IEEE Robotics and Automation Letters (RAL) paper presented at the
2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

Rectangular Pyramid Partitioning using Integrated Depth Sensors (RAPPIDS): A Fast Planner for Multicopter Navigation

Nathan Bucki, Junseok Lee, and Mark W. Mueller

Abstract— **We present RAPPIDS: a novel collision checking and planning algorithm for multicopters that is capable of quickly finding local collision-free trajectories given a single depth image from an onboard camera. The primary contribution of this work is a new pyramid-based spatial partitioning method that enables rapid collision detection between candidate trajectories and the environment. By leveraging the efficiency of our collision checking method, we shown how a local planning algorithm can be run at high rates on computationally constrained hardware, evaluating thousands of candidate trajectories in milliseconds. The performance of the algorithm is compared to existing collision checking methods in simulation, showing our method to be capable of evaluating orders of magnitude more trajectories per second. Experimental results are presented showing a quadcopter quickly navigating a previously unseen cluttered environment by running the algorithm on an ODROID-XU4 at 30 Hz.**

I. Introduction

The ability to perform high-speed flight in cluttered, unknown environments can enable a number of useful tasks, such as the navigation of a vehicle through previously unseen areas and rapid mapping of new environments. Many existing planning algorithms for navigation in unknown environments have been developed for multicopters, and can generally be classified as map-based algorithms, memoryless algorithms, or a mixture of the two.

Map-based algorithms first fuse sensor data into a map of the surrounding environment, and then perform path planning and collision checking using the stored map. For example, [1] uses a local map to solve a nonconvex optimization problem that returns a smooth trajectory which remains far from obstacles. Similarly, [2]–[5] each find a series of convex regions in the free-space of a dynamically updated map, and then use optimization-based methods to find a series of trajectories through the convex regions. Although these methods are generally able to avoid getting stuck in local minima (e.g. a dead end at the end of a hallway), they generally require long computation times to fuse recent sensor data into the global map.

Memoryless algorithms, however, only use the latest sensor measurements for planning. For example, [6] and [7] both use depth images to perform local planning by organizing the most recently received depth data into a k-d tree, which enables the distance from a given point to the nearest obstacle to be quickly computed. The k-d tree is then used to perform collision checking on a number of candidate trajectories, at which point the optimal collision-free candidate trajectory is chosen to track. A different memoryless algorithm is presented in [8] which inflates obstacles in the depth image based on the size of the vehicle, allowing for trajectories to be evaluated directly in image space. In [9], a significant portion of computation is performed offline in order to speed up online collision checking. The space around the vehicle is first split into a grid, a finite set of candidate trajectories are generated, and the grid cells with which each trajectory collides are then computed. Thus, when flying the vehicle, if an obstacle is detected in a given grid cell, the corresponding trajectories can be quickly determined to be in collision. However, such offline methods have the disadvantage of constraining the vehicle to a less expressive set of possible candidate trajectories, e.g. forcing the vehicle to only travel at a single speed.

Several algorithms also leverage previously gathered data while handling the latest sensor measurements separately, allowing for more collision-free trajectories to be found than when using memoryless methods while maintaining a fast planning rate. For example, in [10] a stereo camera pair is used onboard a fixed-wing UAV to detect obstacles at a specific distance in front of the vehicle, allowing for a local map of obstacles to be generated as the vehicle moves forward. In [11] a number of recent depth images are used to find the minimum-uncertainty view of a queried point in space, essentially giving the vehicle a wider field of view for planning. Additionally, in [12] the most recent depth image is both organized into a k-d tree and fused into a local map, allowing for rapid local planning in conjunction with slower global planning.

Although the previously discussed planning algorithms have been shown to perform well in complex environments, they typically require the use of an onboard computer with processing power roughly equivalent to a modern laptop. This requirement can significantly increase the cost, weight, and power consumption of a vehicle compared to one with more limited computational resources. We address this problem by introducing a novel spatial partitioning and collision checking method to find collision-free trajectories through the environment at low computational cost, enabling rapid path planning on vehicles with significantly lower computational resources than previously developed systems.

The proposed planning algorithm, classified as a memoryless algorithm, takes the latest vehicle state estimate and a single depth image from an onboard camera as input. The depth image is used to generate a number of rectangular pyramids that approximate the free space in the environment.

The authors are with the High Performance Robotics Lab, University of California, Berkeley, CA, USA. {nathan_bucki, junseok_lee, mwm}@berkeley.edu

As described in later sections, the use of pyramids in conjunction with a continuous-time representation of the vehicle trajectory allows for any given trajectory to be efficiently labeled as either remaining in the free space, i.e. inside the generated pyramids, or as being potentially in collision with the environment. Thus, a large number of candidate trajectories can be quickly generated and checked for collisions, allowing for the lowest cost trajectory, as specified by a user provided cost function, to be chosen for tracking until the next depth image is captured. Furthermore, by choosing a continuous-time representation of the candidate trajectories, each trajectory can be quickly checked for input feasibility using existing methods.

The use of pyramids to approximate the free space is advantageous because they can be created efficiently by exploiting the structure of a depth image, they can be generated on an as-needed basis (avoiding the up-front computation cost associated with other spatial partitioning data structures such as k-d trees), and because they inherently prevent occluded/unknown space from being marked as free space. Additionally, because our method is a memoryless method rather than a map-based method, it is robust to common mapping errors resulting from poor state estimation (e.g. incorrect loop closures).

II. System Model and Relevant Properties

In this section we describe the form of the trajectories used for planning and several of their relevant properties. These properties are later exploited to perform efficient collision checking between the trajectories and the environment.

We assume the vehicle is equipped with sensors capable of producing depth images that can be modeled using the standard pinhole camera model with focal length f. Let the depth-camera-fixed frame C be located at the focal point of the image with z-axis $\boldsymbol{z}_C$ perpendicular to the image plane. The point at position (X, Y, Z) written in the depth-camera-fixed frame is then located $x = f\frac{X}{Z}$ pixels horizontally and $y = f\frac{Y}{Z}$ pixels vertically from the image center with depth value Z.

A. Trajectory and Collision Model

We follow [13] and [14] in modeling the desired position trajectory of the center of mass of the vehicle as the minimum jerk trajectory between two states, which corresponds to a fifth order polynomial in time. Let $\boldsymbol{s}(t)$, $\dot{\boldsymbol{s}}(t)$, and $\ddot{\boldsymbol{s}}(t) \in \mathbb{R}^3$ denote the position, velocity, and acceleration of the center of mass of the vehicle relative to a fixed point in an inertial frame. The desired position trajectory is then defined by the initial state of the vehicle, defined by $\boldsymbol{s}(0)$, $\dot{\boldsymbol{s}}(0)$, and $\ddot{\boldsymbol{s}}(0)$, the duration of the trajectory T, and the desired state of the vehicle at the end of the trajectory, defined by $\boldsymbol{s}(T)$, $\dot{\boldsymbol{s}}(T)$, and $\ddot{\boldsymbol{s}}(T)$. A desired position trajectory of the vehicle can then be written as

$$\boldsymbol{s}(t) = \frac{\boldsymbol{\alpha}}{120}t^5 + \frac{\boldsymbol{\beta}}{24}t^4 + \frac{\boldsymbol{\gamma}}{6}t^3 + \frac{\ddot{\boldsymbol{s}}(0)}{2}t^2 + \dot{\boldsymbol{s}}(0)t + \boldsymbol{s}(0) \quad (1)$$

where $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\gamma} \in \mathbb{R}^3$ are constants that depend only on $\boldsymbol{s}(T)$, $\dot{\boldsymbol{s}}(T)$, $\ddot{\boldsymbol{s}}(T)$, and T. Note that the thrust direction of the vehicle (and thus its roll and pitch) is defined by its acceleration $\ddot{\boldsymbol{s}}(t)$. We refer the reader to [13] for details regarding this relation, as well as methods for quickly checking whether constraints on the minimum and maximum thrust and magnitude of the angular velocity of the multicopter are satisfied throughout the duration of the trajectory. We define $\boldsymbol{s}(t)$ as a collision-free trajectory if a sphere $\mathcal{S}$ centered at $\boldsymbol{s}(t)$ that contains the vehicle does not intersect with any obstacles for the duration of the trajectory.

We additionally define a similar trajectory $\boldsymbol{s}_c(t)$ with initial position $\boldsymbol{s}_c(0)$ coincident with the depth-camera-fixed frame C such that

$$\boldsymbol{s}_c(t) = \frac{\boldsymbol{\alpha}}{120}t^5 + \frac{\boldsymbol{\beta}}{24}t^4 + \frac{\boldsymbol{\gamma}}{6}t^3 + \frac{\ddot{\boldsymbol{s}}(0)}{2}t^2 + \dot{\boldsymbol{s}}(0)t + \boldsymbol{s}_c(0) \quad (2)$$

The trajectory $\boldsymbol{s}_c(t)$ is used for collision checking rather than directly using the trajectory of the center of mass $\boldsymbol{s}(t)$ because $\boldsymbol{s}_c(t)$ originates at the focal point of the depth image, allowing for the use of the advantageous properties of $\boldsymbol{s}_c(t)$ described in the following subsections. Let $\mathcal{S}_C$ be a sphere centered at $\boldsymbol{s}_c(t)$ with radius r that contains the sphere $\mathcal{S}$. If the larger sphere $\mathcal{S}_C$ does not intersect with any obstacles for the duration of the trajectory, we can then also verify that the sphere containing the vehicle $\mathcal{S}$ remains collision-free. Thus, we can use $\boldsymbol{s}_c(t)$ and its advantageous properties to check if $\boldsymbol{s}(t)$ is collision-free at the expense of a small amount of conservativeness related to the difference in size between the outer sphere $\mathcal{S}_C$ and inner sphere $\mathcal{S}$.

B. Trajectory sections with monotonically changing depth

We split a given trajectory, e.g. $\boldsymbol{s}_c(t)$, into different sections with monotonically increasing or decreasing distance along the z-axis $\boldsymbol{z}_C$ of the depth-camera-fixed frame C (i.e. into the depth image) as follows. First, we compute the rate of change of $\boldsymbol{s}_c(t)$ along $\boldsymbol{z}_C$ as $\dot{d}_z(t) = \boldsymbol{z}_C \cdot \dot{\boldsymbol{s}}_c(t)$. Then, by solving $\dot{d}_z(t) = 0$ for $t \in [0, T]$, we can find points $\mathcal{T}_\mathrm{z}$ at which $\boldsymbol{s}_c(t)$ might change direction along $\boldsymbol{z}_C$, defined as

$$\mathcal{T}_\mathrm{z} = \{t_i : t_i \in [0, T],\ \dot{d}_z(t_i) = 0\} \cup \{0, T\} \quad (3)$$

Note $\mathcal{T}_\mathrm{z}$ can be computed in closed-form because it only requires finding the roots of the fourth order polynomial $\dot{d}_z(t) = 0$.

Splitting the trajectory into these monotonic sections is advantageous for collision checking because we can compute the point of each monotonic section with the deepest depth from the camera by simply evaluating the endpoints of the section. Thus, we can avoid performing collision checking with any obstacles at a deeper depth than the deepest point of the trajectory.

C. Trajectory-Plane Intersections

A similar method can be used to quickly determine if and/or when a given trajectory intersects with an arbitrary plane defined by a point $\boldsymbol{p} \in \mathbb{R}^3$ and unit normal $\boldsymbol{n} \in \mathbb{R}^3$. Let the distance of the trajectory from the plane be written as $d(t) = \boldsymbol{n} \cdot (\boldsymbol{s}_c(t) - \boldsymbol{p})$. The set of times $\mathcal{T}_\mathrm{cross}$ at which $\boldsymbol{s}_c(t)$ intersects the given plane are then defined as

$$\mathcal{T}_\mathrm{cross} = \{t_i : t_i \in [0, T],\ d(t_i) = 0\} \quad (4)$$

requiring the solution of the equation $d(t) = 0$. Unfortunately, $d(t)$ is in general a fifth order polynomial, meaning that its roots cannot be computed in closed-form and require more computationally expensive methods to find.

To this end, we extend [13] and [14] in presenting the conditions under which finding $\mathcal{T}_{\text{cross}}$ can be reduced to finding the roots of a fourth order polynomial. Specifically, if a single crossing time of $d(t)$ is known a priori, $d(t) = 0$ can be solved by factoring out the known root and solving the remaining fourth order polynomial. This property is satisfied, for example, by any plane with $\boldsymbol{p} := \boldsymbol{s}_c(0)$ (i.e. a plane that intersects the initial position of the trajectory), resulting in the following equation for $d(t)$:

$$d(t) = \boldsymbol{n} \cdot \left(\frac{\boldsymbol{\alpha}}{120} t^4 + \frac{\boldsymbol{\beta}}{24} t^3 + \frac{\boldsymbol{\gamma}}{6} t^2 + \frac{\ddot{\boldsymbol{s}}(0)}{2} t + \dot{\boldsymbol{s}}(0) \right) t \quad (5)$$

Thus, the remaining four unknown roots of (5) can be computed using the closed-form equation for the roots of a fourth order polynomial, allowing for $\mathcal{T}_{\text{cross}}$ to be computed extremely quickly. As described in the following section, we exploit this property in order to quickly determine when a given trajectory leaves a given pyramid.

III. Algorithm Description

In this section we describe the our novel pyramid-based spatial partitioning method, its use in performing efficient collision checking, and the algorithm used to search for the best collision-free trajectory.[1]

A. Pyramid generation

For each depth image generated by the vehicle's depth sensor, we partition the free space of the environment using a number of rectangular pyramids, where the apex of each pyramid is located at the origin of the depth camera-fixed frame C (i.e. at $\boldsymbol{s}_c(0)$), and the base of each pyramid is located at some depth Z and is perpendicular to the z-axis of the depth camera-fixed frame $\boldsymbol{z}_C$ as shown in Figure 1.

The depth value stored in each pixel of the image is used to define the separation of free space $\mathcal{F}$ and occupied space $\mathcal{O}$. We additionally treat the space $\mathcal{U}$ located outside the field of view of the camera at depth l from the camera focal point as occupied space. Pyramid $\mathcal{P}$ is defined such that while trajectory $\boldsymbol{s}_c(t)$ remains inside $\mathcal{P}$, the sphere containing the vehicle $\mathcal{S}_C$ will not intersect with any occupied space, meaning that the segment of $\boldsymbol{s}_c(t)$ inside $\mathcal{P}$ can be considered collision-free.

Note that if $\boldsymbol{s}_c(t)$ remains inside the pyramid, we can not only guarantee that the vehicle will not collide with any obstacles detected by the depth camera, but that the vehicle will not collide with any occluded obstacles either. This differs from other methods that treat each pixel in the depth image as an individual obstacle to be avoided, which can result in the generation of over-optimistic trajectories that may collide with unseen obstacles. Furthermore, our method straightforwardly incorporates field of view constraints, allowing for the avoidance of nearly all unseen obstacles in addition to those detected by the depth camera.

[1]An implementation of the algorithm can be found at `https://github.com/nlbucki/RAPPIDS`

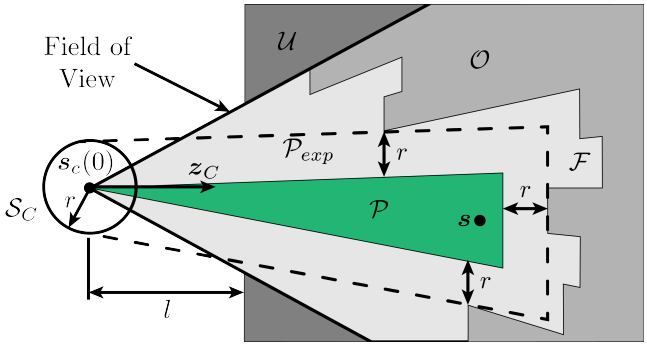

Fig. 1. 2D side view illustrating the generation of a single pyramid $\mathcal{P}$, shown in green, from a single depth image and given point $\boldsymbol{s}$. The depth values of each pixel are used to define the division between free space $\mathcal{F}$ and occupied space $\mathcal{O}$. Because the depth camera has a limited field of view, we additionally consider any space outside the field of view farther than distance l from the camera to be unknown space $\mathcal{U}$, which is treated the same as occupied space. The expanded pyramid $\mathcal{P}_{exp}$ (dash-dotted line) is first generated such that it does not contain any portion of $\mathcal{O}$ or $\mathcal{U}$, and then used to define pyramid $\mathcal{P}$ such that it is distance r from any obstacles.

The function InflatePyramid is used to generate a pyramid $\mathcal{P}$ by taking an initial point $\boldsymbol{s}$ as input and returning either a pyramid containing $\boldsymbol{s}$ or a failure indicator. In this work we choose $\boldsymbol{s}$ to be the endpoint of a given trajectory that we wish to check for collisions, and only generate a new pyramid if $\boldsymbol{s}$ is not already contained in an existing pyramid (details are provided in the following subsection). We start by projecting $\boldsymbol{s}$ into the depth image and finding the nearest pixel p. Then, pixels of the image are read in a spiral about pixel p in order to compute the largest possible expanded rectangle $\mathcal{P}_{exp}$ that does not contain any occupied space. Finally, pyramid $\mathcal{P}$ is computed by shrinking the expanded pyramid $\mathcal{P}_{exp}$ such that each face of $\mathcal{P}$ is not within vehicle radius r of occupied space. Further details regarding our implementation of InflatePyramid can be found online.[1]

This method additionally allows for pyramid generation failure indicators to be returned extremely quickly. For example, consider the case where the initial point $\boldsymbol{s}$ exists inside occupied space $\mathcal{O}$. Then, only the depth value of the nearest pixel p must be read before finding that no pyramid containing $\boldsymbol{s}$ can be generated, requiring only a single pixel of the depth image to be processed. This property greatly reduces the number of operations required to determine when a given point is in collision with the environment.

B. Collision checking using pyramids

Algorithm 1 describes how the set of all previously generated pyramids $\mathcal{G}$ is used to determine whether a given trajectory $\boldsymbol{s}_c(t)$ will collide with the environment. A trajectory is considered collision-free if it remains inside the space covered by $\mathcal{G}$ for the full duration of the trajectory. An example of Algorithm 1 is shown in Figure 2.

We first split the trajectory $\boldsymbol{s}_c(t)$ into sections with

Algorithm 1 Single Trajectory Collision Checking

1: **input:** Trajectory $\boldsymbol{s}_c(t)$ to be checked for collisions, set of all previously found pyramids $\mathcal{G}$, depth image $\mathcal{D}$
2: **output:** Boolean indicating if trajectory is collision-free, updated set of pyramids $\mathcal{G}$
3: **function** IsCollisionFree($\boldsymbol{s}_c(t)$, $\mathcal{G}$, $\mathcal{D}$)
4: $\quad\mathcal{M} \leftarrow$ GetMonotonicSections($\boldsymbol{s}_c(t)$)
5: $\quad$**while** $\mathcal{M}$ is not empty **do**
6: $\quad\quad\bar{\boldsymbol{s}}_c(t) \leftarrow$ pop($\mathcal{M}$)
7: $\quad\quad\bar{\boldsymbol{s}} \leftarrow$ DeepestPoint($\bar{\boldsymbol{s}}_c(t)$)
8: $\quad\quad\mathcal{P} \leftarrow$ FindContainingPyramid($\mathcal{G}, \bar{\boldsymbol{s}}$)
9: $\quad\quad$**if** $\mathcal{P}$ is null **then**
10: $\quad\quad\quad\mathcal{P} \leftarrow$ InflatePyramid($\bar{\boldsymbol{s}}, \mathcal{D}$)
11: $\quad\quad\quad$**if** $\mathcal{P}$ is null **then**
12: $\quad\quad\quad\quad$**return** false
13: $\quad\quad\quad$push($\mathcal{P}$) $\rightarrow \mathcal{G}$
14: $\quad\quad t^{\downarrow} \leftarrow$ FindDeepestCollisionTime($\mathcal{P}, \bar{\boldsymbol{s}}_c(t)$)
15: $\quad\quad$**if** $t^{\downarrow}$ is not null **then**
16: $\quad\quad\quad$push(GetSubsection($\bar{\boldsymbol{s}}_c(t), t^{\downarrow}$)) $\rightarrow \mathcal{M}$
17: $\quad$**return** true

monotonically changing depth as described in Section II-B, and insert the sections into list $\mathcal{M}$ using GetMonotonicSections (line 4). Then, for each monotonic section $\bar{\boldsymbol{s}}_c(t)$, we compute the deepest point $\bar{\boldsymbol{s}}$ (i.e. one of the endpoints of the section), and try to find a pyramid containing that point (line 6-8). The function FindContainingPyramid (line 8) returns either a pyramid that contains $\bar{\boldsymbol{s}}$ or null, indicating no pyramid containing $\bar{\boldsymbol{s}}$ was found. If no pyramid in $\mathcal{G}$ contains $\bar{\boldsymbol{s}}$, we attempt to generate a new pyramid using the method described in the previous subsection (line 10), but if pyramid generation fails we declare the trajectory to be in collision (line 12).

Next, we try to compute the deepest point at which the monotonic section $\bar{\boldsymbol{s}}_c(t)$ intersects one of the four lateral faces of the pyramid $\mathcal{P}$. Using the method described in Section II-C, we compute the times at which $\bar{\boldsymbol{s}}_c(t)$ intersects each lateral face of the pyramid, and choose the time $t^{\downarrow}$ at which $\bar{\boldsymbol{s}}_c(t)$ has the greatest depth (line 14). If $\bar{\boldsymbol{s}}_c(t)$ is found to not collide with any of the lateral faces of the pyramid, then it necessarily must remain inside the pyramid and the section can be declared collision-free. However, if $\bar{\boldsymbol{s}}_c(t)$ does collide with one of the lateral faces of the pyramid, we split it at $t^{\downarrow}$ and add the section of $\bar{\boldsymbol{s}}_c(t)$ that is outside of the pyramid to $\mathcal{M}$ (line 16). Thus, if each subsection of the trajectory is found to be inside the space covered by the set of pyramids $\mathcal{G}$, then the trajectory is declared collision-free (line 17).

Note that this method of collision checking allows for pyramids to be generated on an as-needed basis rather than requiring all pyramids to be generated in a batch process when a new depth image arrives. This additionally avoids generating unneeded pyramids; only those required for collision checking are created.

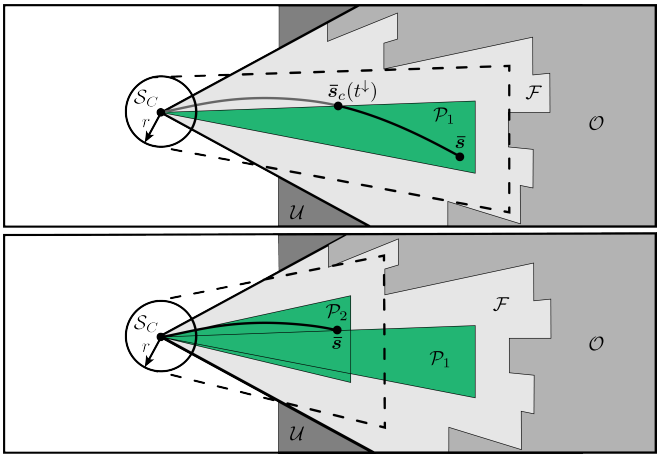

Fig. 2. 2D example of the collision checking method described by Algorithm 1 as used to check a given trajectory for collisions. The trajectory is first split into sections with monotonically changing depth, which are stored in list $\mathcal{M}$. Top: A single trajectory section $\bar{\boldsymbol{s}}_c(t)$ is chosen from list $\mathcal{M}$. The deepest point of the trajectorysection $\bar{\boldsymbol{s}}$ is computed and pyramid $\mathcal{P}_1$ containing $\bar{\boldsymbol{s}}$ is generated. The trajectory $\bar{\boldsymbol{s}}_c(t)$ is then subdivided into a section that remains inside the pyramid (black) and a section that leaves the pyramid (gray). Bottom: The trajectory section that leaves $\mathcal{P}_1$ is checked for collisions in the same manner. Pyramid $\mathcal{P}_2$ is generated using the deepest point of the trajectory section, and then used to verify that the trajectory section does not collide with the environment.

C. Planning algorithm

Algorithm 2 describes the path planning algorithm used in this work. The algorithm takes as input the most recently received depth image and vehicle state estimate, where the state estimate partially defines each candidate trajectory as given in (2). Within a user-specified time budget, the algorithm repeatedly generates and evaluates candidate trajectories for cost and the satisfaction of constraints, returning the lowest cost trajectory that satisfies all constraints. We choose to use a random search algorithm due to its simplicity and probabilistic optimality, though the collision checking algorithm presented in the previous subsection can be used in conjunction with other planning algorithms as well (see [15], for example).

Algorithm 2 Lowest Cost Trajectory Search

1: **input:** Latest depth image $\mathcal{D}$ and vehicle state
2: **output:** Lowest cost collision-free trajectory found $\boldsymbol{s}_c^*(t)$ or an undefined trajectory (indicating failure)
3: **function** FindLowestCostTrajectory()
4: $\quad\boldsymbol{s}_c^*(t) \leftarrow$ undefined with Cost($\boldsymbol{s}_c^*(t)$) $= \infty$
5: $\quad\mathcal{G} \leftarrow \emptyset$
6: $\quad$**while** computation time not exceeded **do**
7: $\quad\quad\boldsymbol{s}_c(t) \leftarrow$ GetNextCandidateTraj()
8: $\quad\quad$**if** Cost($\boldsymbol{s}_c(t)$) $<$ Cost($\boldsymbol{s}_c^*(t)$) **then**
9: $\quad\quad\quad$**if** IsDynamicallyFeas($\boldsymbol{s}_c(t)$) **then**
10: $\quad\quad\quad\quad$**if** IsCollisionFree($\boldsymbol{s}_c(t)$, $\mathcal{G}$, $\mathcal{D}$) **then**
11: $\quad\quad\quad\quad\quad\boldsymbol{s}_c^*(t) \leftarrow \boldsymbol{s}_c(t)$
12: $\quad$**return** $\boldsymbol{s}_c^*(t)$

Let GETNEXTCANDIDATETRAJ be defined as a function that returns a randomly generated candidate trajectory $\boldsymbol{s}_c(t)$ using the methods described in [13] (line 7). The function COST is a user-specified function used to compare candidate trajectories (line 8). In this work, we define COST to be the following, where $\boldsymbol{d}$ is a desired exploration direction:

$$\text{COST}(\boldsymbol{s}_c(t)) = \frac{\boldsymbol{d} \cdot (\boldsymbol{s}_c(0) - \boldsymbol{s}_c(T))}{T} \tag{6}$$

That is, better trajectories are those that cause the vehicle to move quickly in the desired direction $\boldsymbol{d}$. Note, however, that COST can be defined arbitrarily by the user to include other objectives based on the desired behavior of the vehicle (e.g. to favor increased distance to other vehicles or people).

The function ISDYNAMICALLYFEAS (line 9) checks whether the given candidate trajectory satisfies constraints on the total thrust and angular velocity of the vehicle using methods described in [13]. Finally, the candidate trajectory is checked for collisions with the environment using ISCOLLISIONFREE (line 10). We check for collisions with the environment last because it is the most computationally demanding step of the process.

In this way, Algorithm 2 can be used as a high-rate local planner that ensures the vehicle avoids obstacles, while a global planner that may require significantly more computation time can be used to specify high-level goals (e.g. the exploration direction $\boldsymbol{d}$) without the need to worry about obstacle avoidance or respecting the dynamics of the vehicle. We run Algorithm 2 in a receding-horizon fashion, where each new depth image is used to compute a new collision-free trajectory. We additionally constrain our candidate trajectories to bring the vehicle to rest, so that if no feasible trajectories are found during a given planning step, the last feasible trajectory can be tracked until the vehicle comes to rest.

IV. ALGORITHM PERFORMANCE

In this section we assess the performance of the proposed algorithm in terms of its conservativeness in labeling trajectories as collision-free, its speed, and its ability to evaluate a dense set of candidate trajectories in various amounts of compute time. We additionally compare our method to other state-of-the-art memoryless planning algorithms.

To benchmark our collision checking method, we conduct various Monte Carlo simulations using a series of randomly generated synthetic depth images and vehicle states. Examples of several generated depth images are shown in Figure 3. The image is generated by placing two $20\,\text{cm}$ thick rectangles with random orientations in front of the camera at distances sampled uniformly at random on $(1.5\,\text{m}, 3\,\text{m})$. Note that this choice of obstacles is arbitrary; any number, distribution, or type of obstacles could be used to conduct such tests. However, rather than trying to emulate a specific type of environment, we choose to use obstacles in our benchmark that are primarily easy to both visualize and reason about conceptually. Furthermore, the use of such obstacles does not unfairly benefit the proposed collision checking method by, e.g., breaking the free-space into regions that may be easier to describe using pyramids.

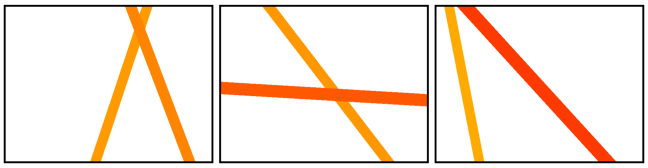

Fig. 3. Three examples of synthetic depth images used for benchmarking the proposed algorithm. Two rectangular obstacles are generated at different constant depths. The background is considered to be at infinite depth.

The initial velocity of the vehicle in the camera-fixed z-direction $\boldsymbol{z}_C$ is sampled uniformly on $(0\,\text{m}\,\text{s}^{-1}, 4\,\text{m}\,\text{s}^{-1})$, and the initial velocity of the vehicle in both the x-direction $\boldsymbol{x}_C$ and y-direction $\boldsymbol{y}_C$ is sampled uniformly on $(-1\,\text{m}\,\text{s}^{-1}, 1\,\text{m}\,\text{s}^{-1})$. We assume the camera is mounted such that $\boldsymbol{z}_C$ is perpendicular to the thrust direction of the vehicle, and thus set the initial acceleration of the vehicle in both the $\boldsymbol{x}_C$ and $\boldsymbol{z}_C$ directions to zero. The initial acceleration in the $\boldsymbol{y}_C$ direction is sampled uniformly on $(-5\,\text{m}\,\text{s}^{-2}, 5\,\text{m}\,\text{s}^{-2})$.

The planner generates candidate trajectories that come to rest at randomly sampled positions in the field of view of the depth camera. Specifically, positions in the depth image are sampled uniformly in pixel coordinates and then deprojected to a depth that is sampled uniformly on $(1.5\,\text{m}, 3\,\text{m})$. The duration of each trajectory is sampled uniformly on $(2\,\text{s}, 3\,\text{s})$.

The algorithm was implemented in C++ and compiled using GCC version 5.4.0 with the -O3 optimization setting. Three platforms were used to assess performance: a laptop with an Intel i7-8550U processor set to performance mode, a Jetson TX2, and an ODROID-XU4. The algorithm is run as a single thread in all scenarios.

A. Conservativeness

We first analyze the accuracy of the collision checking method described by Algorithm 1. A key property of our method is that it will never erroneously label a trajectory as collision-free that either collides with an obstacle or has the potential to collide with an occluded obstacle. Such a property is typically a requirement for collision checking algorithms used with aerial vehicles, as a trajectory mislabeled as collision-free can result in a catastrophic crash resulting in a total system failure.

However, because the generated pyramids cannot exactly describe the free space of the environment, our method may erroneously label some collision-free trajectories as being in-collision. In order to quantify this conservativeness, we compare our method to a ground-truth, ray-tracing based collision checking method capable of considering both field-of-view constraints and occluded obstacles. We define conservativeness as the number of trajectories erroneously labeled as in-collision divided by the total number of trajectories labeled (both correctly and incorrectly) as in-collision. A single test consists of first generating a synthetic scene and random initial state of the vehicle as previously described. We then generate 1000 random trajectories for each scene,

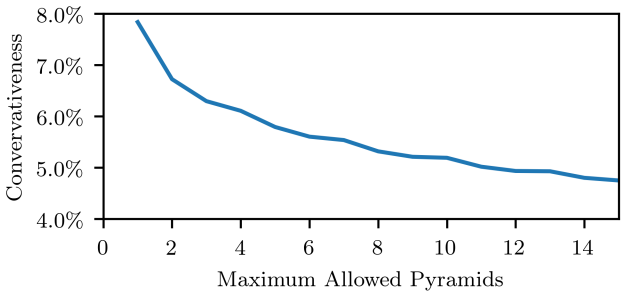

Fig. 4. Conservativeness of the collision checking algorithm as a function of the maximum number of pyramids allowed to be generated. We define conservativeness as the number of trajectories erroneously labeled as in-collision divided by the total number of trajectories labeled as in-collision. The free-space of the environment is described with increasing detail as more pyramids are allowed to be generated, leading to a lower number of trajectories being erroneously labeled as in-collision.

and perform collision checking both with our method and the ground-truth method. The number of trajectories both correctly and incorrectly labeled as in-collision are averaged over 10^4 such scenes. Additionally, in order to quantify how well the environment can be described using the pyramids generated by our method, we limit the number of pyramids the collision checking algorithm is allowed to use, and repeat this test for various pyramid limits.

Figure 4 shows how the over-conservativeness of our method decreases as the number of pyramids allowed to be used for collision checking increases. The percent of mislabeled trajectories is initially higher because the environment cannot be described with high accuracy using fewer pyramids. However, conservativeness remains nearly constant for larger pyramid limits, indicating that our method may erroneously mislabel a small number of collision-free trajectories (e.g. those in close proximity to obstacles). Note that we do not limit the number of pyramids generated when using the planning algorithm described in Algorithm 2, as we have found it to be unnecessary in practice.

B. Collision Checking Speed

Next we compare our collision checking method to the state-of-the-art k-d tree based methods described in [6] and [7]. Both our method and k-d tree methods involve two major steps: the building of data structures (i.e. a k-d tree, or the pyramids described in this paper) and the use of those data structures to perform collision checking with the environment. Our method differs from k-d tree based methods, however, in its over-conservativeness. Specifically, we consider trajectories that pass through occluded space to be in collision with the environment, while k-d tree based methods only consider trajectories that pass within the vehicle radius of detected obstacles to be in collision.

We compare our method to the k-d tree methods by first limiting the amount of time allocated for pyramid generation such that it is similar to the time required to build a k-d tree as reported in [6] and [7] (roughly $1.81\,\mathrm{ms}$). We then check 1000 trajectories for collisions, and compute the average time required to check a single trajectory for collisions using the generated pyramids. Similar to [6] and [7], we use a 160×120 resolution depth image which we generate using the previously described method, and average our results over 10^4 Monte Carlo trails.

TABLE I
AVERAGE COLLISION CHECKING TIME PER TRAJECTORY

	Computer	Single Trajectory Collision Check (µs)
Florence et al.[2][6]	i7 NUC	56
Lopez et al.[2] [7]	i7-2620M	48
RAPPIDS (ours)	i7-8550U	1.20
RAPPIDS (ours)	Jetson TX2	3.81
RAPPIDS (ours)	ODROID-XU4	8.72

Table I shows how the average performance of our method outperforms the best-case results reported by [6] and [7]. On average 27.5, 19.3, and 15.3 pyramids were generated during the allocated $1.81\,\mathrm{ms}$ pyramid generation time on i7, TX2, and ODROID platforms respectively. The difference in computation time can be reasoned about using a time complexity analysis. Let a given depth image contain n pixels. Then $O(n\log(n))$ operations are required to build a k-d tree, while $O(n)$ operations are required to generate a single pyramid (of which there are typically few). Because a single k-d tree query takes $O(\log(n))$ time, if the trajectory must be checked for collisions at m sample points along the trajectory, then the time complexity for checking a single trajectory for collisions is $O(m\log(n))$. However, collision checking a single trajectory using our method can be done in near constant time, as it only requires finding the roots of several fourth order polynomials (which is done in closed-form) as described in Section II-C. Additionally, note that while an entire k-d tree must be built before being used to check trajectories for collisions, the pyramids generated by our method can be built on an as-needed basis, reducing computation time even further.

C. Overall Planner Performance

Finally, we describe the overall performance of the planner, i.e. Algorithm 2, using the same Monte Carlo simulation but with 640×480 resolution depth images, which are the same resolution as those used in the physical experiments described in the following section. The number of trajectories evaluated by the planner is used as a metric of performance, where a larger number of generated trajectories indicates a better coverage of the trajectory search space and thus higher likelihood of finding the lowest possible cost trajectory within the allocated planning time.

Figure 5 shows the results of running the planner for 10^4 Monte Carlo trails each on the i7-8550U processor, the Jetson TX2, and the ODROID-XU4 for computation time limits between $0\,\mathrm{ms}$ and $50\,\mathrm{ms}$. Naturally, as computation time increases, the average number of trajectories evaluated increases monotonically. Furthermore, we observe that the

[2]The best reported average collision checking time required per trajectory is used for comparison.

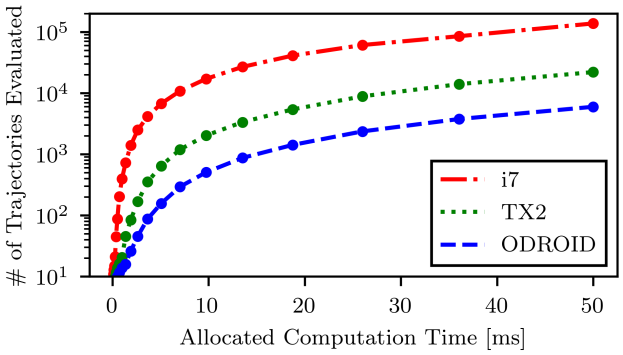

Fig. 5. Average planner performance as a function of allocated computation time across various platforms. As computation time increases, the number of trajectories evaluated increases at different rates for platforms with different amounts of computation power.

i7-8550U outperforms the Jetston TX2, which outperforms the ODROID-XU4. The difference in performance can be explained by the fact that the Jetson TX2 and especially the ODROID-XU4 are intended to be low-power devices capable of being used in embedded applications. However, due to the computational efficiency of our collision checking method, we found that even the ODROID-XU4 is capable of evaluating a sufficiently large number of trajectories within a small amount of time. This makes it feasible to use low-power devices such as the ODROID-XU4 as onboard flight controllers while still achieving fast, reactive flight.

V. Experimental Results

In this section we demonstrate the use of the proposed algorithm on an experimental quadcopter, shown in Figure 6. The quadcopter has a mass of $1.0\,\mathrm{kg}$, and is equipped with an ODROID-XU4, RealSense D435i depth camera, RealSense T265 tracking camera, and Crazyflie 2.0 flight controller. The ODROID is used in order to demonstrate the feasibility of running the proposed algorithm at high rates on cheap, low mass, and low power hardware. The tracking camera provides pose estimates to the ODROID at $200\,\mathrm{Hz}$, which a Kalman filter uses to compute translational velocity estimates. Filtered acceleration estimates are obtained at $100\,\mathrm{Hz}$ using the IMU onboard the crazyflie flight controller. The depth camera is configured to capture 640×480 resolution depth images at $30\,\mathrm{Hz}$, and the proposed planning algorithm is run for $30\,\mathrm{ms}$ when each new depth image arrives using the latest state estimate provided by the Kalman filter. If no collision-free trajectories can be found during a given planning stage, the vehicle continues to track the most recently found collision-free trajectory from a previous planning stage until either a new collision-free trajectory is found or the vehicle comes to rest.

The vehicle was commanded to fly in a U-shaped tunnel environment that was previously unseen by the vehicle, shown in Figure 7. Each branch of the tunnel measured roughly $2.5\,\mathrm{m}$ in width and height, $20\,\mathrm{m}$ in length, and was filled with various obstacles for the vehicle to avoid.

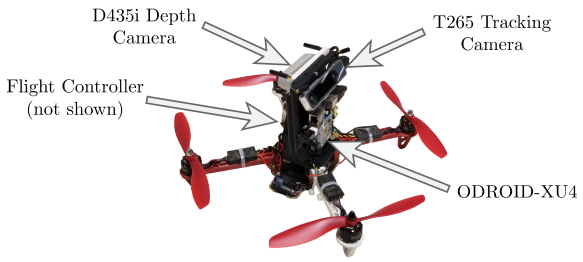

Fig. 6. Vehicle used in experiments. A RealSense D435i depth camera is used to acquire depth images, and a RealSense T265 tracking camera is used to obtain state estimates of the vehicle. The proposed algorithm is run on an ODROID-XU4, which sends desired thrust and angular velocity commands to a Crazyflie 2.0 flight controller.

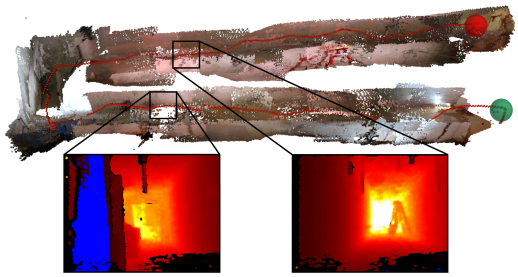

Fig. 7. Visualization of flight experiment in U-shaped tunnel environment. The path of the vehicle is shown as a red line. The vehicle starts at the green sphere and ends at the red sphere. The map of the environment (top) is generated at the end of the experiment using depth images captured by the depth camera. Two depth images (bottom) where no collision-free trajectories were found are shown to illustrate cases where the planner fails. Pixels with depth values less than $0.75\,\mathrm{m}$ but greater than the vehicle radius are highlighted in blue. In the left image, an obstacle occludes a significant portion of the image, preventing collision-free trajectories from being found due to the proximity of the obstacle to the vehicle. In the right image, a very small amount of noise is present near the bottom of the image, causing the planner to hallucinate the presence of close proximity obstacles in what would otherwise be free-space. A full video of the experiment is attached to this paper.[3]

The candidate trajectories generated by the planner were generated using the same method described in Section IV. A video of the experiment is attached to this paper.[3]

The desired exploration direction $\boldsymbol{d}$ used to compute the cost of each candidate trajectory as given by (6) is set as follows. We initialize $\boldsymbol{d}$ to be horizontal and to point down the first hallway. When the vehicle is at rest and no feasible trajectories are found by the planner, the desired exploration direction $\boldsymbol{d}$ is rotated $90°$ to the right of the vehicle, allowing the vehicle to navigate around corners. We then stop the test when the vehicle reaches the end of the second hallway. We use this method of choosing the exploration direction simply as a matter of convenience in demonstrating the use of our algorithm in a cluttered environment. However, many other suitable methods of providing high-level goals to our algorithm can be used (e.g. [16]), but are typically application dependent and thus are not discussed here.

[3]The video can also be viewed at `https://youtu.be/Pp-HIT9S6ao`

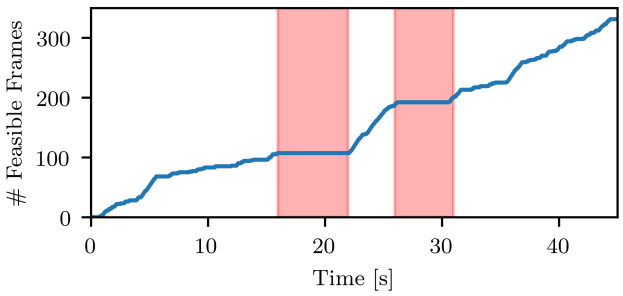

Fig. 8. Cumulative number of planning stages where at least one collision-free trajectory was found. The sections of the graph highlighted in red correspond to periods in which the vehicle is facing the wall at the end of the hallway. A large increase in successful planning stages is observed between 22 s and 26 s when the vehicle is navigating in the relatively uncluttered area between the two hallways.

During the experiment, the vehicle was able to find at least one collision-free trajectory in 35.3% of the 30 ms planning stages. Of the planning stages where at least one feasible trajectory was found, 2069.2 candidate trajectories and 2.9 pyramids were generated on average. The vehicle traveled approximately 40 m over 43 s, and attained a maximum speed of $2.66\,\mathrm{m\,s^{-1}}$.

The low percentage of planning stages where at least one collision-free trajectory was found primarily are cases where the vehicle passes closely to obstacles and also by the significant amount of noise present in the depth images. Figure 7 shows examples of both cases. Note that the amount of noise present in the depth images can be reduced via filtering, although this may lead to the potential misdetection of small and/or thin obstacles. Additionally, Figure 8 shows how the successful planning stages are distributed throughout the experiment. A lower percent of successful planning stages is observed when the vehicle is navigating the cluttered hallways than the relatively open area between the two hallways, which is potentially due to the difference in obstacle density and lighting conditions (leading to a difference in depth image noise levels).

VI. Conclusion

In this paper we presented a novel pyramid-based spatial partitioning method that allows for efficient collision checking between a given trajectory and the environment as represented by a depth image. The method allows multicopters with limited computational resources to quickly navigate cluttered environments by generating collision-free trajectories at high rates. A comparison to existing state-of-the-art depth-image-based path planning methods was performed via Monte Carlo simulation, showing our method to significantly reduce the computation time required to perform collision checking with the environment while being more conservative than other methods by implicitly considering occluded obstacles. Finally, real-world experiments were presented that demonstrate the use of our algorithm on computationally low-power hardware to perform fully autonomous flight in a previously unseen, cluttered environment.

Acknowledgement

This material is based upon work supported by the Berkeley Fellowship for Graduate Study, the Berkeley DeepDrive project 'Autonomous Aerial Robots in Dense Urban Environments', and the China High-Speed Railway Technology Co., Ltd. The experimental testbed at the HiPeRLab is the result of contributions of many people, a full list of which can be found at `hiperlab.berkeley.edu/members/`.

References

[1] H. Oleynikova, M. Burri, Z. Taylor, J. Nieto, R. Siegwart, and E. Galceran, "Continuous-time trajectory optimization for online uav replanning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 5332–5339.

[2] F. Gao, Y. Lin, and S. Shen, "Gradient-based online safe trajectory generation for quadrotor flight in complex environments," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 3681–3688.

[3] S. Liu, M. Watterson, S. Tang, and V. Kumar, "High speed navigation for quadrotors with limited onboard sensing," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 1484–1491.

[4] J. Chen, T. Liu, and S. Shen, "Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 1476–1483.

[5] J. Tordesillas, B. T. Lopez, and J. P. How, "Fastrap: Fast and safe trajectory planner for flights in unknown environments," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[6] P. Florence, J. Carter, and R. Tedrake, "Integrated perception and control at high speed: Evaluating collision avoidance maneuvers without maps," in *Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2016.

[7] B. T. Lopez and J. P. How, "Aggressive 3-d collision avoidance for high-speed navigation," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 5759–5765.

[8] L. Matthies, R. Brockers, Y. Kuwata, and S. Weiss, "Stereo vision-based obstacle avoidance for micro air vehicles using disparity space," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 3242–3249.

[9] J. Zhang, C. Hu, R. G. Chadha, and S. Singh, "Maximum likelihood path planning for fast aerial maneuvers and collision avoidance," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[10] A. J. Barry and R. Tedrake, "Pushbroom stereo for high-speed navigation in cluttered environments," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 3046–3052.

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

[12] M. Ryll, J. Ware, J. Carter, and N. Roy, "Efficient trajectory planning for high speed flight in unknown environments," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 732–738.

[13] M. W. Mueller, M. Hehn, and R. D'Andrea, "A computationally efficient motion primitive for quadrocopter trajectory generation," *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1294–1310, 2015.

[14] N. Bucki and M. W. Mueller, "Rapid collision detection for multicopter trajectories," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[15] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[16] T. Cieslewski, E. Kaufmann, and D. Scaramuzza, "Rapid exploration with multi-rotors: A frontier selection method for high speed flight," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 2135–2142.