FAST-LIO2: Fast Direct LiDAR-inertial Odometry

Wei Xu*[1], Yixi Cai*[1], Dongjiao He[1], Jiarong Lin[1], Fu Zhang[1]

***Abstract*—This paper presents FAST-LIO2: a fast, robust, and versatile LiDAR-inertial odometry framework. Building on a highly efficient tightly-coupled iterated Kalman filter, FAST-LIO2 has two key novelties that allow fast, robust, and accurate LiDAR navigation (and mapping). 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 an incremental k-d tree data structure, *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 deg/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* are both open-sourced on Github[2,3].**

I. 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 [1]–[4] 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 [5]–[8] 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 [9, 10]. Over the last decade, LiDARs have been playing an increasingly important role in many autonomous robots, such as self-driving cars [11] and autonomous UAVs [12, 13]. 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 [14, 15], drawing much recent research interests [16]–[20]. 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.

40The 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 [16]. Furthermore, the feature extraction also varies from LiDAR to LiDAR, depending on the scanning pattern (e.g., spinning, prism-based [15], MEMS-based [14]) 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

*These two authors contribute equally to this work.

[1]All authors are with Department of Mechanical Engineering, University of Hong Kong. {*xuweii,yixicai,hdj65822,jiarong.lin*}*@connect.hku.hk, fuzhang@hku.hk*

[2]https://github.com/hku-mars/FAST_LIO

[3]https://github.com/hku-mars/ikd-Tree

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 develop an incremental k-d tree data structure, *ikd-Tree*, to represent a large dense point cloud map efficiently. Besides efficient nearest neighbor search, the new data structure 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. The *ikd-Tree* data structure toolbox is open-sourced on Github[3]. 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 [21]. 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 [22] 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. To further speed up the computation, a new and mathematically equivalent formula of computing the Kalman gain is used to reduce the computation complexity to the dimension of the state (as opposed to measurements). The new system is termed as FAST-LIO2 and is open-sourced at Github[2] 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 deg/s) and structure-less environments.

The remaining paper is organized as follows: In Section. II, we discuss relevant research works. We give an overview of the complete system pipeline and the details of each key components in Section. III, IV and V, respectively. The benchmark comparison on open datasets are presented in Section. VI and the real-world experiments are reported in Section. VII, followed by conclusions in Section. VIII.

II. Related Works

A. LiDAR(-Inertial) Odometry

Existing works on 3D LiDAR SLAM typically inherit the LOAM structure proposed in [23]. 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., $10Hz$) 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) [24]–[26] method wherein each iteration, several closest points in the map form a plane or edge where a target point belongs to. 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 (1-2Hz).

Subsequent LiDAR odometry works keep a framework similar to LOAM. For example, Lego-LOAM [27] introduces a ground point segmentation to decrease the computation and a loop closure module to reduce the long-term drift. Furthermore, LOAM-Livox [16] 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 increases odometry accuracy at the cost of increased computation 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 providing a good initial pose required by ICP. Moreover, the high-rate IMU measurements can effectively compensate for the motion distortion in a LiDAR scan. LION [28] is a loosely-coupled LiDAR inertial SLAM method that keeps the scan-to-scan registration of LOAM and introduces an observability awareness check into the *odometry* to lower the point number and hence save the computation. More tightly-coupled LiDAR-inertial fusion works [17, 29]–[31] 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 [29] presents a tightly-coupled LiDAR inertial fusion method where the IMU preintegrations are introduced into the *odometry*. LILI-OM [17] develops a new feature extraction method for non-repetitive scanning LiDAR and performs scan registration in a small map consisting of 20 recent LiDAR scans for the *odometry*. The *odometry* of LIO-SAM [30] requires a 9-axis IMU to produce attitude measurement as the prior of scan registration within a small local map. LINS [31] 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 [31]), sliding window joint optimization (LILI-OM [17] and LIOM [29]) and factor graph smoothing [32] (LIO-SAM [30]). Compared to the above methods, FAST-LIO [22] introduces a formal back-propagation that precisely considers the sampling time of every single point in a scan 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 new formula is proved to be mathematically equivalent to the conventional one but reduces the computation by several orders of magnitude. 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. The timely mapping of all recent scan points leads to increased odometry accuracy. However, to prevent the growing time of building a k-d tree of the map, the system can only work in small environments (e.g., hundreds of meters).

FAST-LIO2 builds on FAST-LIO [22] 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 propose 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 [28], which is however a loosely-coupled method as reviewed above. This idea is also very similar to the generalized-ICP (G-ICP) proposed in [26], 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 [33]. Other works based on Normal Distribution Transformation (NDT) [34]–[36] also register raw points, but NDT has lower stability compared to ICP and may diverge in some scenes [36].

B. 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 [37] 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 [38]. The R*-tree handles insertion by minimum overlap criteria and applies a forced re-insertion principle for the node splitting algorithm.

Octree [39] and k-dimensional tree (k-d tree) [40] 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 [41]. 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 [42, 43]. 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 [44], *libnabo* [43] and FLANN [45], 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 [46]–[49], 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 [50]. Another approach to enable incremental operations is maintaining a set of k-d trees in a logarithmic method similar to [51, 52] 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 $2^{(i-1)}M$ [53]. 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) [54].

We propose a dynamic data structure based on the scapegoat k-d tree [50], named incremental k-d tree (*ikd-Tree*), to achieve real-time mapping. Our *ikd-Tree* supports point-wise insertion

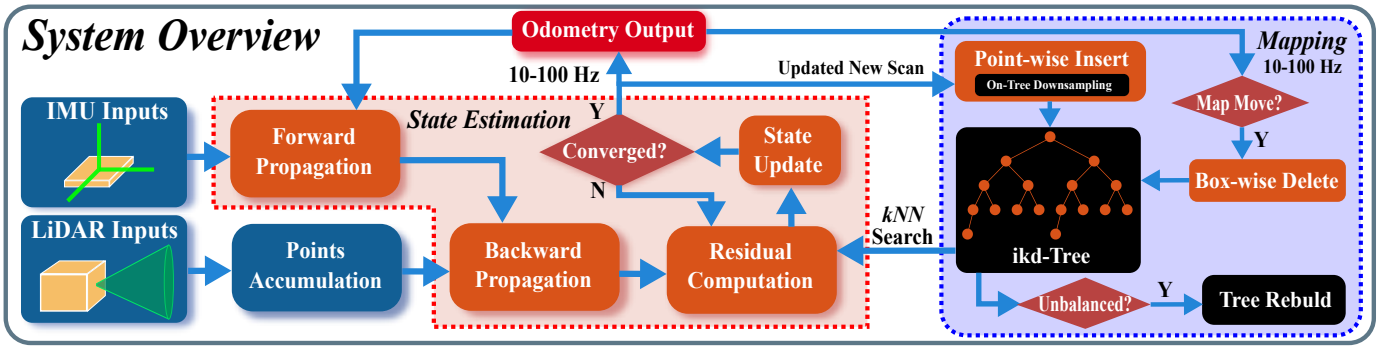


Fig. 1. System overview of FAST-LIO2.

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 [38, 39, 54]. 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 [50] and *nanoflann* k-d tree [54], 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.

III. System Overview

The pipeline of FAST-LIO2 is shown in Fig. 1. The sequentially sampled LiDAR raw points are first accumulated over a period between $10ms$ (for $100Hz$ update) and $100ms$ (for $10Hz$ 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. IV). 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. V). 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*).

IV. State Estimation

The state estimation of FAST-LIO2 is a tightly-coupled iterated Kalman filter inherited from FAST-LIO [22] 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 [22] for more details.

A. Kinematic Model

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

1) State Transition Model:

Take the first IMU frame (denoted as I) as the global frame (denoted as G) and denote ${}^{I}\mathbf{T}_{L} = ({}^{I}\mathbf{R}_{L},\ {}^{I}\mathbf{p}_{L})$ the unknown extrinsic between LiDAR and IMU, the kinematic model is:

$$\begin{aligned} {}^{G}\dot{\mathbf{R}}_{I} &= {}^{G}\mathbf{R}_{I}\lfloor\boldsymbol{\omega}_{m}-\mathbf{b}_{\boldsymbol{\omega}}-\mathbf{n}_{\boldsymbol{\omega}}\rfloor_{\wedge},\ {}^{G}\dot{\mathbf{p}}_{I} = {}^{G}\mathbf{v}_{I}, \\ {}^{G}\dot{\mathbf{v}}_{I} &= {}^{G}\mathbf{R}_{I}\left(\mathbf{a}_{m}-\mathbf{b}_{\mathbf{a}}-\mathbf{n}_{\mathbf{a}}\right)+{}^{G}\mathbf{g} \\ \dot{\mathbf{b}}_{\boldsymbol{\omega}} &= \mathbf{n}_{\mathbf{b}\boldsymbol{\omega}},\ \dot{\mathbf{b}}_{\mathbf{a}} = \mathbf{n}_{\mathbf{ba}}, \\ {}^{G}\dot{\mathbf{g}} &= \mathbf{0},\ {}^{I}\dot{\mathbf{R}}_{L} = \mathbf{0},\ {}^{I}\dot{\mathbf{p}}_{L} = \mathbf{0} \end{aligned} \tag{1}$$

where ${}^{G}\mathbf{p}_{I}$, ${}^{G}\mathbf{R}_{I}$ denote the IMU position and attitude 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. Based on the $\boxplus$ operation defined in [22], the continuous kinematic model (1) can be discretized at the IMU sampling period Δt [55]:

$$\mathbf{x}_{i+1} = \mathbf{x}_{i}\boxplus\left(\Delta t\mathbf{f}(\mathbf{x}_{i},\mathbf{u}_{i},\mathbf{w}_{i})\right) \tag{2}$$

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

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

$$\mathbf{x}\triangleq\begin{bmatrix}{}^{G}\mathbf{R}_{I}^{T} & {}^{G}\mathbf{p}_{I}^{T} & {}^{G}\mathbf{v}_{I}^{T} & \mathbf{b}_{\boldsymbol{\omega}}^{T} & \mathbf{b}_{\mathbf{a}}^{T} & {}^{G}\mathbf{g}^{T} & {}^{I}\mathbf{R}_{L}^{T} & {}^{I}\mathbf{p}_{L}^{T}\end{bmatrix}^{T}\in\mathcal{M}$$

$$\mathbf{u}\triangleq\begin{bmatrix}\boldsymbol{\omega}_{m}^{T} & \mathbf{a}_{m}^{T}\end{bmatrix}^{T},\ \mathbf{w}\triangleq\begin{bmatrix}\mathbf{n}_{\boldsymbol{\omega}}^{T} & \mathbf{n}_{\mathbf{a}}^{T} & \mathbf{n}_{\mathbf{b}\boldsymbol{\omega}}^{T} & \mathbf{n}_{\mathbf{ba}}^{T}\end{bmatrix}^{T}$$

$$\mathbf{f}(\mathbf{x},\mathbf{u},\mathbf{w})=\begin{bmatrix}\boldsymbol{\omega}_{m}-\mathbf{b}_{\boldsymbol{\omega}}-\mathbf{n}_{\boldsymbol{\omega}} \\ {}^{G}\mathbf{v}_{I}+\frac{1}{2}\left({}^{G}\mathbf{R}_{I}\left(\mathbf{a}_{m}-\mathbf{b}_{\mathbf{a}}-\mathbf{n}_{\mathbf{a}}\right)+{}^{G}\mathbf{g}\right)\Delta t \\ {}^{G}\mathbf{R}_{I}\left(\mathbf{a}_{m}-\mathbf{b}_{\mathbf{a}}-\mathbf{n}_{\mathbf{a}}\right)+{}^{G}\mathbf{g} \\ \mathbf{n}_{\mathbf{b}\boldsymbol{\omega}} \\ \mathbf{n}_{\mathbf{ba}} \\ \mathbf{0}_{3\times1} \\ \mathbf{0}_{3\times1} \\ \mathbf{0}_{3\times1}\end{bmatrix}\in\mathbb{R}^{24}$$

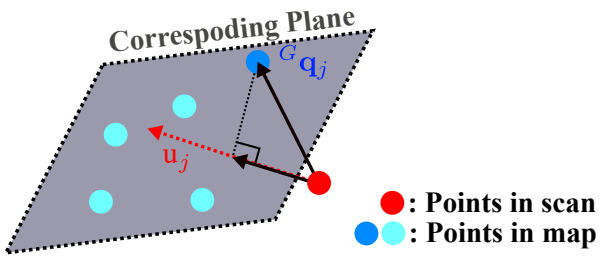


Fig. 2. The measurement model.

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 [22], 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 $^L\mathbf{p}_j^{\text{gt}}$:

$$^L\mathbf{p}_j^{\text{gt}} = {}^L\mathbf{p}_j + {}^L\mathbf{n}_j. \tag{3}$$

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.,

$$\mathbf{0} = {}^G\mathbf{u}_j^T \left({}^G\mathbf{T}_{I_k} {}^I\mathbf{T}_L \left({}^L\mathbf{p}_j + {}^L\mathbf{n}_j\right) - {}^G\mathbf{q}_j\right) \tag{4}$$

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. 2). It should be noted that the $^G\mathbf{T}_{I_k}$ and $^I\mathbf{T}_{L_k}$ 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 (4) to a more compact form as below:

$$\mathbf{0} = \mathbf{h}_j\left(\mathbf{x}_k, {}^L\mathbf{p}_j + {}^L\mathbf{n}_j\right), \tag{5}$$

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

B. Iterated Kalman Filter

Based on the state model (2) and measurement model (5) formulated on manifold $\mathcal{M} \triangleq SO(3)\times\mathbb{R}^{15}\times SO(3)\times\mathbb{R}^3$, we employ an iterated Kalman filter directly operating on the manifold $\mathcal{M}$ following the procedures in [55] and [22]. 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., $200Hz$ for IMU measurement and $10Hz \sim 100Hz$ for LiDAR scans), multiple propagation steps are usually performed before an update.

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

$$\begin{aligned}
&\widehat{\mathbf{x}}_{i+1} = \widehat{\mathbf{x}}_i \boxplus \left(\Delta t \mathbf{f}(\widehat{\mathbf{x}}_i, \mathbf{u}_i, \mathbf{0})\right); \ \widehat{\mathbf{x}}_0 = \bar{\mathbf{x}}_{k-1}, \\
&\widehat{\mathbf{P}}_{i+1} = \mathbf{F}_{\widetilde{\mathbf{x}}_i}\widehat{\mathbf{P}}_i\mathbf{F}_{\widetilde{\mathbf{x}}_i}^T + \mathbf{F}_{\mathbf{w}_i}\mathbf{Q}_i\mathbf{F}_{\mathbf{w}_i}^T; \ \widehat{\mathbf{P}}_0 = \bar{\mathbf{P}}_{k-1},
\end{aligned} \tag{6}$$

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 [55] and more concrete derivation in [22]):

$$\begin{aligned}
\mathbf{F}_{\widetilde{\mathbf{x}}_i} &= \frac{\partial(\mathbf{x}_{i+1} \boxminus \widehat{\mathbf{x}}_{i+1})}{\partial \widetilde{\mathbf{x}}_i}\Big|_{\widetilde{\mathbf{x}}_i=\mathbf{0},\ \mathbf{w}_i=\mathbf{0}} \\
\mathbf{F}_{\mathbf{w}_i} &= \frac{\partial(\mathbf{x}_{i+1} \boxminus \widehat{\mathbf{x}}_{i+1})}{\partial \mathbf{w}_i}\Big|_{\widetilde{\mathbf{x}}_i=\mathbf{0},\ \mathbf{w}_i=\mathbf{0}}
\end{aligned} \tag{7}$$

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 $\widehat{\mathbf{x}}_k, \widehat{\mathbf{P}}_k$.

2) Residual Computation: Assume the estimate of state $\mathbf{x}_k$ at the current iterate update (see Section. IV-B3) 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 (6). 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. V-A). 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 (4) and (5)). Moreover, approximating the measurement equation (5) by its first order approximation made at $\widehat{\mathbf{x}}_k^\kappa$ leads to

$$\begin{aligned}
\mathbf{0} &= \mathbf{h}_j\left(\mathbf{x}_k, {}^L\mathbf{n}_j\right) \simeq \mathbf{h}_j\left(\widehat{\mathbf{x}}_k^\kappa, \mathbf{0}\right) + \mathbf{H}_j^\kappa \widetilde{\mathbf{x}}_k^\kappa + \mathbf{v}_j \\
&= \mathbf{z}_j^\kappa + \mathbf{H}_j^\kappa \widetilde{\mathbf{x}}_k^\kappa + \mathbf{v}_j
\end{aligned} \tag{8}$$

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 $\mathbf{h}_j\left(\widehat{\mathbf{x}}_k^\kappa \boxplus \widetilde{\mathbf{x}}_k^\kappa, {}^L\mathbf{n}_j\right)$ with respect to $\widetilde{\mathbf{x}}_k^\kappa$, evaluated at zero, $\mathbf{v}_j \in \mathcal{N}(\mathbf{0}, \mathbf{R}_j)$ is due to the raw measurement noise $^L\mathbf{n}_j$, and $\mathbf{z}_j^\kappa$ is called the residual:

$$\mathbf{z}_j^\kappa = \mathbf{h}_j\left(\widehat{\mathbf{x}}_k^\kappa, \mathbf{0}\right) = \mathbf{u}_j^T\left({}^G\widehat{\mathbf{T}}_{I_k}^\kappa {}^I\widehat{\mathbf{T}}_{L_k}^\kappa {}^L\mathbf{p}_j - {}^G\mathbf{q}_j\right) \tag{9}$$

3) Iterated Update: The propagated state $\widehat{\mathbf{x}}_k$ and covariance $\widehat{\mathbf{P}}_k$ from Section. IV-B1 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:

$$\begin{aligned}
\mathbf{x}_k \boxminus \widehat{\mathbf{x}}_k &= \left(\widehat{\mathbf{x}}_k^\kappa \boxplus \widetilde{\mathbf{x}}_k^\kappa\right) \boxminus \widehat{\mathbf{x}}_k = \widehat{\mathbf{x}}_k^\kappa \boxminus \widehat{\mathbf{x}}_k + \mathbf{J}^\kappa \widetilde{\mathbf{x}}_k^\kappa \\
&\sim \mathcal{N}(\mathbf{0}, \widehat{\mathbf{P}}_k)
\end{aligned} \tag{10}$$

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

$$\mathbf{J}^{\kappa}=\begin{bmatrix}\mathbf{A}\left(\delta^{G}\boldsymbol{\theta}_{I_k}\right)^{-T} & \mathbf{0}_{3\times15} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3}\\ \mathbf{0}_{15\times3} & \mathbf{I}_{15\times15} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3}\\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times15} & \mathbf{A}\left(\delta^{I}\boldsymbol{\theta}_{L_k}\right)^{-T} & \mathbf{0}_{3\times3}\\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times15} & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3}\end{bmatrix} \tag{11}$$

where $\mathbf{A}(\cdot)^{-1}$ is defined in [22, 55], $\delta^{G}\boldsymbol{\theta}_{I_k}={}^{G}\widehat{\mathbf{R}}_{I_k}^{\kappa}\boxminus{}^{G}\widehat{\mathbf{R}}_{I_k}$ 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 (8):

$$-\mathbf{v}_j=\mathbf{z}_j^{\kappa}+\mathbf{H}_j^{\kappa}\widetilde{\mathbf{x}}_k^{\kappa}\sim\mathcal{N}(\mathbf{0},\mathbf{R}_j) \tag{12}$$

Combining the prior distribution in (10) with the measurement model from (12) 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):

$$\min_{\widetilde{\mathbf{x}}_k^{\kappa}}\left(\|\mathbf{x}_k\boxminus\widehat{\mathbf{x}}_k\|^2_{\widehat{\mathbf{P}}_k}+\sum\nolimits_{j=1}^{m}\|\mathbf{z}_j^{\kappa}+\mathbf{H}_j^{\kappa}\widetilde{\mathbf{x}}_k^{\kappa}\|^2_{\mathbf{R}_j}\right) \tag{13}$$

where $\|\mathbf{x}\|^2_{\mathbf{M}}=\mathbf{x}^T\mathbf{M}^{-1}\mathbf{x}$. This MAP problem can be solved by iterated Kalman filter as below (to simplify the notation, let $\mathbf{H}=[\mathbf{H}_1^{\kappa^T},\cdots,\mathbf{H}_m^{\kappa^T}]^T$, $\mathbf{R}=\mathrm{diag}\,(\mathbf{R}_1,\cdots\mathbf{R}_m)$,$\mathbf{P}=(\mathbf{J}^{\kappa})^{-1}\,\widehat{\mathbf{P}}_k(\mathbf{J}^{\kappa})^{-T}$, and $\mathbf{z}_k^{\kappa}=\left[\mathbf{z}_1^{\kappa^T},\cdots,\mathbf{z}_m^{\kappa^T}\right]^T$):

$$\begin{aligned}\mathbf{K}&=\left(\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}+\mathbf{P}^{-1}\right)^{-1}\mathbf{H}^T\mathbf{R}^{-1},\\ \widehat{\mathbf{x}}_k^{\kappa+1}&=\widehat{\mathbf{x}}_k^{\kappa}\boxplus\left(-\mathbf{K}\mathbf{z}_k^{\kappa}-(\mathbf{I}-\mathbf{K}\mathbf{H})(\mathbf{J}^{\kappa})^{-1}\left(\widehat{\mathbf{x}}_k^{\kappa}\boxminus\widehat{\mathbf{x}}_k\right)\right).\end{aligned} \tag{14}$$

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:

$$\bar{\mathbf{x}}_k=\widehat{\mathbf{x}}_k^{\kappa+1},\ \bar{\mathbf{P}}_k=(\mathbf{I}-\mathbf{K}\mathbf{H})\,\mathbf{P} \tag{15}$$

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:

$${}^{G}\bar{\mathbf{p}}_j={}^{G}\bar{\mathbf{T}}_{I_k}{}^{I}\bar{\mathbf{T}}_{L_k}{}^{L}\mathbf{p}_j;\ j=1,\cdots,m. \tag{16}$$

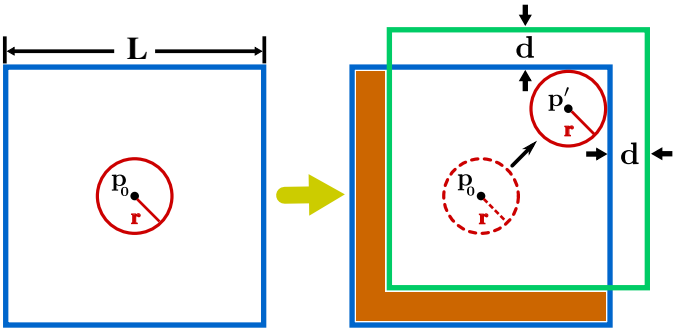
The transformed LiDAR points $\{{}^{G}\bar{\mathbf{p}}_j\}$ are inserted to the map represented by *ikd-Tree* (see Section. V). Our state estimation is summarized in **Algorithm 1**.

Algorithm 1: 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 (6);
Backward propagation to compensate motion [22];
$\kappa=-1$, $\widehat{\mathbf{x}}_k^{\kappa=0}=\widehat{\mathbf{x}}_k$;
repeat
  $\kappa=\kappa+1$;
  Compute $\mathbf{J}^{\kappa}$ via (11) 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 (8) (9);
  Compute the state update $\widehat{\mathbf{x}}_k^{\kappa+1}$ via (14);
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 (16).

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



(a) Initial map (b) Moving the map

Fig. 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*).

V. 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*. In order to prove the time efficiency of *ikd-Tree* theoretically, a complete analysis of time complexity is provided.

A. 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. 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 (15). 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 V-C.

B. Tree Structure and Construction

1) Data Structure: The *ikd-Tree* is a binary search tree. The attributes of a tree node in *ikd-Tree* is presented in **Data Structure**. Different from many existing implementations of k-d trees which store a "bucket" of points only on leaf nodes [43]–[45, 53, 54], our *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 [41], which is the case of our *ikd-Tree*. Since a point corresponds to a single node on the *ikd-Tree*, we will use points and nodes interchangeably. 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. V-C2 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.

Data Structure: Tree node structure

```
1 Struct TreeNode:
2     PointType point;
3     TreeNode * leftchild, * rightchild;
4     int axis;
5     int treesize, invalidnum;
6     bool deleted, treedeleted;
7     CuboidVertices range;
8 end
```

2) Construction: Building the *ikd-Tree* is similar to building a static k-d tree in [40]. 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.

C. Incremental Updates

The incremental updates on *ikd-Tree* refer to incremental operations followed by dynamic re-balancing detailed in Section. V-D. Two types of incremental operations are supported: point-wise operations and box-wise operations. The point-wise operations insert, delete or re-insert a single point to/from the k-d tree, while the box-wise operations insert, delete or re-insert all points in a given axis-aligned cuboid. In both cases, the point insertion is further integrated with on-tree downsampling, which maintains the map at a pre-determined resolution. In this paper, we only explain the point-wise insertion and box-wise delete as they are required by the map management of FAST-LIO2. Readers can refer to our open-source full implementation of *ikd-Tree* at Github repository[3] and technical documents contained therein for more details.

TABLE I
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.

1) Point Insertion with On-tree Downsampling: In consideration of robotic applications, our *ikd-Tree* supports simultaneous point insertion and map downsampling. The algorithm is detailed in **Algorithm 2**. For a given point $\mathbf{p}$ in $\{{}^{G}\bar{\mathbf{p}}_j\}$ from the state estimation module (see **Algorithm 1**) 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. V-C2.

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 I. 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. V-C3. 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. V-D.

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 `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.

Algorithm 2: Point Insertion with On-tree Downsampling

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

1 **Algorithm Start**
2 $\mathbf{C}_D \leftarrow$ `FindCube` $(l, \mathbf{p})$
3 $\mathbf{p}_{center} \leftarrow$ `Center` $(\mathbf{C}_D)$;
4 $V \leftarrow$ `BoxwiseSearch(`*RootNode*`,`$\mathbf{C}_D$`)`;
5 $V.push(\mathbf{p})$;
6 $\mathbf{p}_{nearest} \leftarrow$ `FindNearest` $(V, \mathbf{p}_{center})$;
7 `BoxwiseDelete(`*RootNode*`,`$\mathbf{C}_D$`)`
8 `Insert(`*RootNode*`,`$\mathbf{p}_{nearest}$`,`*NULL*`,`*SW*`)`;
9 **Algorithm End**
10
11 **Function** `Insert(`*T*`,` $\mathbf{p}$`,` *father*`,`*SW*`)`
12 **if** *T is empty* **then**
13 `Initialize(`*T*`,`$\mathbf{p}$`,` *father*`)`;
14 **else**
15 `ax` $\leftarrow$ `T.axis`;
16 **if** $\mathbf{p}[ax] <$ *T.point*$[ax]$ **then**
17 `Insert(`*T.leftchild*`,`$\mathbf{p}$`,`*T*`,`*SW*`)`;
18 **else**
19 `Insert(`*T.rightchild*`,`$\mathbf{p}$`,`*T*`,`*SW*`)`;
20 **end**
21 `AttributeUpdate(`*T*`)`;
22 `Rebalance(`*T*`,`*SW*`)`;
23 **end**
24 **End Function**

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. V-D).

Box-wise delete is implemented utilizing the range information in attribute `range` and the lazy labels on the tree nodes. As mentioned in V-B, the attribute *range* is represented by a circumscribed cuboid $\mathbf{C}_T$. The pseudo-code is shown in **Algorithm 3**. 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).

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.

Algorithm 3: Box-wise Delete

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

1 **Function** `BoxwiseDelete(`*T*`,`$\mathbf{C}_O$`,`*SW*`)`
2 $\mathbf{C}_T \leftarrow T.range$;
3 **if** $\mathbf{C}_T \cap \mathbf{C}_O = \varnothing$ **then return**;
4 **if** $\mathbf{C}_T \subseteq \mathbf{C}_O$ **then**
5 `T.treedelete`, `T.delete` $\leftarrow$ true;
6 `T.invalidnum` = `T.treesize`;
7 **else**
8 $\mathbf{p} \leftarrow$ `T.point`;
9 **if** $\mathbf{p} \subset \mathbf{C}_O$ **then** `T.treedelete` = true;
10 `BoxwiseDelete(`*T.leftchild*`,`$\mathbf{C}_O$`,`*SW*`)`;
11 `BoxwiseDelete(`*T.rightchild*`,`$\mathbf{C}_O$`,`*SW*`)`;
12 `AttributeUpdate(`*T*`)`;
13 `Rebalance(`*T*`,`*SW*`)`;
14 **end**
15 **End Function**

D. Re-balancing

The *ikd-Tree* actively monitors the balance property after each incremental operation and dynamically re-balances itself by re-building only the relevant sub-trees.

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 *ikd-Tree* is rooted at T. The sub-tree is α-balanced if and only if it satisfies the following condition:

$$\begin{aligned} S(T.leftchild) &< \alpha_{bal}\Big(S(T)-1\Big) \\ S(T.rightchild) &< \alpha_{bal}\Big(S(T)-1\Big) \end{aligned} \tag{17}$$

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

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

$$I(T) < \alpha_{del} S(T) \tag{18}$$

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

If a sub-tree of the *ikd-Tree* meets both criteria, 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 tree's maximum height. 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

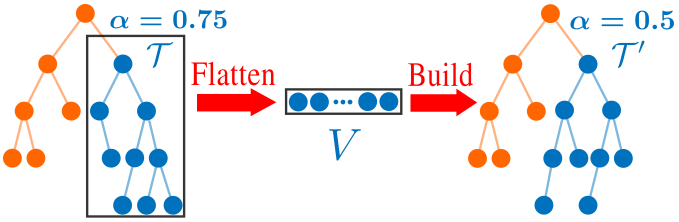


Fig. 4. Re-building an unbalanced sub-tree

nodes (i.e., labeled as "deleted") on the sub-trees are removed to reduce tree size. Reducing the height and size of the k-d tree allows highly efficient incremental operations and queries in the future.

2) Re-build & Parallel Re-build: Assuming re-building is triggered on a subtree $\mathcal{T}$ (see Fig. 4), the sub-tree is firstly flattened into a point storage array V. The tree nodes labeled as "deleted" are discarded during the flattening. A new perfectly balanced k-d tree is then built with all points in V as Section. V-B. When re-building a large sub-tree on the *ikd-Tree*, a considerable delay could occur and undermine the real-time performance of FAST-LIO2. To preserve high real-time ability, we design a double-thread re-building method. Instead of simply re-building in the second thread, our proposed method avoids information loss and memory conflicts in both threads by an operation logger, thus retaining full accuracy on k-nearest neighbor search at all times.

The re-building method is presented in **Algorithm 4**. When the balance criterion is violated, the sub-tree is re-built in the main thread when its tree size is smaller than a predetermined value $N_{\max}$; Otherwise, the sub-tree is re-built in the second thread. The re-building algorithm on the second thread is shown in function `ParRebuild`. 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., point insertion and delete) but not queries on this sub-tree (Line 12). 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 13). After the flattening, the original sub-tree is unlocked for the main thread to take further requests of incremental updates (Line 14). These requests will be simultaneously recorded in a queue named operation logger. Once the second thread completes building a new balanced k-d tree $\mathcal{T}'$ from the point array V (Line 15), the recorded update requests will be performed again on $\mathcal{T}'$ by function `IncrementalUpdates` (Line 16-18). Note that the parallel re-building switch is set to false as it is already in the second thread. After all pending requests are processed, the point information on the original sub-tree $\mathcal{T}$ is completely the same as that on the new sub-tree $\mathcal{T}'$ except that the new sub-tree is more balanced than the original one in the tree structure. The algorithm locks the node T from incremental updates and queries and replaces it with the new one T' (Line 20-22). Finally, the algorithm frees 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 proceeds still at the odometry rate

Algorithm 4: Rebuild (sub-) tree for re-balancing

Input: Root node T of (sub-) tree $\mathcal{T}$ 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
```

without any interruption, albeit at a lower efficiency due to the temporarily unbalanced k-d tree structure. We should 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).

E. K-Nearest Neighbor Search

Though being similar to existing implementations in those well-known k-d tree libraries [43]–[45], the nearest search algorithm is thoroughly optimized on the *ikd-Tree*. 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 [41]. A priority queue `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 `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 [43]. 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.

F. Time Complexity Analysis

The time complexity of *ikd-Tree* breaks into the time for incremental operations (insertion and delete), re-building, and k-nearest neighbor search. Note that all analyses are provided under the assumption of low dimensions (e.g., three dimensions in FAST-LIO2).

1) Incremental Operations: Since the insertion with on-tree downsampling relies on box-wise delete and box-wise search, the box-wise operations are discussed first. Suppose n denotes the tree size of the *ikd-Tree*, the time complexity of box-wise operations on the *ikd-Tree* is:

Lemma 1. *Suppose points on the ikd-Tree are in 3-d space $S_x \times S_y \times S_z$ and the operation cuboid is $\mathbf{C}_D = L_x \times L_y \times L_z$. The time complexity of box-wise delete and search of **Algorithm** 3 with cuboid $\mathbf{C}_D$ is*

$$O(H(n)) = \begin{cases} O(\log n) & \text{if} \Delta_{\min} \geqslant \alpha(\frac{2}{3}) \text{(*)} \\ O(n^{1-a-b-c}) & \text{if} \Delta_{\max} \leqslant 1-\alpha(\frac{1}{3}) \text{(**)} \\ O(n^{\alpha(\frac{1}{3})-\Delta_{\min}-\Delta_{\text{med}}}) & \text{if (*) and (**) fail and} \\ & \Delta_{\text{med}} < \alpha(\frac{1}{3}) - \alpha(\frac{2}{3}) \\ O(n^{\alpha(\frac{2}{3})-\Delta_{\min}}) & \text{otherwise.} \end{cases} \tag{19}$$

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 asymptotic time complexity for range search of an axis-aligned hypercube on a k-d tree is provided in [56]. The box-wise delete can be viewed as a range search except that lazy labels are attached to the tree nodes, which is $O(1)$. Therefore, the conclusion of range search can be applied to the box-wise delete and search on the *ikd-Tree* which leads to $O(H(n))$. ■

The time complexity of insertion with on-tree downsampling is given as

Lemma 2. *The time complexity of point insertion with on-tree downsampling in **Algorithm** 2 on ikd-Tree is $O(\log n)$.*

Proof. The downsampling method on the *ikd-Tree* is composed of box-wise search and delete followed by the point insertion. By applying **Lemma** 1, the time complexity of downsample is $O(H(n))$. Generally, the downsample cube $\mathbf{C}_D$ 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 the time complexity of $O(\log n)$.

The maximum height of the *ikd-Tree* can be easily proved to be $\log_{1/\alpha_{bal}}(n)$ from Eq. (17) while that of a static k-d tree is $\log_2 n$. Hence the lemma is directly obtained from [40] where the time complexity of point insertion on a k-d tree was proved to be $O(\log n)$. Summarizing the time complexity of both downsample and insertion concludes that the time complexity of insertion with on-tree downsampling is $O(\log n)$. ■

2) Re-build: The time complexity for re-building falls into two types: single-thread re-building and parallel double-thread re-building. In the former case, the re-building is performed by the main thread recursively. Each level takes the time of sorting (i.e., $O(n)$) and the total time over $\log n$ levels is $O(n \log n)$ [40] when the dimension k is low. 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 12-14) and tree update (which takes constant time $O(1)$, **Algorithm 4**, Line 20-22) but not building (which is performed in parallel by the second thread, **Algorithm 4**, Line 15-18), leading to time complexity of $O(n)$ (viewed from the main thread). In summary, the time complexity of re-building the *ikd-Tree* is $O(n)$ for double-thread parallel re-building and $O(n \log n)$ for single-thread re-building.

3) K-Nearest Neighbor Search: As the maximum height of the *ikd-Tree* is maintained no larger than $\log_{1/\alpha_{bal}}(n)$, where n is the tree size, the time complexity to search down from root to leaf nodes is $O(\log n)$. During the process of searching k-nearest neighbors on the tree, the number of backtracking is proportional to a constant $\bar{l}$ which is independent of the tree size [41]. Therefore, the expected time complexity to obtain k-nearest neighbors on the *ikd-Tree* is $O(\log n)$.

VI. 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. VI-C, 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 [15] and spinning LiDARs. The first dataset is from the work LILI-OM [17] and is collected by a solid-state 3D LiDAR Livox Horizon[4], 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 dataset is from the work LIO-SAM [30] in MIT campus and contains several sequences collected by a VLP-16 LiDAR[5] 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

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

[5]https://velodynelidar.com/products/puck-lite/

TABLE II
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 to 100 Hz through zero-order interpolation.

dataset "*utbm*" [57] is collected with a human-driving robocar in maximum 50 km/h speed which has two 10 Hz Velodyne HDL-32E LiDAR[6] and 100 Hz Xsens MTi-28A53G25 IMU. In this paper, we only consider the left LiDAR. The fourth dataset "*ulhk*" [58] 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*" [59] 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.II. The details about all the 37 sequences used in this section, including name, duration, and distance, are listed in Table. VIII of Appendix. A.

A. 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 [55]. In the default configuration, the local map size 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 2**) is set to $l = 0.5m$ 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 computation platform for benchmark comparison is a lightweight UAV onboard computer: DJI Manifold 2-C[7] 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[8] 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.

[6]https://velodynelidar.com/products/hdl-32e/
[7]https://www.dji.com/cn/manifold-2/specs
[8]https://www.khadas.com/vim3

B. Data structure Evaluation

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 [60], the Point Cloud Library implementation of octree [61] and the *nanoflann* [54] 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 V-C 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 the target cuboid $\mathbf{C}_D$ (see **Algorithm 2**). For *nanoflann* k-d tree, the target cuboid $\mathbf{C}_D$ is firstly split into small cubes whose edge length equals the minimal edge length of the cuboid. Then the points inside the circumcircle of each small cube are obtained by radius search, after which points outside the cuboid are filtered out via a linear approach while points inside the target cuboid $\mathbf{C}_D$ remain. Finally, similar to **Algorithm 2**, 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 (see Section. V-A), it is achieved by removing points within the specified cuboid 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.

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 of $[S-5\%S, S+5\%S]$ to obtain a sufficient number of samples. Fig. 5 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 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

TABLE III
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.

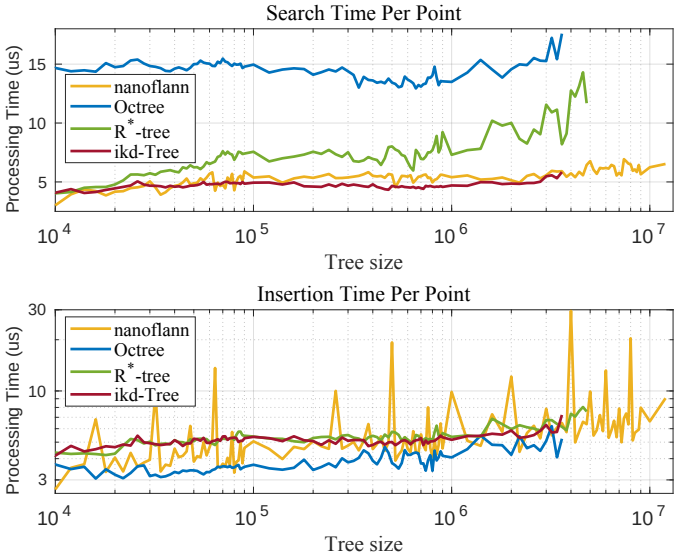


Fig. 5. Data structure comparison over different tree size.

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. V-F.

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 III. 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 [42, 43]. 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. 5. 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* does not heavily affect the overall real-time ability due to its low occurrence, it causes a catastrophic delay for subsequent control.

C. 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 [17], LIO-SAM [30], and LINS [31]. 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,

TABLE IV
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	8.63	6.66	21.01	6.59	30.59	5.72	17.14	4.62
FAST-LIO2 (1000m)	27.29	**51.6**	**16.8**	2.57	8.71	6.68	20.96	**6.58**	**30.08**	**5.56**	**16.29**	**4.58**
FAST-LIO2 (800m)	25.8	51.86	17.23	2.57	8.72	**6.65**	21.03	6.99	30.74	5.95	16.73	**4.58**
FAST-LIO2 (600m)	27.75	52.09	17.3	2.57	8.58	6.69	20.96	6.82	30.24	5.8	16.81	**4.58**
FAST-LIO2 (Feature)	27.21	53.81	22.59	2.61	**8.5**	7.82	**20.57**	6.77	31.17	6.09	16.61	7.85
LILI-OM	59.48	782.11	17.59	**2.29**	317.77	12.42	260.76	12.17	276.74	7.39	328.87	18.78
LIO-SAM	—[1]	—	—	3.52	9461	7.15	×[2]	22.26	44.83	7.43	1077.5	4.75
LINS	48.17	54.35	60.48	3.11	65.95	1051	243.87	378.99	106.03	11.13	2995.9	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.

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 [22] (optimized for solid-state LiDAR) and BALM [20] (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.

1) RMSE Benchmark: The RMSE are computed and reported in Table. IV. 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 LiDAR revisits a previous place. When the map size is over 2000 m, the accuracy increment is not persistent as the odometry drift may cause possible false point match 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.

Compared with other LIO methods, FAST-LIO2 or its variant achieves the best performances in 18 of all 19 data sequences and is the most robust LIO method among all the experiments. The only exception is on *ulhk_4* 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 [30] so the algorithm has been well-tuned for the data. However, FAST-LIO2 still achieves higher accuracy.

2) Drift Benchmark: The end-to-end errors are reported in Table. V. 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 [30]) 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.

D. Processing Time Evaluation

Table.VI 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.VI) 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 [17], incremental smoothing and mapping in LIO-SAM [30] and Map-refining in LINS [31]), whose average processing time per LiDAR scan

TABLE V
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.

are referred to as "Odo." and "Map." respectively in Table. VI. The two processing time is summed up to compare with FAST-LIO2.

From Table. VI, we can see that the FAST-LIO2 consumes considerably less time than other LIO methods, being x8 faster than LILI-OM, x10 faster than LIO-SAM, and x6 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, x3 faster than LILI-OM and over x2 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 VI-C) 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.

VII. Real-world Experiments

A. Platforms

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. 6), 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 VII-B2, so that the LiDAR is installed face-forward. The 750 mm wheelbase quadrotor UAV, developed by Ambit-Geospatial company[9], is used for the aerial scanning, see section VII-C, so that the LiDAR is facing down to the ground. In all platforms, we use a solid-state 3D LiDAR Livox Avia[10] 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. VI. 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.

Fig. 6. 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.

B. Private Dataset

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 $650m$. 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. 7, 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 [22] 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. VII. The preprocessing refers to data reception and formatting, which are identical for

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

TABLE VI
THE AVERAGE PROCESSING TIME PER SCAN BENCHMARK 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

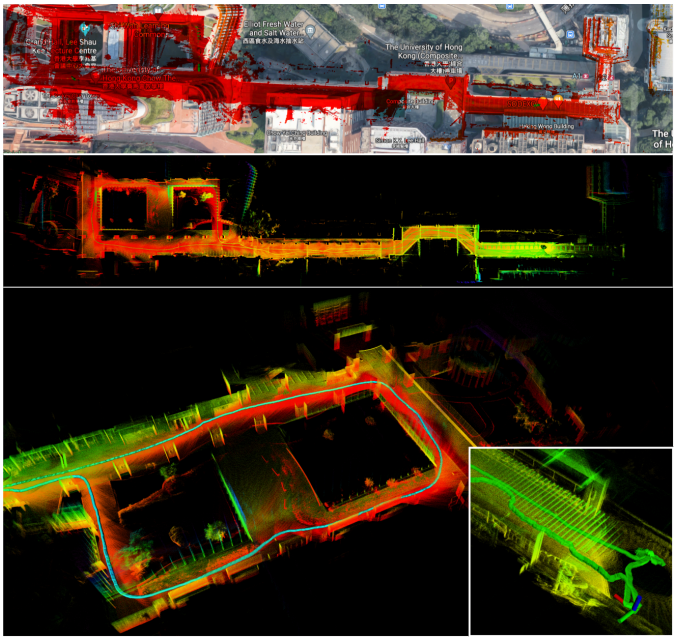


Fig. 7. Large-scale scene experiment.

TABLE VII
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

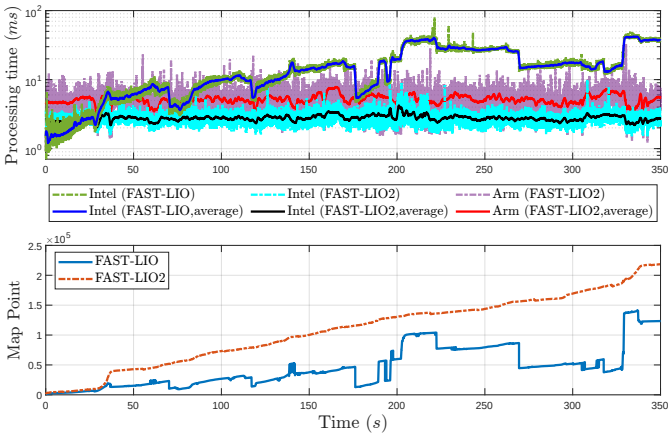


Fig. 8. The processing time for each LiDAR scan of FAST-LIO and FAST-LIO2.

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 (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.

The time consumption and the number of map points at each scan are shown in Fig. 8. As can be seen, the processing time for FAST-LIO2 running on the ARM processor occasionally

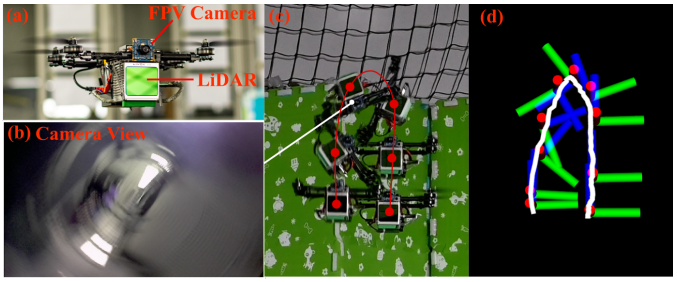


Fig. 9. The flip experiment. (a) the small scale UAV; (b) the onboard camera showing first person view (FPV) images during the flip; (c) the third person view images of the UAV during the flip; (d) the estimated UAV pose with FAST-LIO2.

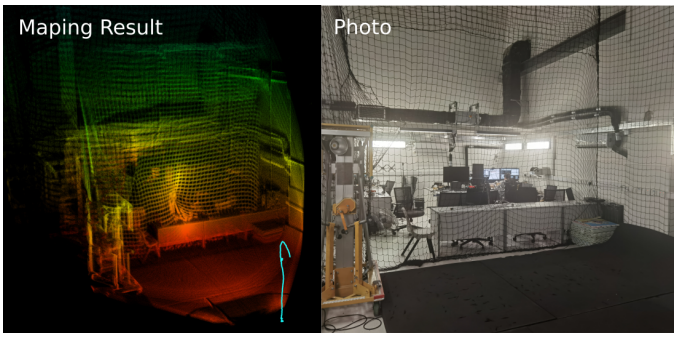


Fig. 10. The actual environment and the 3D map built by FAST-LIO2 during the flip.

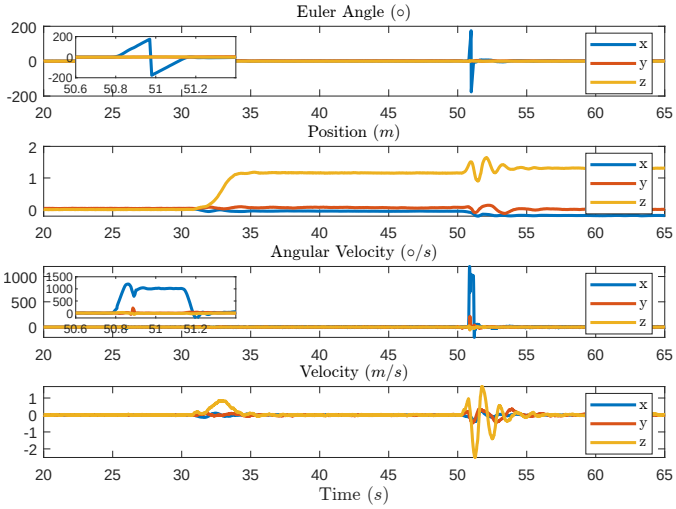


Fig. 11. The attitude, position, angular velocity and linear velocity in the UAV flip experiments.

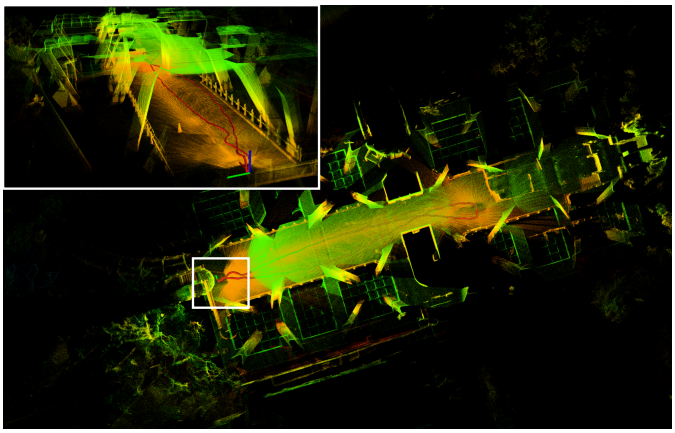


Fig. 12. The mapping results of FAST-LIO2 in the fast motion handheld experiment.

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 FAST-LIO builds a new k-d tree at every step, the building time has a time complexity $O(n \log n)$ [40] 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.

2) Aggressive UAV Flight Experiment: 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. 9. 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 [62] that takes state feedback from the FAST-LIO2. The pose estimated by FAST-LIO2 is shown in Fig. 9 (d), which agrees well with the actual UAV pose. The real-time mapping of the environment is shown in Fig. 10. In addition, Fig. 11 shows the position, attitude, angular velocity, and linear velocity during the experiments. The average and maximum angular velocity during the flip reaches 912 deg/s and 1198 deg/s, respectively (from 50.8 s to 51.2 s). FAST-LIO2 takes only 2.01 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 [63] 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) 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. 12). Fig. 13 shows the attitude, position, angular velocity, and linear velocity in the fast motion handheld experiments. It is seen that the maximum

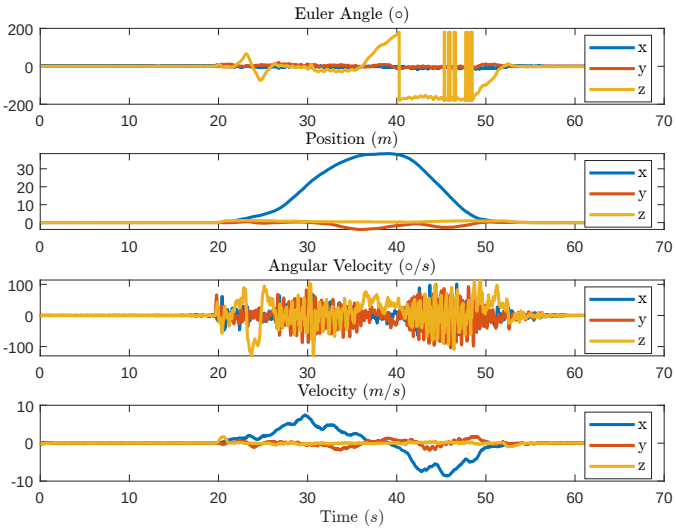


Fig. 13. The attitude, position, angular velocity and linear velocity in the fast motion handheld experiments.

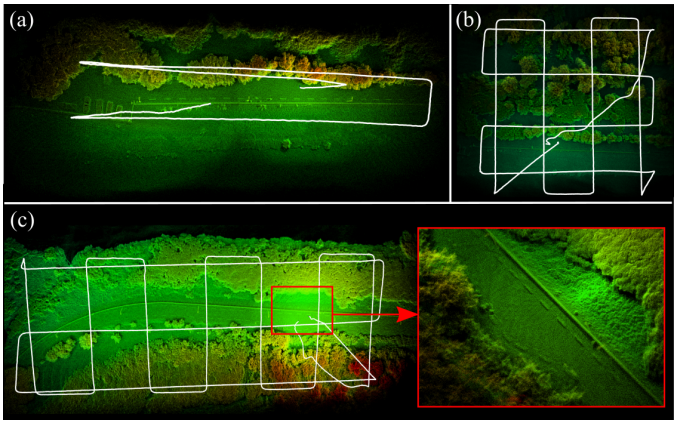


Fig. 14. Real-time mapping results with FAST-LIO2 for airborne mapping. The data is collected in the Hong Kong Wetland Park by a UAV with a down-facing Livox Avia LiDAR. The flight heights are 30 m (a), 30 m (b) and 50 m (c).

velocity reaches 7 m/s and angular velocity varies around ± 100 deg/s. 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. 13) while the total trajectory length is 81 m.

C. 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. 14. 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. 14 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 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.

VIII. 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. A novel incremental k-d tree (*ikd-Tree*) data structure, which supports dynamically point insertion, delete and parallel rebuilding, is developed and validated. 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.

Acknowledgement

This project is supported by DJI under the grant 200009538. The authors gratefully acknowledge DJI for the fund support, and Livox Technology for the equipment support during the whole work. The authors would like to thank Ambit-Geospatial for the helps in the outdoor aerial experiment. The authors also appreciate Zheng Liu, Guozheng Lu, and Fangcheng Zhu for the helpful discussions.

Appendix

The detail information about all 37 sequences used in Section. VI are listed in Table. VIII.

References

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

[2] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "On-manifold preintegration for real-time visual–inertial odometry," *IEEE Transactions on Robotics*, vol. 33, no. 1, pp. 1–21, 2016.

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

[4] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. Montiel, and J. D. Tardós, "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam," *IEEE Transactions on Robotics*, 2021.

TABLE VIII
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

[5] R. Newcombe, "Dense visual slam," Ph.D. dissertation, Imperial College London, 2012.

[6] 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.

[7] 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.

[8] 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.

[9] 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," *Journal of field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.

[10] 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," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[11] 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.

[12] 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," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1688–1695, 2017.

[13] F. Gao, W. Wu, W. Gao, and S. Shen, "Flying on point clouds: Online trajectory generation and autonomous navigation for quadrotors in cluttered environments," *Journal of Field Robotics*, vol. 36, no. 4, pp. 710–733, 2019.

[14] D. Wang, C. Watkins, and H. Xie, "Mems mirrors for lidar: A review," *Micromachines*, vol. 11, no. 5, p. 456, 2020.

[15] Z. Liu, F. Zhang, and X. Hong, "Low-cost retina-like robotic lidars based on incommensurable scanning," *IEEE/ASME Transactions on Mechatronics*, pp. 1–1, 2021.

[16] 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.

[17] 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.

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

[19] H. Wang, C. Wang, and L. Xie, "Lightweight 3-d localization and mapping for solid-state lidar," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1801–1807, 2021.

[20] Z. Liu and F. Zhang, "Balm: Bundle adjustment for lidar mapping," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 3184–3191, 2021.

[21] 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.

[22] 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*, pp. 1–1, 2021.

[23] J. Zhang and S. Singh, "Loam: Lidar odometry and mapping in real-time." in *Robotics: Science and Systems*, vol. 2, no. 9, 2014.

[24] G. C. Sharp, S. W. Lee, and D. K. Wehe, "Icp registration using invariant features," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 1, pp. 90–102, 2002.

[25] K.-L. Low, "Linear least-squares optimization for point-to-plane icp surface registration," *Chapel Hill, University of North Carolina*, vol. 4, no. 10, pp. 1–3, 2004.

[26] A. Segal, D. Haehnel, and S. Thrun, "Generalized-icp." in *Robotics: science and systems*, vol. 2, no. 4. Seattle, WA, 2009, p. 435.

[27] 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.

[28] 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," *arXiv preprint arXiv:2102.03443*, 2021.

[29] 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.

[30] 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.

[31] 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.

[32] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, "isam2: Incremental smoothing and mapping using the bayes tree," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 216–235, 2012.

[33] K. Koide, M. Yokozuka, S. Oishi, and A. Banno, "Voxelized gicp for fast and accurate 3d point cloud registration," *EasyChair Preprint*, no. 2703, 2020.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] D. Meagher, "Geometric modeling using octree encoding," *Computer graphics and image processing*, vol. 19, no. 2, pp. 129–147, 1982.

[40] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[41] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "an algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[42] 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, p. e1775, 2017.

[43] 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.

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

[45] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, no. 331-340, p. 2, 2009.

[46] 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.

[47] 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.

[48] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes," *Computer Graphics Forum*, vol. 26, no. 3, pp. 395–404, 2007.

[49] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 1–11, 2008.

[50] I. Galperin and R. L. Rivest, "Scapegoat trees." in *SODA*, vol. 93, 1993, pp. 165–174.

[51] J. L. Bentley and J. B. Saxe, "Decomposable searching problems i: Static-to-dynamic transformation," *J. algorithms*, vol. 1, no. 4, pp. 301–358, 1980.

[52] M. H. Overmars, *The design of dynamic data structures*. Springer Science & Business Media, 1987, vol. 156.

[53] 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.

[54] 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.

[55] D. He, W. Xu, and F. Zhang, "Embedding manifold structures into kalman filters," *arXiv preprint arXiv:2102.03804*, 2021.

[56] P. Chanzy, L. Devroye, and C. Zamora-Cura, "Analysis of range search for random kd trees," *Acta informatica*, vol. 37, no. 4-5, pp. 355–383, 2001.

[57] 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.

[58] 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.

[59] 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.

[60] 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/, September 2017.

[61] 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.

[62] G. Lu, W. Xu, and F. Zhang, "Model predictive control for trajectory tracking on differentiable manifolds," *arXiv preprint arXiv:2106.15233*, 2021.

[63] F. Kong, W. Xu, and F. Zhang, "Avoiding dynamic small obstacles with onboard sensing and computating on aerial robots," *arXiv preprint arXiv:2103.00406*, 2021.