2017 IEEE International Conference on Robotics and Automation (ICRA)
Singapore, May 29 - June 3, 2017

Improving Octree-Based Occupancy Maps using Environment Sparsity with Application to Aerial Robot Navigation

Jing Chen and Shaojie Shen

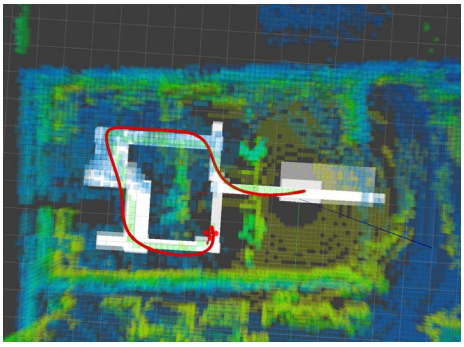
Abstract— **In this paper, we present an improved octree-based mapping framework for autonomous navigation of mobile robots. Octree is best known for its memory efficiency for representing large-scale environments. However, existing implementations, including the state-of-the-art OctoMap [1], are computationally too expensive for online applications that require frequent map updates and inquiries. Utilizing the sparse nature of the environment, we propose a ray tracing method with early termination for efficient probabilistic map update. We also propose a divide-and-conquer volume occupancy inquiry method which serves as the core operation for generation of free-space configurations for optimization-based trajectory generation. We experimentally demonstrate that our method maintains the same storage advantage of the original OctoMap, but being computationally more efficient for map update and occupancy inquiry. Finally, by integrating the proposed map structure in a complete navigation pipeline, we show autonomous quadrotor flight through complex environments.**

I. Introduction

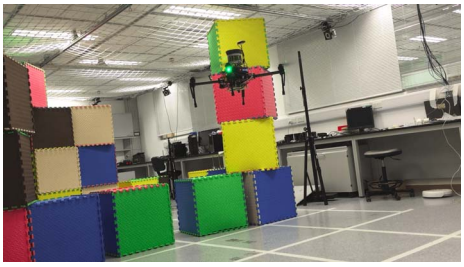
Autonomous navigation, which consists of systematic integration of localization, mapping, motion planning, and control, is the core capability of mobile robotic systems. The central component in such a complex system is the map, which is built from onboard sensor measurements, then stored within the onboard memory, and used by motion planning modules to achieve safe navigation. These steps run continuously and asynchronously throughout the mission to allow online exploration of complex environments. To this end, we identify that a good map structure should satisfy three important properties. First, it should be memory efficient, such that we can store maps covering large-scale environments using limited onboard storage. Second, it should be equipped with computationally efficiency map update routines in order to incorporate sensor measurements in an online fashion. The map update should be probabilistic in order to take the sensor noise into account. Finally, the map should support fast occupancy inquiry that serves as the core operation for most of the obstacle avoidance and motion planning algorithms. We note that optimization-based planning methods, such as our previous method for autonomous aerial navigation [3, 4], often requires free-space information as optimization constraints. Therefore, the occupancy inquiry of a volume, rather than a point, is required.

All authors are with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong, China. jchenbr@connect.ust.hk, eeshaojie@ust.hk

This work was supported by HKUST project R9341. The authors would also like to thank the research donation and equipment support from DJI.



(a) A visualization of the map structure (Sect. II-B) and the free-space flight corridor (Sect. III) generated using efficiency volume occupancy inquiry (Sect. II-C). The flight trajectory generated (Sect. III-C) within the corridor is shown in red.



(b) A snapshot of autonomous aerial navigation in unknown indoor environments

Fig. 1. We outfit a DJI Matrice 100 quadrotor with an Intel NUC (Intel Core i54250U, 16GB RAM) and a Velodyne VLP-16 laser range finder. State estimation is obtained by fusion of IMU measurements and laser-based odometry [2] using UKF. Thanks to the proposed map structure, we are able to incorporate all sensor information for map update in an online fashion. Occupancy checking, which is an essential operation for motion planning, can also be performed online. A video of the autonomous navigation experiment can be found at `http://www.ece.ust.hk/~eeshaojie/icra2017jing.mp4`.

The form of popular metric map structures can be roughly put into two categories: 1) obstacle-based; 2) spatially-organized. The former may simply represent the map as a unordered collection of 3D points, but such naive implementation cannot handle large number of obstacles. To achieve better computational efficiency, K-D tree [5] or octree [6] may be used to organize point cloud in a tree structure and provide fast closet point inquiry [7]. Hashing can also be used for rapid access to obstacles via spatial indexing [8, 9]. Nevertheless, obstacle-based representations are unfriendly for spatial fusion of sensor readings, and they also do not carry free-space information of the environment,

On the other hand, spatial map always preserves the occupancy information of regions which are organized according to their spatial configurations. Occupancy grid maps [10]

provide constant-time access to the spatial occupancy information of unit grids for map update and point collision inquiry, at the expense of large storage consumption. Probabilistic updates are done through integration of sensor measurements independently for each uniformly spaced grids. Although occupancy grids have been widely used in ground robotics [11, 12], it often fails to maintain affordable memory consumption for applications in full 3D environments. To overcome this problem, evaluation map [13] and multi-level surface map [14] build uniform 2D grids as the "foundation" for the 3D environment, and then store column-wise occupancy information within each grid. However, some obstacles, such as hanging wires and leaves, which are frequently encountered for flying robots, are not suitable to be represented as surface levels. More importantly, existing occupancy maps do not provides an efficient way to query about a large volume, thus preventing real-time use of optimization-based planning algorithms.

Utilizing the octree data structure, the authors in [1] proposed a multi-resolution grid map, OctoMap[1], for memory-efficient storing of occupancy information. This is accomplished by realizing the fact that most environments contain large volumes of free space that can be represented by a few large grids. OctoMap is widely used to reduce memory consumption for robots operating in 3D environments, such as quadrotors [15], humanoids [16], and underwater vehicles [17]. However, OctoMap treats range sensor update and occupancy inquiry in a uniform grid-based way and consequently is always slower than similar operations in uniform occupancy grid maps. For instance, OctoMap perform ray tracking by descretizing the ray into finest resolution grids and then check those grids one by one, where the complexity of each grid checking is linearly with respect to the logarithm of the number of total map grids. Moreover, volume occupancy inquiry in OctoMap can only be done by dividing the volume into finest resolution grids follow by checking every small grid sequentially. In summary, while the state-of-the-art OctoMap implementation shows significant advantage in terms of storage consumption, its computational complexity is insufficient for high speed autonomous navigation.

For aerial robot navigation, in order to generate the free space flight corridor for optimization-based trajectory generation [4], we are interested in efficient range inquiry of the occupancy information, which means checking the the occupancy status for a cubic volume specified by a range on each dimension. We realize that in many cases of the inquiry, we do not need to go all the way to the finest resolution grids. We may terminate the search if the coarser grid is able to provide enough occupancy information about the volume of interest. This is done by our divide-and-conquer occupancy inquiry method (Sect. II-C). Let the inquired volume be v_r and the finest map resolution be V_ϵ, our analysis (Sect. II-D) shows that the inquiry has the computational complexity of $O((v_r/V_\epsilon)^{2/3})$ on CPU and $O(\log(v_r/V_\epsilon))$ on GPU, comparing to the $O((v_r/V_\epsilon) * \log(v_r/V_\epsilon))$ complexity in the original OctoMap. Similar design principle holds for probabilistic map updates using ray tracing with early termination. When a sensor measurement hits a coarse grid that is confirmed to be occupied, we avoid going to finer levels, as we know that the measurement will not cause any further change to the map. Experimental results (Sect. IV-A) show that our ray tracing significantly outperforms the ray tracing in OctoMap, especially in open environments (Fig. 5(c)). Finally, we integrate the proposed octree-based mapping module into our motion planning system (Sect. III) onboard a quadrotor testbed and demonstrate autonomous navigation capability in 3D complex environments (Sect. IV-B).

[1] https://github.com/OctoMap/octomap

II. Improved Octree-Based Occupancy Map

Comparing to OctoMap [1], which is primarily designed to improve storage efficiency, our implementation optimizes not only for memory consumption, but also for computational efficiency. As such, we are able to satisfy all the three requirements mentioned in Sect. I for real-time onboard use of metric maps for autonomous navigation, by implementing two important operations on our octree-based occupancy map: 1) a ray tracing method (Sect. II-E) with early termination for updating map via spatial fusion of noisy sensor measurements; 2) a divide-and-conquer range inquiry method (Sect. II-C) for occupancy checking of an axis-aligned cuboid specified by a range on each dimension.

A. Probabilistic Sensor Data Fusion

In this section, we review the commonly-used sensor fusion method for probabilistic map update [1]. Specifically, given a sensor measurement $z_{1:t}$, the probability of a grid being occupied is $P(\mathrm{o}|z_{1:t})$, which is updated according to

$$P(\mathrm{o}|z_{1:t}) = \left[1 + \frac{1-P(\mathrm{o}|z_t)}{P(\mathrm{o}|z_t)} \frac{1-P(\mathrm{o}|z_{1:t-1})}{P(\mathrm{o}|z_{1:t-1})} \frac{P(\mathrm{o})}{1-P(\mathrm{o})}\right]^{-1} \tag{1}$$

where $P(o)$ is the prior probability, $P(o|z_{1:t-1})$ is the previous occupancy estimation, and $P(o|z_t)$ is the probability of a grid being occupied given the measurement z_t. Subsequently, utilizing log-odds notation $L(o) = \log\left[\frac{P(o)}{1-P(o)}\right]$, we rewrite (1) as

$$L(o|z_{1:t}) = L(o|z_{1:t}) + L(o|z_t) + L(o), \tag{2}$$

The log-odd formulation is more efficient and numerically stable, as it replaces the multiplications with addition operations and makes better use of the computer float unit range. L_{occ} is a chosen threshold value to define the occupied situation. Furthermore, we also adopt a clamping update like in OctoMap [1], by putting a lower threshold lo_{lb} and an upper threshold lo_{ub} in (2). In this way, the grid occupancy will converge to a stable status after confirmed by a fixed number of measurements, allowing our map to automatically compresses many small grids of identical occupancy status to a big grid (Sect. II) in order to reduce memory consumption and enable early termination in map updates (Sect. II-E).

TABLE I

DATA STRUCTURE OF THE OCTREE NODE.

attribute	data type	description
$\mathbf{lb}$	float[3]	lower boundaries
$\mathbf{ub}$	float[3]	upper boundaries
lo_{min}	float	minimum log-odd of its region
lo_{max}	float	maximum log-odd of its region
$chld$	*octree_node[8]	pointers of its child nodes

B. Octree Data Structure

The octree [18] data structure provides an efficient way for representing a large volume of girds with the same occupancy status. In the octree, we associate each map grid with a node, who has zero or eight *child* nodes representing the sub-grids of its octants. For simplicity, we use the tree node and its corresponding map grid interchangeably, as this should not cause any confusion in the context. Besides this, we follow the classical tree data structure conventions in the following discussion. For instance, a node is called a *leaf* node if it has no child node in the octree, otherwise the node is called a *inner* node. A node is the *parent* of its child node, and is the *ancestor* of the children and their "*descendants*". There is only one node with no parent and no ancestor, called the *root* node. A complete list of node attributes can be found in Table I.

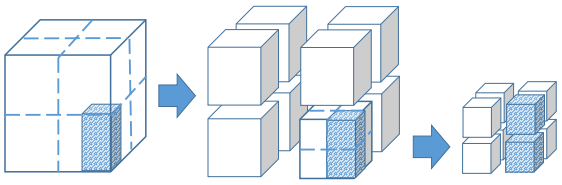


Fig. 2. An illustration of the octree structure.

The whole environment is enclosed by a huge grid that consists of eight sub-grids. This strategy is applied to each sub-grid recursively, until the grid satisfies any of the following two conditions: 1) it is completely occupied or empty, or 2) it reaches the finest resolution. Specifically, the sub-grids of a grid becomes the child nodes of the corresponding node in the octree, as shown in Fig.2. Note every inner node have maintained the summarized occupancy statics about its child nodes. Our range query (Sect. II-C) and ray tracing (Sect. II-E) starts from the root to leaves recursively. Both operations may terminate early if the requested data is already available or the map is already up-to-date.

To address the dynamic nature of the real-world flight environment, the map is equipped with automatic boundary expansion, grid pruning, and grid sub-division. When sensors observe obstacles that are out of the current map boundary, automatic expansion allows the octree to set up a new root node to preserve existing map data and simultaneously include the new obstacle in the map. Furthermore, an inner node will release its child nodes if all of them reach the same occupancy threshold as in (**??**), and therefore carry no more information than its parent node. On the other hand, a grid with size larger than the finest resolution shall be divided when an operation, such as ray tracing, affects the occupancy statics of its corresponding space. Note that an operation will only affect a node's occupancy statics if the occupancy log-odd has not reach the lower/upper threshold, otherwise the operation will terminate early.

We highlight the fact that the storage efficiency of octree-based occupancy map lies in the environment sparsity that a large part of the space mainly consists of large volumes with identical occupancy status, and therefore can be efficiently represented by a small set of big grids. Meanwhile, we shall notice that the map updates via ray tracing and volumetric occupancy checking, both originate from the same real world environment, enjoy the sparsity as well. We accelerate these operations through accessing the occupancy information stored in coarser grids, and try to avoid subsequent visits to finer grids by means of early termination. We will discuss the details in Sect. II-C and Sect. II-E.

C. Divide-and-Conquer Range Query

One of the core operation in our octree-based map is a range query method that checks the occupancy status of a volume that is specified by a range on each dimension. A range query (Algo. 1) is a function $f_r : \{\text{octree node pointers}\} \times \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \{\text{true}, \text{false}\}$ returning, in the space of a certain tree node, whether a specified axis-aligned cuboid contains any obstacle.

For collision check of 3D ranges with given lower bounds $\mathbf{lb}_r$ and upper bounds $\mathbf{ub}_r$, Algo. 1 starts with the root node by calling $RangeQuery(root, \mathbf{lb}_r, \mathbf{ub}_r)$. First of all, the function returns "false" whenever the current grid is not intersected with the queried range or the grid is entirely obstacle-free. Secondly, we check whether if the node is entirely occupied or is completely enclosed by the queried space, which means there is no need to fetch the child nodes for further information. If the condition holds, it will immediately return the answer and terminate. Otherwise, it will recursively performs the range query on each child nodes. Based on our complexity analysis in Sect. II-D, we found an upper bound complexity of querying a volume of v_r via Algo. 1 is $O((v_r/V_\epsilon)^{2/3})$ on single-core CPU. The complexity on a GPU with sufficient stream processors is reduced to $O(\log(v_r/V_\epsilon))$.

D. Complexity Analysis of Range Query

In this section, we provide an analysis of the runtime complexity of the divide-and-conquer range query (Algo. 1) in our octree structure. Let us consider the recursive procedure that we query the rectangular range on certain octree node.

We introduce the conception of *open* face and *close* face for the queried range in a cuboid shape based on the relation between map grid face and queried box face. Note that a queried range as well as the visited grid has six faces, each of which has an inner side and an outer side. A queried face is associated to the map grid face on the same orientation,

Algorithm 1 Divide-and-conquer range query

Require: the pointer of a octree node, nd; the lower bound of the queried range, $\mathbf{lb}_r$; the upper bound of the queried range, $\mathbf{ub}_r$;

1: **function** RANGEQUERY($nd, \mathbf{lb}_r \in \mathbb{R}^3, \mathbf{ub}_r \in \mathbb{R}^3$)
2: **if** $nd.lo_{max} < lo_{occ} \vee$
3: $\max(\mathbf{lb}_r, nd.\mathbf{lb}) \succeq \min(\mathbf{ub}_r, nd.\mathbf{ub})$ **then**
4: **return** false; //Early termination
5: **else if** $nd.lo_{min} \geq lo_{occ} \vee$
6: $(\mathbf{lb}_r \preceq nd.\mathbf{lb} \wedge \mathbf{ub}_r \succeq nd.\mathbf{ub})$ **then**
7: **return** true; //Early termination
8: **end if**
9: $res :=$ false;
10: **for** $ch \in nd.chld$ **do** //divide-and-conquer
11: $res := res \vee$RANGEQUERY($ch, \mathbf{lb}_r, \mathbf{ub}_r$);
12: **end for**
13: **return** res;
14: **end function**

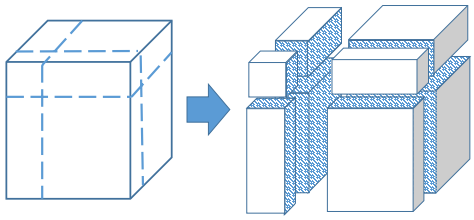


Fig. 3. Separation of a queried cuboid.

and is called *"close"* if it locates at or on the outer side of the map grid face. Contrarily, a cuboid face is *"open"* if it is unknown or on the inner side of the associated grid face. As a result, a queried cuboid with all its faces being close must completely enclose the visited map grid which implies the requested occupancy information can be directly obtained from the current node without accessing its child nodes.

The queried range with n_o open faces and volume of r_q is denoted as $\mathbf{r}_q^{n_o}$, and the problem size of the inquiry is measured by the volume of the visited node, v_q. Despite the fact that there can be different distributions of n_o open faces on a cuboid, only one situation will appear in our division and will be discussed below. Hence, for brevity, we do not distinguish the different distributions in our notation, such that $\mathbf{r}_q^{n_0}$ only stands for the involved case.

Denote the complexity of the range query by $g(\mathbf{r}_q^{n_o}, v_q)$ and abbreviate it as $g_{n_o}(v_q)$. We note that, as shown in Fig. 3 where the shadow represents the cutting faces created by the query division, [2] the cutting faces become close in the recursive sub-procedures. In this way, a cuboid with all the faces open is partitioned into at most eight cuboids having three open faces. Notice that the open faces must be distributed in a certain configuration that they are adjacent to each other. For running on a CPU, we conclude that

$$g_6(v_r) \leq O(1) + 8g_3(v_r/8). \tag{3}$$

Similarly, we examine the relevant occasions:

$$\begin{aligned} g_3(v_r) \leq & O(1) + g_0(v_r/8) + \\ & 3g_1(v_r/8) + 3g_2(v_r/8) + g_3(v_r/8), \end{aligned} \tag{4}$$

$$\begin{aligned} g_2(v_r) \leq & O(1) + 2g_0(v_r/8) + \\ & 4g_1(v_r/8) + 2g_2(v_r/8), \end{aligned} \tag{5}$$

$$g_1(v_r) \leq O(1) + 4g_0(v_r/8) + 4g_1(v_r/8), \tag{6}$$

$$g_0(v_r) = O(1), \tag{7}$$

$$g_{n_o}(v_r) = O(1), \text{ where } v_r \leq V_\epsilon. \tag{8}$$

Here, (7) results from the early termination step of Algo. 1 and (8) suggests it reaches the leaf node.

Applying the Master Theorem [19], from (6) and (7), we have

$$g_1(v_r) \leq O(1) + 4O(1) + 4g_1(v_r/8) \tag{9}$$

$$= O((v_r/V_\epsilon)^{\log_8 4}) = O((v_r/V_\epsilon)^{2/3}). \tag{10}$$

Plug (10) into (5):

$$\begin{aligned} g_2(v_r) \leq & O(1) + 2O(1) + 4O((v_r/8/V_\epsilon)^{2/3}) + \\ & 2g_2(v_r/8) \end{aligned} \tag{11}$$

$$= O((v_r/V_\epsilon)^{2/3}) + 2g_2(v_r/8) \tag{12}$$

$$= O((v_r/V_\epsilon)^{2/3}). \tag{13}$$

In the end, we plug in the result of (13) and get

$$\begin{aligned} g_3(v_r) \leq & O(1) + O(1) + 3O((v_r/8/V_\epsilon)^{2/3}) + \\ & 3O((v_r/8/V_\epsilon)^{2/3}) + g_3(v_r/8)) \end{aligned} \tag{14}$$

$$= O((v_r/V_\epsilon)^{2/3}) + g_3(v_r/8) \tag{15}$$

$$= O((v_r/V_\epsilon)^{2/3}). \tag{16}$$

Likewise, for the range query on a GPU with sufficient stream processors, we have each divided procedure executed by a different stream processor. Therefore, we have:

$$g_6^g(v_r) \leq O(1) + g_3(v_r/8). \tag{17}$$

$$\begin{aligned} g_3^g(v_r) \leq & O(1) + \max(g_0(v_r/8), g_1(v_r/8), \\ & g_2(v_r/8), g_3(v_r/8)), \end{aligned} \tag{18}$$

$$\begin{aligned} g_2^g(v_r) \leq & O(1) + \max(g_0(v_r/8), g_1(v_r/8), \\ & g_2(v_r/8)), \end{aligned} \tag{19}$$

$$g_1^g(v_r) \leq O(1) + \max(g_0(v_r/8), g_1(v_r/8)), \tag{20}$$

$$g_0^g(v_r) = O(1), \tag{21}$$

$$g_{n_o}^g(v_r) = O(1), \text{ where } v_r \leq V_\epsilon. \tag{22}$$

Again, utilizing the Master Theorem, the parallel range query implemented on a GPU has a complexity of $g_6^g(v_r) = O(\log(v_r/V_\epsilon))$.

[2]This is not the actual implementation but a conceptual division, as in Algo. 1.

E. Ray Tracing Utilizing Map Sparsity

Many sensors that have direct depth measurement capability, such as laser scanners, stereo cameras, and RGB-D sensors, carry not only the information about obstacles they sense, but also the information to tell that the space covered by the ray towards the obstacle is unoccupied. Probabilistic map update (Sect. II-A) therefore requires ray tracing to clear out false obstacle detections. A naive implementation of ray tracing is known to be time consuming on data structure that does not have constant time spatial access, such as the octree. In this section, we present a ray tracing that utilize the environment sparsity to achieve early termination, leading to significant computation savings.

Formally, a ray is defined by a function $f(\tau) = \mathbf{p}_r + \tau \cdot \mathbf{d}_r$, where $\mathbf{p}_r \in \mathbb{R}^3$ is the ray origin, $\mathbf{d}_r \in \mathbb{R}^3$ is the ray direction, and $\tau \in [0, 1]$. Our ray tracing (Algo. 2) is a procedure with inputs of the visited node nd, the ray origin $\mathbf{p}_r$, the ray direction $\mathbf{d}_r$, and the measurement log-odd lo_r. Starting from the root node by calling $RayTracing(root, \mathbf{p}_r, \mathbf{d}_r, lo_r)$, the algorithm recursively call itself on the children of the visited octree node unless the node is of the finest resolution. After updating data on the leaves, the ray tracing renew the parent node using the occupancy status of its children.

We emphasize that the sparsity of the map means that large volumes with the same occupancy status can be efficiently represented by a small number of big grids, and we cannot increase/decrease the occupancy probabilities that already reach the upper/lower threshold. Meanwhile, we thereby adopt the voxel ray tracing tactic in [20] to take advantage of the sparsity of the environment. This is done through early termination (step 1-5 of Algo. 2) such that the operation is terminated as long as further changes are impossible.

As the occupancy information has changed, we arrange the automatic sub-division (step 13-15 of Algo. 2) and pruning (step 23-35 of Algo. 2) to maintain a minimal storage consumption for storing necessary information. In this way, only when the ray tracing requests to access the sub-space of a non-"atom" leaf node, will the node be further divided. On the other hand, an inner node become a leaf by releasing its child nodes once all its children achieve the same cut-off occupancy threshold.

In the worst case, the algorithm runs at $O(\log(v_b/V_\epsilon))$ on a GPU and $O(\log(v_b/V_\epsilon) \cdot L_r/V_\epsilon^{1/3})$ on a CPU, where L_r stands for the ray length, V_ϵ denotes the volume resolution of the finest map grids, and v_b is the volume of a axis-aligned bounding box that encloses the ray. Again, we claim that the algorithm runs much faster in practice, exploiting the map sparsity via early termination, such that it is capable of inserting laser scan, consisting of more than $10,000$ rays, in a few milliseconds (Fig. 5 and Fig. 7(b)).

Algorithm 2 Divide-and-conquer voxel ray tracing

Require: the pointer of a octree node, nd; the ray origin, $\mathbf{p}_r$; the ray direction, $\mathbf{d}_r$; the measurement log-odd lo_r;

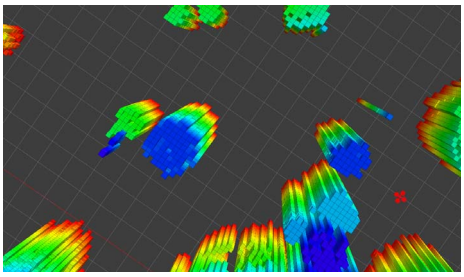
1: **function** RAYTRACING($nd, \mathbf{p}_r \in \mathbb{R}^3, \mathbf{d}_r \in \mathbb{R}^3, lo_r$)
2: **if** $(lo_r < 0 \wedge nd.lo_{max} = lo_{lb}) \vee$
3: $(lo_r > 0 \wedge nd.lo_{min} = lo_{ub})$ **then**
4: **return** ; //Early termination due to map sparsity
5: **end if**
6: **if** ISINTERSECTED($nd, \mathbf{p}_r, \mathbf{d}_r$) **then**
7: **if** ISFINEST(nd) **then**
8: $nd.lo_{min} :=$
9: $\min(\max(nd.lo_{min} + lo_r, lo_{lb}), lo_{ub})$;
10: $nd.lo_{max} :=$
11: $\min(\max(nd.lo_{max} + lo_r, lo_{lb}), lo_{ub})$;
12: **else**
13: **if** $nd.chld = \emptyset$ **then** //Automatic sub-division
14: DIVIDENODE(nd);
15: **end if**
16: $nd.lo_{min} := lo_{ub}$;
17: $nd.lo_{max} := lo_{lb}$;
18: **for** $ch \in nd.chld$ **do** //Divide-and-Conquer
19: RAYTRACING($ch, \mathbf{p}_r, \mathbf{d}_r, lo_r$);
20: $nd.lo_{min} := \min(nd.lo_{min}, ch.lo_{min})$;
21: $nd.lo_{max} := \max(nd.lo_{max}, ch.lo_{max})$;
22: **end for**
23: **if** $nd.lo_{min} = lo_{ub} \vee nd.lo_{max} = lo_{lb}$ **then**
24: PRUNENODE(nd); //Automatic pruning
25: **end if**
26: **end if**
27: **end if**
28: **end function**

III. Trajectory Generation for Autonomous Aerial Navigation

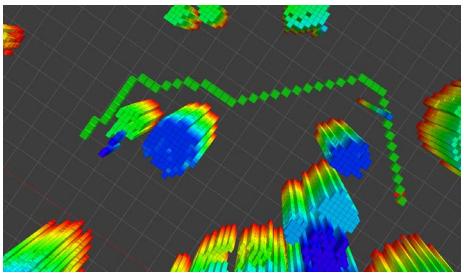
As the proposed map is intended for online navigation in unknown cluttered environment, it is integrated into our optimization-based trajectory planning framework [4], which is briefly presented in this section for completeness. We first construct a collision-free flight corridor, and then generate a smooth trajectory that fits entirely within the corridor to guarantee safety, as shown in Fig. 4. Our map have a significant impact on the overall performance (Sect. IV-B) by providing online spatial fusion of 3-D LiDAR data and a volumetric occupancy checking for real-time generation of the flight corridor.

A. Flight Corridor Generation

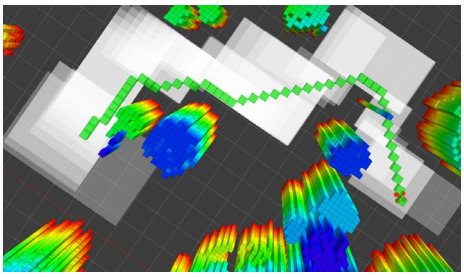
The main idea of the flight corridor is to approximate the free space that the optimal trajectory will most likely pass through, as shown in Fig. 4(b) and Fig. 4(c). We start with an A^* method to search a collision-free grid path to initialize the flight corridor. Each candidate grid is validated by our range query operation (Algo. 1). Afterwards, each of the corridor grids is inflated to contain more usable space. Since grid inflation is done via a range query-based binary search [4], our range query plays a critical role in achieving real-time efficiency for onboard corridor generation (Sect. IV-B).



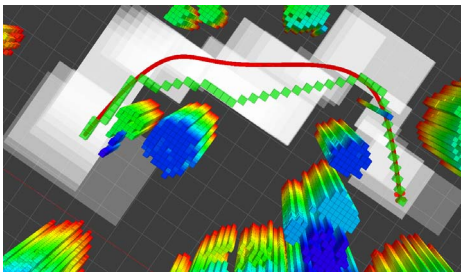
(a) The occupancy map



(b) The flight corridor (Sect. III-A) is initialized by classical A^* algorithm. The candidate corridor grids are validated by our range query (Algo. 1).



(c) The flight corridor (Sect. III-A) is inflated via range query to obtain more usable space.



(d) The trajectory (Sect. III-B) that fits entirely the flight corridor is generated using quadratic programming (Sect. III-C).

Fig. 4. Illustration of the trajectory generation (Sect. III). Cubes in different colors are obstacles at different altitudes. Green grids represent the initial corridor generated by A^* search. White patches on the ground illustrate the inflated corridor regions. The smooth trajectory is shown in red.

B. Quadrotor Dynamics and Trajectory Formulation

Denote the quadrotor position and angular velocity as $\mathbf{r}$ and $\mathbf{\Omega}$, we write the equation of quadrotor motion as

$$\begin{aligned} m\ddot{\mathbf{r}} &= -mg\mathbf{z}_W + f\mathbf{z}_B \\ \mathbf{M} &= \mathbf{J}\dot{\mathbf{\Omega}} + \mathbf{\Omega} \times \mathbf{J}\mathbf{\Omega} \end{aligned}, \tag{23}$$

where f, $\mathbf{M}$, $\mathbf{J}$ is the total thrust, moment and the inertial matrix, respectively, and $\mathbf{z}_B$ and $\mathbf{z}_W$ represents the vertical axes in the body and the world frame.

[21] shows that quadrotor UAVs enjoy the *differential flatness* that their control signal can be determined via four flat outputs $\{x, y, z, \psi\}$, which is the combination of the position and the yaw angle, and their derivatives. As such, the smooth trajectory for the motion of a quadrotor $\mathbf{f} = [x, y, z]^T$ can be presented in a piecewise-polynomial form [21]:

$$\mathbf{f}(t) = \begin{cases} \sum_{j=0}^{N} \mathbf{a}_{1j}(t-t_0)^j & t_0 \le t \le t_1 \\ \sum_{j=0}^{N} \mathbf{a}_{2j}(t-t_1)^j & t_1 \le t \le t_2 \\ \vdots & \\ \sum_{j=0}^{N} \mathbf{a}_{Mj}(t-t_{M-1})^j & t_{M-1} \le t \le t_M \end{cases}, \tag{24}$$

where $\mathbf{a}_{ij} \in \mathbb{R}^3$, N and M are the polynomial coefficients, the polynomial degree and the number of the trajectory segments which equals the number of grids in the flight corridor, and t_i specifies the end-time of the $(i-1)^{th}$ segment as well as the start-time of the i^{th}segment. We use omni-directional sensor (Fig. 1) to avoid planning the yaw angle.

C. Trajectory Generation via Quadratic Programming

The trajectory coefficients is solved by minimizing the integral of the N_ϕ^{th} derivative of the quadrotor motion:

$$min \int_{t_0}^{t_M} \left(\frac{d^{N_\phi}\mathbf{f}(t)}{dt^{N_\phi}} \right)^2 dt, \tag{25}$$

which have a quadratic form $\mathbf{p}^{\mathrm{T}}\mathbf{H}\mathbf{p}$, where $\mathbf{p}$ is the collection of all the polynomial coefficients and $\mathbf{H}$ is the corresponding Hessian matrix. Its construction is omitted for brevity.

In our previous work [3, 4], we showed that corridor constraints (Sect. III-A) can be written as linear inequalities ($\mathbf{A}_{lq}\mathbf{p} \le \mathbf{b}_{lq}$) form. Consequently, the trajectory generation becomes a quadratic programming (QP) problem, which can be solved efficiently online:

$$\begin{aligned} \min \quad & \mathbf{p}^{\mathrm{T}}\mathbf{H}\mathbf{p} \\ \text{s.t.} \quad & \mathbf{A}_{eq}\mathbf{p} = \mathbf{b}_{eq} \\ & \mathbf{A}_{lq}\mathbf{p} \le \mathbf{b}_{lq}. \end{aligned} \tag{26}$$

An example trajectory is shown in Fig. 4(d).

IV. Experimental Results

A. Ray Tracing Comparison against OctoMap

In this experiment, we compare our octree-based map and the original OctoMap implementation [1] regarding the computation time for map updates. Both implementations are able to fuse noisy sensor measurements using ray casting. However, our method with early termination (Sect. II-E) is significantly more efficient in terms of computation.

The experiment data is collected by a Velodyne VLP-16 3-D LiDAR at 20 Hz. Each laser scan contains about $12,000$ rays with a range of 100m. As seen in Fig. 5(b), the environment consists of two sections with different characteristics (Fig. 5(a)): 1) an open space on grassland, 2) a narrow corridor between two buildings. As in Fig. 5(c), due to the use of map sparsity, our method shows a significant decrease on computation time, especially in the open space. We note that our map is less sensitive to the distance to obstacles, and thus is preferable for robotic systems operating in complex environments.

B. Autonomous Navigation in Outdoor Environments

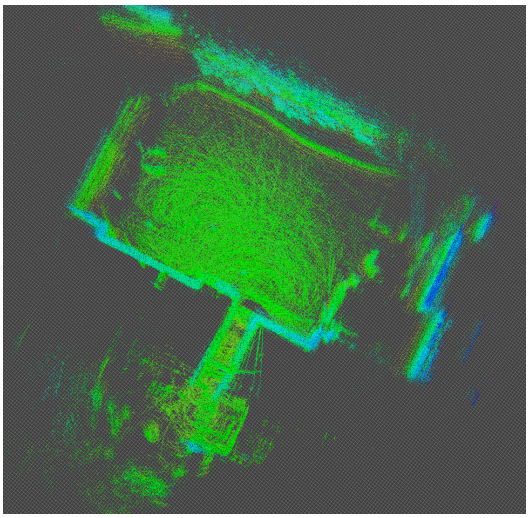
Our mapping system is further validated in a 3D autonomous outdoor flight experiment (Fig. 6). All software run onboard our experimental quadrotor platform which is equipped with a Velodyne-3D VLP-16 LiDAR, a downward-facing sonar and a Intel NUC mini-computer (Intel Core i54250U, 16GRAM). State estimation is obtained by onboard fusion IMU and laser-based odometry [2]. The maximum flight velocity and acceleration are 1m/s and 1m/s^2.

Using the proposed map structure, the flight corridor is generated for planning a 52.414m trajectory (Fig. 6(a)). It takes only 0.0249 and 0.0328 seconds (Fig. 7(a)) for the flight corridor initialization with $6,124$ range query operations (Algo. 1) and the corridor inflation with $10,235$ range query operations. Even in this enclosed outdoor environment with complex open space, ray tracing in our map demonstrate a superior efficiency compared with OctoMap (Fig. 7(b)). As shown in Fig. 6, our system operates in full 3D, and enables the quadrotor smoothly flying through cluttered environments. We highlight both vertical and horizontal movements to avoid tree branches, bushes, and walls.

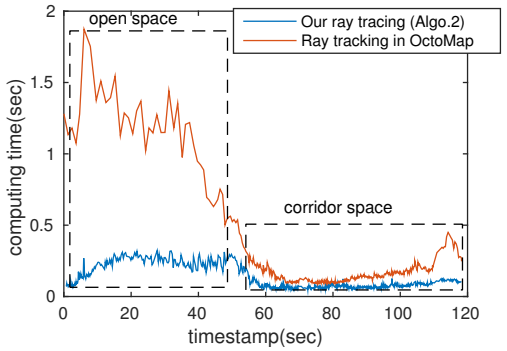
V. Conclusion

In this work, we present an improved octree-based mapping for autonomous robotic navigation. We equipped our map with two efficient operations: ray tracing-based map updates and volumetric occupancy inquiry, utilizing the sparsity nature of the environment. We presented both theoretical and experimental studies to verify the efficiency of our approach comparing to the state-of-the-art OctoMap implementation [1]. The proposed map is integrated into our trajectory generator onboard a quadrotor platform with onboard estimation and control to accomplish an autonomous flight in an unknown 3D environment.

(a) An overview of the environment with open space on grassland and corridor space between two buildings.



(b) Snapshot of the final map. Different colors represent different altitudes.



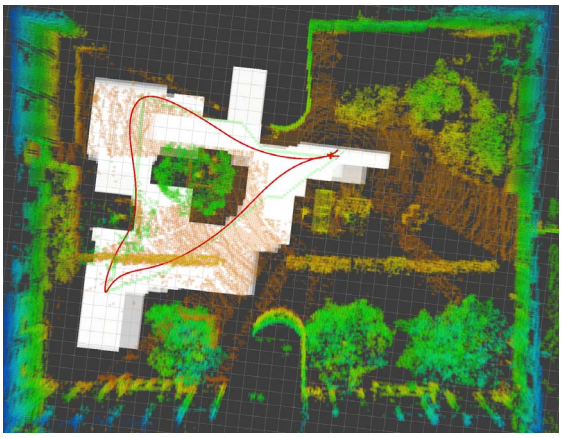
(c) The computing time for inserting 3D scans using ray tracking. Each scan contains more than $10,000$ rays at the max range of 100m. It is demonstrated that our map outperforms OctoMap [1], and especially for long-range scans in the open space.

Fig. 5. Using the 3D scan data collected by a Velodyne LiDAR (Sect. IV-A), we compare our method with the state-of-the-art OctoMap [1].

References

[1] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Auton. Robots*, 2013. [Online]. Available: http://octomap.github.com

[2] J. Zhang and S. Singh, "LOAM: Lidar odometry and mapping in real-time," in *Proc. of Robot.: Sci. and Syst.*, Berkely, CA, USA, July 2014.

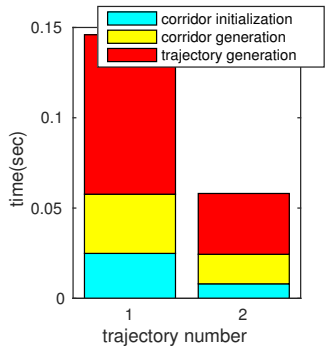


(a) Illustration of the occupancy map and the generated trajectory. Cubes with different color are obstacles at different heights. Green grids is the initial corridor. White patches on the ground illustrate the final flight corridor inflated by range query (Algo. 1) The generated trajectory is shown in red.

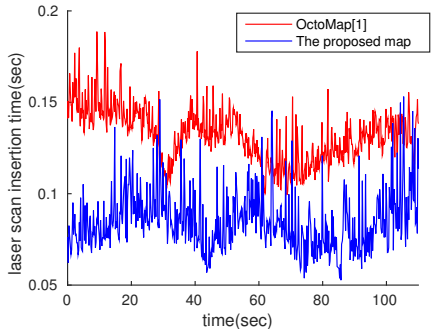
(b) (c)

(d) (e)

Fig. 6. Autonomous flight in a cluttered, unknown outdoor environment (Sect. IV-B). Our ray tracing (Algo. 2) allows the onboard computer update the map with more than 10,000 rays per scan in less than 20ms. The generation of the 52.414m-long trajectory takes less than 60ms to execute more than 16,000 range query operations (Algo. 1).



(a) Computing time for online trajectory generation. The second generation happended due to newly detected obstacle.



(b) The computation times for insertion of laser scans.

Fig. 7. Illustration of onboard calculation in the outdoor flight experiment (Sect. IV-B).

[3] J. Chen, K. Su, and S. Shen, "Real-time safe trajectory generation for quadrotor flight in cluttered environments," in *Proc. of the IEEE Intl. Conf. on Robot. and Biom.*, Zhuhai, China, Aug. 2015, URL http://www.ece.ust.hk/~eeshaojie/robio2015jing.pdf.

[4] J. Chen, T. Liu, and S. Shen, "Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments," in *Proc. of the IEEE Intl. Conf. on Robot. and Autom.*, Stockholm, Sweden, May 2016.

[5] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics*, vol. 27, no. 126, pp. 111–127, 2008.

[6] J. Elseberg, D. Borrmann, and A. Nuchter, "Efficient processing of large 3d point clouds," in *Intl. Sym. on Info., Comm. and Auton. Tech.*, Sarajevo, Oct. 2011, pp. 1–7.

[7] J. Zhang and S. Singh, "Visual-lidar odometry and mapping: Low-drift, robust, and fast," in *Proc. of the IEEE Intl. Conf. on Robot. and Autom.*, Seattle, Washington, May 2015.

[8] M. Teschner, B. Heidelberger, M. M. D. Pomerantes, and M. H. Gross, "Optimized spatial hashing for collision detection of deformable objects," in *Proc. of the Vision, Modeling, and Visualization Conf.*, Munich, Germany, Nov. 2003.

[9] M. Eitz and G. Lixu, "Hierarchical spatial hashing for real-time collision detection," in *IEEE Intl. Conf. on Shape Modeling and Applications*, Lyon, France, July 2007.

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

[11] D. Meyer-Delius, M. Beinhofer, and W. Burgard, "Occupancy grid models for robot mapping in changing environments," in *AAAI*, Toronto, ON, CA, July 2012.

[12] S. Thrun, "Learning occupancy grid maps with forward sensor models," *Auton. Robots*, vol. 15, pp. 111–127, 2003.

[13] M. Hebert, C. Caillias, and E. Krothov, "Terrain mapping for a roving planetary explorer," in *Proc. of the IEEE Intl. Conf. on Robot. and Autom.*, Scottsdale, AZ, USA, May 1989.

[14] R. Triebel, P. Pfaff, and W. Burgard, "Multi-level surface maps for outdoor terrain mapping and loop closing," in *Proc. of the IEEE/RSJ Intl. Conf. on Intell. Robots and Syst.*, Beijing, China, Oct. 2006.

[15] A. Nolan, D. Serrano, A. H. Sabate, D. P. Mussarra1and, and A. M. L. Pena, "Obstacle mapping module for quadrotors on outdoor search and rescue operations," in *International Micro Air Vehicle Conference and Flight Competition*, Toulouse, France, 2013.

[16] J. D. Hernndez, K. Isteni, N. Gracias, N. Palomeras, R. Campos, E. Vidal, R. Garca, and M. Carreras, "Autonomous underwater navigation and optical mapping in unknown natural environments," *Sensors*, vol. 16, 2016.

[17] S. Kohlbrecher, A. Romay, A. Stumpf, A. Gupta, O. Stryk, F. Bacim, D. A. Bowman, A. Goins, R. Balasubramanian, and D. C. Conner, "Human-robot teaming for rescue missions: Team vigirs approach to the 2013 darpa robotics challenge trials," *J. Field Robot.*, vol. 32, pp. 352–377, 2015.

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

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. MIT Press and McGrawHill, 2009.

[20] L. Samuli and K. Tero, "Efficient sparse voxel octrees," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 17, no. 8, pp. 1048–1059, 2011.

[21] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *Proc. of the IEEE Intl. Conf. on Robot. and Autom.*, Shanghai, China, May 2011, pp. 2520–2525.