Mapping and Motion Planning

TL, NUS

I. Introduction

A 3D reconstruction system for MAVs using GPU is implemented in our work, which gives strong support of data to the navigation and decision making in motion planning. Maintaining and updating a 3D map and planning with the global map are both computational costly. Moreover, the global map can be erroneous with noise and estimation drift. Therefore, we maintain a local map only that provides obstacles information around current robot location. Different map representations can be used for planning. Occupancy map is a typical representation. One of the most popular 3D occupancy grid (OG) maps is Octomap [1], which adopts a hierarchical octree structure to save occupancy probabilities. However, only occupancy data is not sufficient for optimization-based planners, which requires distances to obstacles. It is usually obtained by building a Euclidean signed distance field (ESDF), which is a voxel grid map where every grid carries its Euclidean distance to the nearest obstacle. ESDF is commonly constructed based on another map. The method in [2] incrementally build ESDF directly out of truncated signed distance field (TSDF), which is a surface representation [3] using projective distance (distance along the sensor ray to the measured surface), and compute these distances within a short truncation radius around the surface. However, it can only obtain approximated Euclidian distance, which may not be accurate enough for planning. This paper presents a highly efficient approach for updating ESDF with dimensionality reduction Voronoi diagram computation based on the OG map updated with voxel projection method. Moreover, the enormous power of the GPU is exploited to improve efficiency further.

To concatenate with the state estimation and mapping modules, a 3D motion planning module is designed for real-time path and trajectory generation, where a collision-free and dynamically feasible reference trajectory is generated to guide vehicles maneuver safely towards its desired target. For computational efficiency, a coarse, discretized global planner with a simplified vehicle model searches the entire workspace to produce feasible outer-loop references. The finer local planner takes the reference from the global planner and searches in a parameterized space of robot's inputs, creating a dynamically feasible and collision-free local trajectory according to a predefined cost function. Due to the environmental uncertainties, the planning process is typically repeated in a receding horizon fashion, similar to the method used in the model predictive control (MPC). This paper focuses on the local planning problem, which is solved by Motion primitives (MPs), spanning a local search tree to abstract the continuous state space [4]. We sample on the vehicles' boundary state constraints and generate the actual motion by solving a boundary value problem. The generated MPs are termed as the boundary state constrained primitives (BSCPs), which have been applied successfully to various platforms, such as autonomous cars [5, 6], fixed wings [7], and field robots [8]. In this paper, we utilize the neural network (NN) to learn the BSCPs solutions offline and approximate them with NN during online optimization. A gradient-free method using the NN and the particle swarm optimization (PSO) is designed to construct a dynamically evolving single layer tree that gradually converges to the optimal local target in the planning. Complex and long-range motions can then be generated by combining the local target with a sampling based global planner [9].

II. Mapping and Planning

An environmental OG map is generated by updating the stereo camera and 2D Rplidar data with voxel projection method. In our implementation, the depth image is transferred to the point cloud and applied with several filters, such as voxel grid downsampling and range pass-through filters to clean the reduce data size. As the limited field of view of the camera, the 2D Rplidar is also utilized for more accurate and robust mapping. And then ESDF is constructed through Euclidean Distance Transform (EDT), where we adopt the Voronoi diagram computation and use the enormous power of the GPU to speed up the process.

To plan a dynamically feasible collision-free trajectory within a limited time horizon efficiently, the NN is adopted to learn the BSCPs offline, which can then be evaluated efficiently online by avoiding the time consuming numerical optimization [10], while not compromising on motion constraints [11, 12] and optimization targets [13, 14]. Although a similar effect can be achieved with a look-up table, a densely constructed look-up table can be prohibitively large for high-dimensional systems, while a sparse one restricts the BSCPs' initial condition to limited selected states [15].

The gradient-free method PSO combined with NN is designed to construct a dynamically evolving single layer tree to select the optimal local target. Compared to the uniform sampling strategy [13, 16], and the random sampling strategy [17], PSO

The authors are with the Temasek Laboratories, National University of Singapore, Singapore 117411 (e-mail: yu.nico.zhou@gmail.com;).

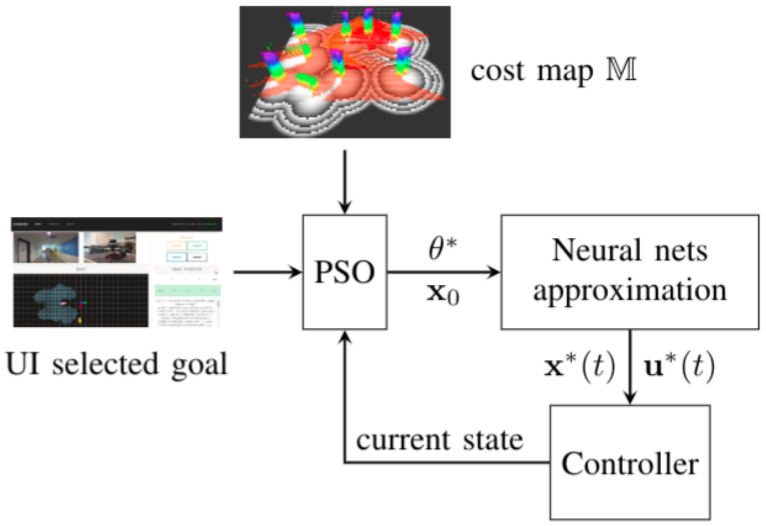


Fig. 1: Local planning framework

TABLE I: Comparison of two OG map construction methods

Method	Work	Step	Processing time (ms) CPU	GPU
Ray Casting	614400	2	210	8
Voxel Projection	1500	1	300	1.3

gives a much smoother trajectory by optimizing in the continuous domain, thus providing a fast evaluation of local trajectory. The local planning is done in a receding horizon fashion to guide the vehicle to maneuver safely in an unknown environment, the framework of which is shown in Fig. 1.

A. Mapping

In our system, an OG map and a ESDF are constructed by point clouds generated from both ZED depth map and Rplidar. Since the local map only contains recent sensor measurements with respect to the current robot location, the estimation error is small compared to a global map. All algorithms are executed on GPU achieving high accuracy and real-time processing time, which can significantly improve the real-time performance without any sacrifice of accuracy.

1) OG map: Our OG map is built with voxel projection[18] method. Different from the traditional ray casting method, which reprojects the pixel from depth image to correspondent depth voxel, the voxel projection method divides voxel space to scanlines and projects voxels in space to the image plane. The latter approach is faster when implemented on a GPU platform, as shown in Table I.

The depth image resolution we use is 640×480. The size of the map is 20m×20m×3m with a resolution of 0.2m, which means the grid volume is 100x100x15. Voxels in the grid map are projected to the camera coordinates and then to the image plane.

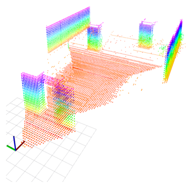
$$p = KT_i^{-1}\mathbf{v}^g/D_g, \tag{1}$$

where $\mathbf{v}^g$ is the voxel's grid coordinate. T_i^{-1} transfers the voxel from grid to camera coordinate, K is the camera matrix. D_g is the voxel's depth to the camera center. If the voxel projection is within the image plane, then the real depth of the voxel D_g is compared to the depth measurement D_m. We then classify all points in the volume to one of three states: unseen ($D_g > D_m$), empty ($D_g < D_m$), or near the surface ($D_g = D_m$). The OG map can be updated with the classified points information. Points near the surface are current obstacles. Empty voxels can be seen as feasible areas. The unseen voxels remain unchanged. The simulation result is shown in Fig. 2b. The MAV is in front of the sports hall Fig. 3a.

2) Exact Euclidean distance transform: To update the cost function for planning, the ESDF has to be constructed after we obtain the OG map. The EDT is applied to compute for each grid the Euclidean distance from itself to the closest OG. In order to decrease the computation complexity, the dimensionality reduction and partial Voronoi diagram computation[19] is applied to compute the exact EDT using the power of the modern GPUs.

In order to calculate EDT for 3D mapping with $N = n^3$ voxels, we first consider 2D case for each plane with $z = k, k \in [0, n-1]$. Take $k = c$ as an example, we get a 2D grid map with size $N = n^2$ with n columns and n rows. Let the upper left corner of 2D grid map be coordinate $(0, 0)$ and lower right corner $(n-1, n-1)$. Define $S_{i,j}$ as the nearest OG to column i,

(a) Simulation environment


(b) OG map

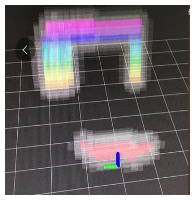
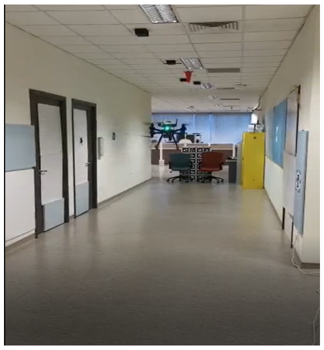

(c) EDT map

Fig. 2: Map in simulation testing


(a) Corridor environment

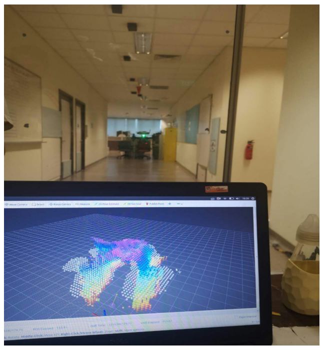

(b) map

Fig. 3: Map in real flight testing

among all OGs in row j, of current plane and $S_i = \{S_{i,j} | S_{i,j} \neq \text{NULL}, j \in [0, n-1]\}$ is the set of such closest OG for all grids in column i. Let S_i^* be the set of OGs whose Voronoi regions intersect the pixels in column i. Three steps are applied to achieve this to improve the efficiency of the computation by applying the three remarks mentioned in [20].

- First, compute $S_{i,j}$ for each voxel (i, j, c).
  The 1D Voronoi diagram using OGs in the same row only.
- Second, find S_i^* for each column i by using S_i.
- Third, for each voxel (i, j, c) using S_i^* to find the closest OG.
  As all the OGs in S_i^* are linked as a list in y-increasing order, checking the distance of two adjacent OGs in S_i^* with the each voxel $(i, j, c), j \in [0, n-1]$, the closest occupied grid can then be found out.

After obtained the closest occupied grid in 2d slices of our 3D space, the closest OG for each z-axis of voxels (a, b, k) can be obtained, where a, b are fixed x, y coordinates. In order to achieve this, thinking each plane as a row, thus we already have the nearest OG to the "column" k, then apply the second and third steps in the 2D calculation again. Since the N/n column computations are independent of each other, all calculations at the current dimension level can be performed in $O(N/p)$ time with p processors utilizing the parallel computing resource from GPU. Finally, We will be able to get the whole 3D EDT mapping. Update the map on top of the vehicle's current location to obtain the final local map. The simulation result is shown in Fig. 2c, where the voxels with gray color are our EDT map. The real-flight testing result is shown in Fig. 3, which was conducted in a corridor environment.

B. Motion Planning

1) Problem description: The local motion can be written as a MPC problem that finds the state and input to minimize the cost and subject to the system dynamics and certain constraints.

$$
\begin{array}{lrcl}
\min\limits_{u(t)} & J = c_f(\mathbf{x}(t_0+T)) + \int_{t_0}^{t_0+T} c(\mathbf{x}(t), \mathbf{u}(t)) & , & \\
\text{s.t.} & \dot{x} & = & f(\mathbf{x}, \mathbf{u}), \\
& \mathbf{x}(t_0) & = & \mathbf{x}_0, \\
& h(\mathbf{x}, \mathbf{u}) & = & 0, \\
& \mathbf{x} & \notin & \mathcal{O},
\end{array}
\tag{2}
$$

where $\mathbf{x}(t), u(t)$ represents the state and input. c_f and c are terminal and running costs. $\mathbf{x}_0$ is the initial condition. $h(\mathbf{x}, u)$ denotes the invariant constraint. $\mathcal{O}$ denotes the environment dependent obstacle constraint.

However, searching for valid trajectories directly over maps renders the optimization problem to be non-linear and non-convex. Therefore, it's challenging to solve this problem reliably and efficiently.

To avoid solving it directly, the BSCPs are designed to reformulate the problem. The boundary value problem or the controller can be constructed as:

$$\min_{\mathbf{x}(t),\mathbf{u}(t),t_f} = g(\mathbf{x}(t), u(t), t_f), \tag{3}$$

subject to an end state constraint $\|g(\mathbf{x}(t_f), \theta)\| < \epsilon$, where ϵ has a small positive value. g is the cost function, t_f and θ denote the final time and parameters of g respectively.

The designed non-linear controller is capable of driven the system subject to all the non environmental dependent constraints in Equation 2. Let $\hat{\mathbf{x}}(t), \hat{\mathbf{u}}(t), \hat{t}_f$ denote the state, input trajectories and final time generated by regulating the system to θ with a initial state $\mathbf{x}_0$. we obtain the mapping relationship

$$M : \langle \mathbf{x}_0, \theta \rangle \to \left\langle \hat{\mathbf{x}}(t), \hat{\mathbf{u}}(t), \hat{t}_f \right\rangle . \tag{4}$$

By fixing the x_0 as the current state of the vehicle, $\hat{\mathbf{x}}(t), \hat{u}(t)$ are then dependent on θ only. The cost function in Equation 2 can be modified as

$$\min_{\theta} J = \widetilde{c_f}(\theta) + \int_{t_0}^{t_0+T} \widetilde{c}(\theta) dt, \tag{5}$$

where $\widetilde{c_f}, \widetilde{c}$ represent terminal and running cost respectively.

2) Trajectory generation: We model the quadrotor as a 9 DOF system with a triple integrator on each of its x, y, z axis, and then construct BSCPs from a controller designed through dynamic programming. Assume $\mathbf{p}, \mathbf{v}, \mathbf{a}, \mathbf{j}$ are all 3×1 vectors representing the position, velocity, acceleration and jerk of the quadrotor respectively, the quadrotor can be simplified into the following model: $\dot{\mathbf{p}} = \mathbf{v},\ \dot{\mathbf{v}} = \mathbf{a},\ \dot{\mathbf{a}} = \mathbf{j}$. Define $\mathbf{x} = [\mathbf{p}^\intercal, \mathbf{v}^\intercal, \mathbf{a}^\intercal]^\intercal$ are the state of the quadrotor and $u = \mathbf{j}$ is the input. According to [14], the state and input constraints are:

$$\mathbf{v} \in [\mathbf{v}_{\min}, \mathbf{v}_{\max}],\ \mathbf{a} \in [\mathbf{a}_{\min}, \mathbf{a}_{\max}],\ \mathbf{j} \in [\mathbf{j}_{\min}, \mathbf{j}_{\max}]. \tag{6}$$

On each axis, the discrete dynamics is obtained by discretizing the state and input space of the single-axis triple integrator:

$$\mathbf{x}[k+1] = A\mathbf{x}[k] + bu[k], \tag{7}$$

where

$$A = \begin{bmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix},\ b = \begin{bmatrix} \frac{\Delta t^3}{6} \\ \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix}.$$

The target of the controller is to regulate the system from any initial state to the desired state $\mathbf{x}_d = [\theta, 0, 0]^\intercal$. Our end state constraint is

$$g(\mathbf{x}, \theta) = \mathbf{x} - \mathbf{x}_d = [p - \theta, v, a]^\intercal = 0. \tag{8}$$

Define the error state dynamic as

$$\delta\mathbf{x}[k+1] = A\delta\mathbf{x}[k] + bu[k], \tag{9}$$

where $\delta\mathbf{x} = \mathbf{x} - \mathbf{x}_d$. Define $V(\mathbf{x})$ as the approximation of the remaining integrated cost from $\delta\mathbf{x}$ to the origin. We have

$$V(\delta\mathbf{x}[n]) = C(\delta\mathbf{x}[n], u[n]) + V(\delta\mathbf{x}[n+1])), \tag{10}$$

where $C(\delta\mathbf{x}[n], u[n])$ is the instantaneous cost. To regulate $\delta\mathbf{x}$ to zero and satisfy the constraint of Equation 6. We define our cost function as

$$C(\delta\mathbf{x}, u) = \delta\mathbf{x}^\intercal R \delta\mathbf{x} + ru^2 + H(\delta\mathbf{x}, u), \tag{11}$$

where $\delta\mathbf{x}^\intercal R\delta\mathbf{x}$ represents target deviation; ru^2 denotes input penalty: we penalize the system from taking aggressive moves; $H(\delta\mathbf{x}, u) = \omega_v \eta^2(s_v, v_{\min}, v_{\max}) + \omega_a \eta^2(s_a, a_{\min}, a_{\max}) + \omega_j \eta(j, j_{\min}, j_{\max})^2$ penalizes the violation of the constraints in Equation 6. In which, $\eta(\kappa, \kappa_1, \kappa_2) = max(\kappa_1 - \kappa, 0) + max(\kappa - \kappa_2, 0), \kappa, \kappa_1, \kappa_2 \in R$ and $\omega_a, \omega_v, \omega_j$ are weights.

With the instantaneous cost, we can design the controller for each individuals axis of the vehicle. Our target is to find a policy map for u that minimize the value function for all the states, which can be achieved when the Bellman equation that satisfies

$$V^*(\delta\mathbf{x}[n]) = \min_{u[n] \in J} C(\delta\mathbf{x}[n], u[n]) + V^*(\delta\mathbf{x}[n+1])). \tag{12}$$

When the value iteration has converged, the policy map which can be seen as a lookup table

$$(\delta\mathbf{x}(t), u) = M(\delta\mathbf{x}_0), \tag{13}$$

that is, trajectories can be calculated by regulating the system from a given initial state to the origin. For an arbitrary state $\delta\mathbf{x}$, the real state trajectory can then be recovered through $\mathbf{x}(t) = \mathbf{x}_d + \delta\mathbf{x}$.

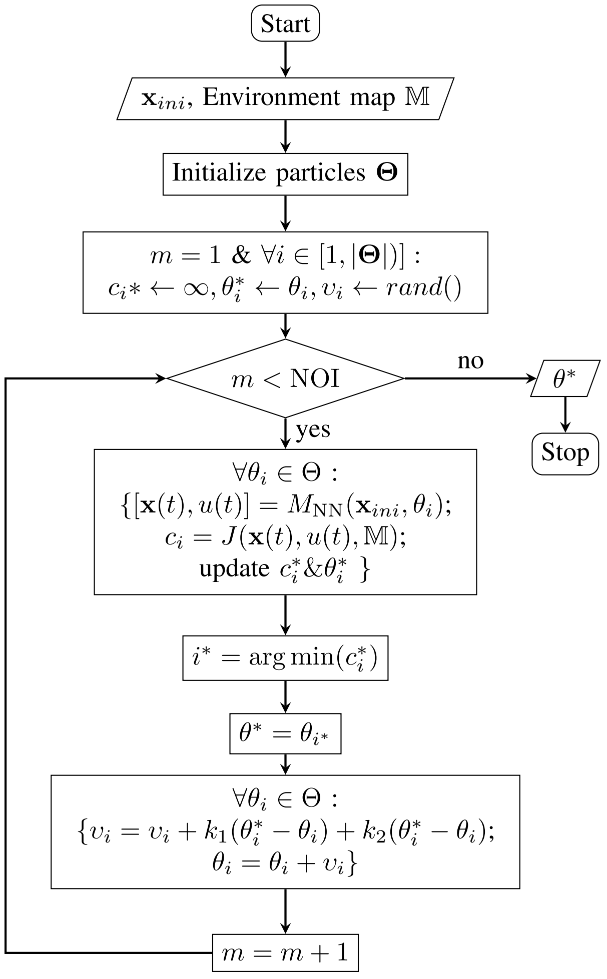

Start
$\mathbf{x}_{ini}$, Environment map $\mathbb{M}$
Initialize particles $\boldsymbol{\Theta}$
$m = 1$ & $\forall i \in [1, |\boldsymbol{\Theta}|)]$:
$c_i* \leftarrow \infty, \theta_i^* \leftarrow \theta_i, \upsilon_i \leftarrow rand()$
$m <$ NOI
no
θ^*
Stop
yes
$\forall \theta_i \in \Theta$:
$\{[\mathbf{x}(t), u(t)] = M_{\text{NN}}(\mathbf{x}_{ini}, \theta_i)$;
$c_i = J(\mathbf{x}(t), u(t), \mathbb{M})$;
update $c_i^* \& \theta_i^*$ $\}$
$i^* = \arg\min(c_i^*)$
$\theta^* = \theta_{i^*}$
$\forall \theta_i \in \Theta$:
$\{\upsilon_i = \upsilon_i + k_1(\theta_i^* - \theta_i) + k_2(\theta_i^* - \theta_i)$;
$\theta_i = \theta_i + \upsilon_i\}$
$m = m + 1$

Fig. 4: PSO algorithm

We propose to use a neural network to approximate the mapping policy M with M_{NN}. With the help of the modern parallel hardware, it makes algorithms such as PSO work in real-time. A four-layer $64 \times 128 \times 128 \times 41$ rectifier (ReLU) multi-layer perceptron (MLP) is applied and trained with Adam optimizer using PyTorch. The input of this NN includes only the 3x1 vector error position $\delta p(t) = p - \theta$ instead of the full state $\delta \mathbf{x}(t)$, thus limits the size of the network and makes it can be executed in a GPU-free device. The output contains the future trajectory $\delta \mathbf{p}(t)$ of the system sampled at 2 Hz for 20 seconds and u, which is a 41×1 vector.

The training data can be obtained through forwarding simulation with the policy M by random sampling $\delta \mathbf{x}_0$. The samples should be large enough making the generated trajectories are able to cover the state limits in Equation 6. The training and testing set size are 800,000 and 200,000 respectively. The batch size is 20 and the epoch is 12. We achieved an average mean squared error (MSE) of $4.72e^{-4}$ and maximum MSE of 0.071.

Although a larger and more complex NN can be used to further improve the training result, the simplified network we chose makes sure it can be evaluated efficiently in a CPU-only platform.

3) Path planning: The BSCPs and the PSO are combined for local motion planning. This section presents how to use the BSCPs for local motion planning, which is achieved by selecting the best target θ_p with the PSO algorithm. Compared with the traditional tree searching based technique, the gradient-free method not only has the advantages of it but also generates a solution with better quality not restricted to limited resolution.

The algorithm detail is shown in Fig. 4. The inputs are vehicle's initial state $\mathbf{x}_{ini}$ and the environment map $\mathbb{M}$. The particles in the algorithm represent the end state constraint. A finite number of particles Θ are first randomly initialized. Each particle θ_i is also assigned a velocity υ_i either randomly or with a special pattern which decides its future movement in the optimization space. θ_i^* represents each particle's best value during each iteration with its correspondent cost value c_i^*. NOI represents for number of iteration.

The cost function J is designed as follows

$$J(\mathbf{x}(t), u(t), \mathbb{M}) = \int_{t_0}^{t_0+T} H(\mathbf{x}(t)) + O(\mathbf{x}(t)). \tag{14}$$

TABLE II: Time consumption comparison

	PyTorch (TX2)	C++ (I7)
NN Approximation	$3\mu s$	$15\mu s$
Forward Simulation	-	$70\mu s$

$H(\mathbf{x}(t))$ evaluates the progress of the trajectory, which is calculated by measuring the remaining distance to the target with Dijkstra's algorithm. $O(\mathbf{x}(t))$ denotes penalization on the obstacles, which can be evaluated by adding up the costs of the grids that are occupied by the vehicle while at certain states. In our map $\mathbb{M}$, each grid is assigned with EDT, which provides a smooth map than a normal binary map. The MPC is executed at 5 Hz with a 20 seconds prediction horizon.

In the particle processing iteration, the state trajectories are obtained through our trained NN mapping M_{NN}. Then the cost of each particle is evaluated by the cost function J with the environment information $\mathbb{M}$. The best target θ^* among all the particles is recorded after each iteration.

If the PSO is limited to a single iteration only, the performance is almost the same as the methods in [13, 16, 21, 22], which constructs a local search tree to find the best motion. However, we update the each particle's velocity and location with the best target θ^* after traversing all the particles once, thus make the PSO iterate multiple times and converge to the optimal value gradually.

To test the reliability of the algorithm, we conducted hundreds of consecutive simulations with randomly sampled initial states and target positions in an indoor environment using both BSCPs, and it turns out that the vehicle can always safely reach the set goal. Regarding time consumption, we compare the NN approximation and forward simulating for 20 seconds. The results in Table shows that the NN approximation method helps to increase the efficiency on both CPU and GPU platform.

References

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

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

[3] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. W. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking.," in *ISMAR*, vol. 11, pp. 127–136, 2011.

[4] T. Howard, M. Pivtoraiko, R. A. Knepper, and A. Kelly, "Model-predictive motion planning: Several key developments for autonomous mobile robots," *IEEE Robotics Automation Magazine*, vol. 21, pp. 64–73, March 2014.

[5] D. Ferguson, T. M. Howard, and M. Likhachev, "Motion planning in urban environments: Part i," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1063–1069, Sep. 2008.

[6] U. Schwesinger, M. Rufli, P. Furgale, and R. Siegwart, "A sampling-based partial motion planning framework for system-compliant navigation along a reference path," in *2013 IEEE Intelligent Vehicles Symposium (IV)*, pp. 391–396, June 2013.

[7] M. Hwangbo, J. Kuffner, and T. Kanade, "Efficient two-phase 3d motion planning for small fixed-wing uavs," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 1035–1041, April 2007.

[8] M. Pivtoraiko, I. A. D. Nesnas, and A. Kelly, "Autonomous robot navigation using advanced motion primitives," in *2009 IEEE Aerospace conference*, pp. 1–7, March 2009.

[9] M. Lan, S. Lai, and B. M. Chen, "Towards the realtime sampling-based kinodynamic planning for quadcopters," in *2017 11th Asian Control Conference (ASCC)*, pp. 772–777, Dec 2017.

[10] J. Tordesillas, B. T. Lopez, J. Carter, J. Ware, and J. P. How, "Real-time planning with multi-fidelity models for agile flights in unknown environments," *arXiv preprint arXiv:1810.01035*, 2018.

[11] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, "Optimal trajectory generation for dynamic street scenarios in a frenét frame," in *2010 IEEE International Conference on Robotics and Automation*, pp. 987–993, May 2010.

[12] 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.

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

[14] M. Hehn and R. D'Andrea, "Real-time trajectory generation for quadrocopters," *IEEE Transactions on Robotics*, vol. 31, pp. 877–892, Aug 2015.

[15] E. Frazzoli, M. A. Dahleh, and E. Feron, "Maneuver-based motion planning for nonlinear systems with symmetries," *IEEE Transactions on Robotics*, vol. 21, pp. 1077–1091, Dec 2005.

[16] X. Yang, K. Sreenath, and N. Michael, "A framework for efficient teleoperation via online adaptation," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5948–5953, May 2017.

[17] A. A. Paranjape, K. C. Meier, X. Shi, S.-J. Chung, and S. Hutchinson, “Motion primitives and 3d path planning for fast flight through a forest,” *The International Journal of Robotics Research*, vol. 34, no. 3, pp. 357–377, 2015.

[18] B. Curless and M. Levoy, “A volumetric method for building complex models from range images,” 1996.

[19] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, “Parallel banding algorithm to compute exact distance transform with the gpu,” in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D ’10, (New York, NY, USA), pp. 83–90, ACM, 2010.

[20] C. R. Maurer, R. Qi, and V. Raghavan, “A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 2, pp. 265–270, 2003.

[21] P. Florence, J. Carter, and R. Tedrake, “Integrated perception and control at high speed : Evaluating collision avoidance maneuvers without maps,” in *Int. Workshop on the Algorithmic Foundations of Robotics*, 2016.

[22] M. W. Mueller, M. Hehn, and R. D’Andrea, “A computationally efficient algorithm for state-to-state quadrocopter trajectory generation and feasibility verification,” in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3480–3486, Nov 2013.