**	1.58	1.39	1.02	0.76	**0.74**	1.07	0.36	0.48	0.54	**0.25**
liosam_3	4.63	1.75	1.80	1.54	**1.07**	2.67	1.39	1.12	**0.76**	**0.76**	1.96	0.36	0.68	0.78	**0.31**
utbm_1	8.37	2.72	2.85	3.04	**1.72**	5.11	2.00	1.93	**1.26**	1.32	3.27	0.72	0.92	1.78	**0.39**
utbm_2	8.37	2.68	2.95	3.20	**1.74**	5.11	1.95	1.95	**1.27**	1.32	3.27	0.73	1.00	1.93	**0.41**
utbm_3	7.84	3.07	3.20	2.74	**1.95**	4.66	2.27	2.23	1.41	**1.37**	3.18	0.80	0.97	1.33	**0.59**
utbm_4	8.75	2.75	3.05	3.28	**1.71**	5.24	1.98	1.93	1.31	**1.30**	3.51	0.77	1.12	1.97	**0.41**
utbm_5	8.22	2.97	3.00	2.81	**1.83**	4.95	2.18	2.02	1.43	**1.30**	3.26	0.79	0.98	1.38	**0.54**
utbm_6	6.81	3.33	3.08	2.58	**1.86**	3.38	2.44	2.08	1.32	**1.27**	3.43	0.89	1.00	1.26	**0.59**
utbm_7	11.75	3.12	3.78	3.50	**1.86**	7.16	2.23	2.66	**1.33**	1.43	4.59	0.89	1.12	2.17	**0.43**

[1]KNN Search: The average time consumption for KNN search in each scan.
[2]Incremental Update: The average time consumption for new point insertion in each scan.

TABLE IV
AVERAGE PROCESSING TIME PER SCAN (MS)

	FAST-LIO2	FASTER-LIO	FASTER-LIO (i-Octree)	DLIO	LIO-SAM		LILI-OM		LIO-HKDT (Serial)	LIO-HKDT
	Total	Total	Total	Total	Pre.	Opt.	Pre.	Opt.	Total	Total
nclt_1	7.27	7.61	7.18	23.76	x	x	x	x	7.13	**6.45**
nclt_2	7.70	7.88	7.46	22.22	x	x	x	x	7.41	**6.66**
nclt_3	7.31	7.43	7.10	20.24	x	x	x	x	7.01	**6.29**
nclt_4	7.75	7.80	7.36	19.68	x	x	x	x	7.33	**6.61**
nclt_5	6.97	6.56	6.60	15.59	8.79	18.98	49.42	28.62	6.55	**5.89**
m2dgr_1	10.30	11.95	10.42	20.90	4.79	17.52	32.45	31.62	10.05	**8.79**
m2dgr_2	9.33	10.52	9.90	20.78	7.37	5.07	25.98	26.26	9.40	**8.52**
m2dgr_3	9.87	10.71	10.02	21.26	8.30	9.73	26.38	25.78	9.40	**8.75**
liosam_1	5.72	5.11	5.21	12.00	8.09	9.95	52.96	31.36	4.00	**3.47**
liosam_2	5.93	5.55	4.62	11.85	8.26	9.90	59.33	32.86	4.30	**3.87**
liosam_3	7.42	5.54	4.83	12.75	7.34	9.95	66.70	42.78	4.30	**3.86**
utbm_1	13.32	8.35	8.55	17.20	-	-	28.12	31.38	8.49	**7.57**
utbm_2	13.32	8.34	8.79	18.33	-	-	29.94	33.39	8.69	**7.61**
utbm_3	13.19	8.96	9.11	18.01	-	-	25.79	21.75	8.70	**8.07**
utbm_4	13.96	8.56	9.29	17.74	-	-	30.48	32.45	8.89	**7.81**
utbm_5	13.42	8.77	8.86	18.57	-	-	25.83	20.79	8.74	**7.89**
utbm_6	14.72	12.18	8.92	17.13	-	-	24.91	20.54	8.38	**7.84**
utbm_7	16.68	8.60	8.95	18.21	-	-	27.34	21.54	8.80	**7.61**

across all datasets. This allows us to independently evaluate the point cloud processing efficiency of each structure within a practical LIO system. The results are presented in Table III. In addition, we record the per-scan processing time of each LIO method to assess the impact of the underlying data structure on the overall runtime efficiency of the system. For FAST-LIO2, FASTER-LIO, FASTER-LIO(i-Octree), DLIO, and LIO-HKDT, the total processing time refers to the entire odometry pipeline (including point cloud preprocessing, undistortion, downsampling, pose estimation and incremental updates). Since LIO-SAM and LILI-OM deploy odometry and mapping in separate ROS nodes, we report the average time for preprocessing (Pre., including feature extraction and rough pose estimation) and optimization (Opt., such as back-end fusion in LILI-OM and incremental smoothing and mapping in LIO-SAM) separately. To enable fair comparison with other methods, the sum of these two components is reported as the total processing time.

As shown in Table III, hkd-Tree consistently achieves the lowest total runtime for KNN search and incremental updates across all sequences, while its serial variant, hkd-Tree(Serial), also outperforms iVox-Linear and ikd-Tree in most cases. In KNN search, hkd-Tree and hkd-Tree(serial) consistently achieve the lowest latency, significantly outperforming other methods. For incremental update, hkd-Tree also demonstrates excellent performance when executing insertion tasks in parallel threads. It is worth noting that direct point-to-voxel assignment enables fast updates in iVox-Linear. However, its search speed degrades rapidly as point density increases, limiting overall performance.

As shown in Table IV, LIO-HKDT achieves the lowest average per-scan processing time across all datasets. Its serial variant, LIO-HKDT(Serial), outperforming most baseline methods, with only FASTER-LIO and FASTER-LIO(i-Octree) achieving comparable performance on a few sequences (e.g., nclt_3 and nclt_5). Although FASTER-LIO performs slightly better on a few sequences (e.g., utbm_1 and utbm_2), its overall efficiency remains lower than that of the full LIO-HKDT. This is because the hkd-Tree in LIO-HKDT maintains a balanced computational cost between point insertion and KNN search, allowing the system to benefit significantly from parallel processing. However, FASTER-LIO suffers from a major performance bottleneck in the search stage, where even parallelization yields only limited improvement in overall efficiency. FAST-LIO2 exhibits higher average processing times across most sequences. Meanwhile, the data structures used in LIO-SAM, LILI-OM and DLIO require frequent reconstruction, resulting in significantly higher processing times.

To further analyze the worst-case performance and peak computational load of LIO-HKDT, we recorded the per-frame processing time of both LIO-HKDT and FASTER-LIO on the liosam_1 sequence. As shown in Fig. 3, LIO-HKDT exhibits more stable processing performance with smaller fluctuations compared to FASTER-LIO. In the liosam_1 sequence, the maximum (worst-case) processing latency of LIO-HKDT is 6.57 ms, which is significantly below the LiDAR frame interval (100 ms). This demonstrates that LIO-HKDT can maintain real-time performance even under peak computational load conditions.

2) Accuracy Evaluation: We use the root mean square error (RMSE) of Absolute Pose Error (APE) as the evaluation metric for pose estimation accuracy. Due to weather and GNSS interference, some public datasets lack reliable ground truth and are therefore excluded from evaluation. To ensure reliable comparison, we select 13 representative sequences with high-

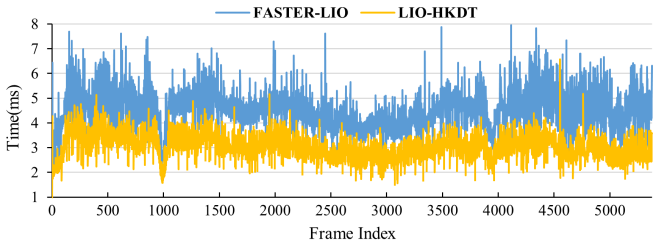


Fig. 3. Per-frame processing time of LIO-HKDT (yellow) and FASTER-LIO (blue) in liosam_1. The x-axis represents the LiDAR frame index, and the y-axis indicates the processing time per frame (ms).

TABLE V
RMSE OF ABSOLUTE TRANSLATION (M) ON SEQUENCES WITH RELIABLE GROUND TRUTH

	FAST-LIO2	FASTER-LIO	FASTER-LIO (i-Octree)	DLIO	LIO-SAM	LILI-OM	LIO-HKDT (Serial)	LIO-HKDT
nclt_1	1.81	1.96	1.73	477.68	×	×	2.30	**1.71**
nclt_2	1.53	1.31	1.37	115.20	×	×	1.35	**1.29**
nclt_3	2.97	2.31	2.30	314.28	×	×	**2.23**	2.74
nclt_4	2.24	**1.59**	1.69	190.84	×	×	1.70	1.85
nclt_5	**0.90**	1.00	0.91	1.36	1.13	1.35	1.02	0.96
m2dgr_1	0.19	0.19	0.19	0.35	0.24	0.41	**0.18**	**0.18**
m2dgr_2	0.34	0.34	0.34	0.45	**0.33**	1.09	**0.33**	**0.33**
m2dgr_3	0.23	0.23	0.23	0.43	**0.12**	0.36	0.21	0.21
liosam_1	0.51	0.52	0.52	0.66	0.66	1.15	0.51	**0.50**
utbm_1	12.36	13.15	12.64	**12.30**	-	18.87	13.48	13.19
utbm_2	**12.49**	12.85	12.86	15.48	-	14.70	12.64	12.66
utbm_4	14.83	15.43	15.09	13.12	-	**13.06**	15.86	14.48
utbm_7	**10.37**	11.13	10.73	55.83	-	15.69	10.86	11.18

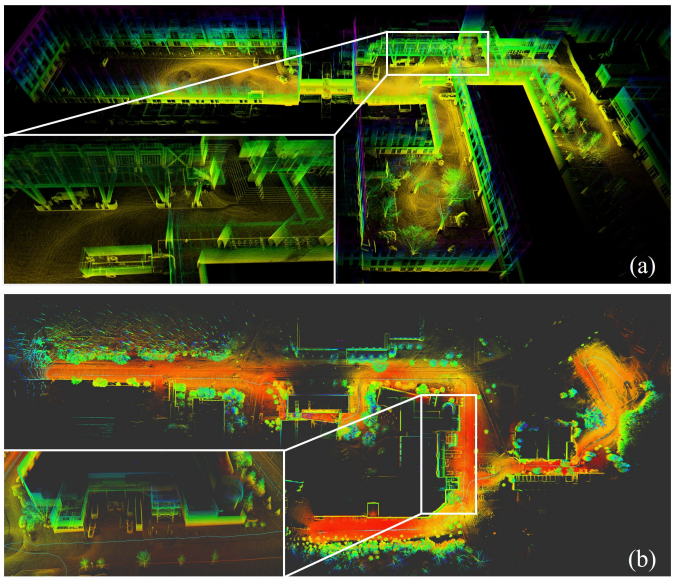
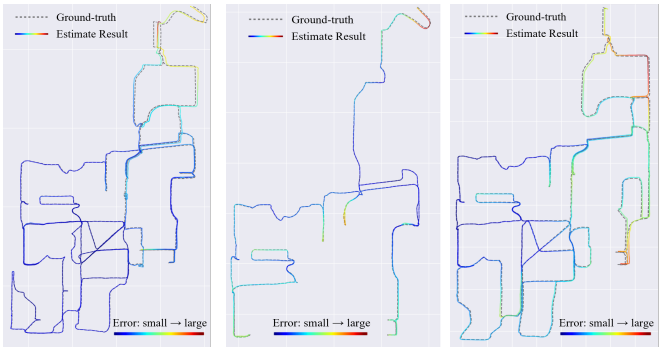


Fig. 4. LIO-HKDT mapping results on the liosam_2 (a) and nclt_5 (b) sequences, demonstrating accurate reconstruction and stable geometry alignment in different outdoor scenes.

quality ground truth from multiple datasets. Additionally, as LIO-HKDT is a pure odometry system without loop closure or global optimization, the loop closure modules in LILI-OM and LIO-SAM are disabled to ensure fair comparison.

As shown in Table V, LIO-HKDT achieves trajectory accuracy comparable to FAST-LIO2 and FASTER-LIO across most sequences. The performance differences observed across datasets are mainly attributed to the inherent randomness in LiDAR–inertial odometry systems, although minor influences from dataset characteristics or algorithmic sensitivity may also contribute. Similar to FAST-LIO2 and FASTER-LIO, the LIO-HKDT performs LiDAR–IMU data synchronization, point cloud preprocessing, and filter-based state estimation during each run, which may not be exactly identical across executions. As a result, slight fluctuations can be observed in the final estimated trajectories. Importantly, LIO-HKDT maintains comparable trajectory accuracy to FAST-LIO2 and FASTER-LIO while achieving significantly higher computational efficiency.

Fig. 4 presents the mapping results of LIO-HKDT in real-world environments, showing good global consistency, clear local details, minimal accumulated drift, and no noticeable ghosting artifacts. Fig. 5 shows representative trajectories from the long-range NCLT dataset, which includes sequences up to 4.01 km and 111 minutes. As illustrated, LIO-HKDT maintains stable and efficient performance throughout these challenging long-term sequences, demonstrating strong robustness and efficiency.



(a) nclt_1 (b) nclt_2 (b) nclt_3

Fig. 5. Ground-truth and estimated trajectories on the NCLT dataset. Dashed lines indicate the ground-truth trajectories, while colored solid lines show the estimated results. The color encodes the trajectory error, with red indicating larger errors and blue indicating smaller errors.

V. CONCLUSIONS

In this letter, we proposed an efficient data structure, hkd-Tree, to improve the efficiency of point cloud search and incremental update in LIO system. The hkd-Tree is a hash table where each key corresponds to a voxel index and each value to a local k-d tree. A voxel distribution mechanism and a buffered update strategy are further introduced to enhance its efficiency. Our method combines the localized search advantages of voxel-based methods with the efficient KNN search of k-d tree. It also avoids the computational overhead caused by exhaustive neighbor traversal and the maintenance of large global trees. Extensive experiments on randomly generated and real-world datasets show that hkd-Tree enables efficient KNN search and insertion, demonstrating its effectiveness in real-time LIO applications. While the hkd-Tree exhibits strong overall performance, its efficiency may decrease under certain extreme conditions. In cases of extremely sparse point clouds, its performance approaches that of iVox-Linear, whereas with very large voxel sizes, it degrades to a level comparable to a static k-d tree. Our future work will focus on preventing such degradation under extreme scenarios to further enhance the robustness of the proposed method.

REFERENCES

[1] Y. Wu, T. Guadagnino, L. Wiesmann, L. Klingbeil, C. Stachniss, and H. Kuhlmann, "Lio-ekf: High frequency lidar-inertial odometry using extended kalman filters," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 13 741–13 747.

[2] 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.

[3] L. Li, L. Schulze, and K. Kalavadia, "Promising slam methods for automated guided vehicles and autonomous mobile robots," *Procedia Computer Science*, vol. 232, pp. 2867–2874, 2024, 5th International Conference on Industry 4.0 and Smart Manufacturing (ISM 2023). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050924002813

[4] L. Yang, H. Ma, Z. Nie, H. Zhang, Z. Wang, and C. Wang, "3d lidar point cloud registration based on imu preintegration in coal mine roadways," *Sensors*, vol. 23, no. 7, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/7/3473

[5] X. Huang, G. Mei, J. Zhang, and R. Abbas, "A comprehensive survey on point cloud registration," *arXiv preprint arXiv:2103.02690*, 2021.

[6] J. Liu, Y. Zhang, X. Zhao, Z. He, W. Liu, and X. Lv, "Fast and robust lidar-inertial odometry by tightly-coupled iterated kalman smoother and robocentric voxels," *IEEE Transactions on Intelligent Transportation Systems*, vol. 25, no. 10, pp. 14 486–14 496, 2024.

[7] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, p. 129–147, Jun 1982. [Online]. Available: http://dx.doi.org/10.1016/0146-664x(82)90104-6

[8] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, p. 509–517, Sep 1975. [Online]. Available: http://dx.doi.org/10.1145/361002.361007

[9] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 1–4.

[10] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[11] Y. Cai, X. Wei, and F. Zhang, "ikd-tree: An incremental k-d tree for robotic applications," *Cornell University - arXiv,Cornell University - arXiv*, Feb 2021.

[12] J. Zhu, H. Li, Z. Wang, S. Wang, and T. Zhang, "i-octree: A fast, lightweight, and dynamic octree for proximity search," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 12 290–12 296.

[13] W. Xu, Y. Cai, D. He, J. Lin, and F. Zhang, "Fast-lio2: Fast direct lidar-inertial odometry," *IEEE Transactions on Robotics*, vol. 38, no. 4, pp. 2053–2073, 2022.

[14] 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," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 4861–4868, 2022.

[15] D. Lee, M. Jung, W. Yang, and A. Kim, "Lidar odometry survey: recent advancements and remaining challenges," *Intelligent Service Robotics*, vol. 17, no. 2, pp. 95–118, 2024.

[16] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Sensor fusion IV: control paradigms and data structures*, vol. 1611. Spie, 1992, pp. 586–606.

[17] A. Guttman, "R-trees," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*, Jan 1984. [Online]. Available: http://dx.doi.org/10.1145/602259.602266

[18] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, no. 2, p. 322–331, May 1990. [Online]. Available: https://doi.org/10.1145/93605.98741

[19] J. Elseberg, S. Magnenat, R. Siegwart, and A. Nüchter, "Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration," *Journal of Software Engineering for Robotics*, vol. 3, no. 1, pp. 2–12, 2012.

[20] J. L. Vermeulen, A. Hillebrand, and R. Geraerts, "A comparative study of k-nearest neighbour techniques in crowd simulation," *Computer Animation and Virtual Worlds*, vol. 28, no. 3–4, May 2017. [Online]. Available: https://doi.org/10.1002/cav.1775

[21] J. Zhang, S. Singh *et al.*, "Loam: Lidar odometry and mapping in real-time." in *Robotics: Science and systems*, vol. 2, no. 9. Berkeley, CA, 2014, pp. 1–9.

[22] 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)*, Oct 2018. [Online]. Available: http://dx.doi.org/10.1109/iros.2018.8594299

[23] W. Xu and F. Zhang, "Fast-lio: A fast, robust lidar-inertial odometry package by tightly-coupled iterated kalman filter," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 3317–3324, 2021.

[24] 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)*. IEEE, 2020, pp. 5135–5142.

[25] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *Computer Science Department, University of British Columbia, Vancouver, BC, Canada*, vol. 5, no. 6, 2009.

[26] 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.

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

[28] 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.

[29] J. Blanco and P. Rai, "Nanoflann: a c++ header-only fork of flann, a library for nearest neighbor (nn) with kd," *Trees*, 2014.

[30] M. Teschner, B. Heidelberger, M. Müller, D. Pomerantes, and M. Gross, "Optimized spatial hashing for collision detection of deformable objects." *Vision Modeling and Visualization,Vision Modeling and Visualization*, Jan 2003.

[31] K. Li, M. Li, and U. D. Hanebeck, "Towards high-performance solid-state-lidar-inertial odometry and mapping," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5167–5174, 2021.

[32] K. Chen, R. Nemiroff, and B. T. Lopez, "Direct lidar-inertial odometry: Lightweight lio with continuous-time motion correction," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 3983–3989.

[33] Z. Yan, L. Sun, T. Krajník, and Y. Ruichek, "Eu long-term dataset with multiple sensors for autonomous driving," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 10 697–10 704.

[34] J. Yin, A. Li, T. Li, W. Yu, and D. Zou, "M2dgr: A multi-sensor and multi-scenario slam dataset for ground robots," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 2266–2273, 2021.

[35] N. Carlevaris-Bianco, A. K. Ushani, and R. M. Eustice, "University of michigan north campus long-term vision and lidar dataset," *The International Journal of Robotics Research*, vol. 35, no. 9, pp. 1023–1035, 2016.