Robust Trajectory Execution for Multi-Robot Teams Using Distributed Real-time Replanning

Baskın Şenbaşlar, Wolfgang Hönig, and Nora Ayanian

University of Southern California, Los Angeles CA, USA,
{baskin.senbaslar, whoenig, ayanian}@usc.edu

Abstract. Robust trajectory execution is an extension of cooperative collision avoidance that takes pre-planned trajectories directly into account. We propose an algorithm for robust trajectory execution that compensates for a variety of dynamic changes, including newly appearing obstacles, robots breaking down, imperfect motion execution, and external disturbances. Robots do not communicate with each other and only sense other robots' positions and the obstacles around them. At the high-level we use a hybrid planning strategy employing both discrete planning and trajectory optimization with a dynamic receding horizon approach. The discrete planner helps to avoid local minima, adjusts the planning horizon, and provides good initial guesses for the optimization stage. Trajectory optimization uses a quadratic programming formulation, where all safety-critical parts are formulated as hard constraints. At the low-level, we use buffered Voronoi cells as a multi-robot collision avoidance strategy. Compared to ORCA, our approach supports higher-order dynamic limits and avoids deadlocks better. We demonstrate our approach in simulation and on physical robots, showing that it can operate in real time.

1 Introduction

Motion planning for multi-robot systems is particularly important in cases where many robots must interact with each other in confined spaces, potentially with many obstacles. Examples include coordination of robots in warehouses [18], traffic management at intersections [6], and airport management [12]. Modern planning algorithms can find trajectories that effectively coordinate hundreds of robots while approximately optimizing objectives such as total energy used [9]; however, all such solutions assume that the resulting trajectories can be executed nearly perfectly, which is an unrealistic assumption for teams of hundreds of robots that must operate persistently.

To compensate for changes in the environment or imperfect execution, one might apply cooperative collision avoidance strategies, such as ORCA [2], at runtime. However, such algorithms often operate locally and do not take the pre-planned trajectories into account. Robust trajectory execution, on the other hand, avoids future collision more effectively because it directly considers pre-planned trajectories. Consider the example in Fig. 1(a), where two robots must

swap positions. The pre-planned trajectories are collision-free, but they do not consider the newly introduced obstacle and the blue robot does not start at its correct location. However, the pre-planned trajectories can be used as guidance for replanning. In this example, robots can get stuck if a local cooperative collision avoidance strategy is applied. Using our method, the robots can successfully swap positions, while staying as close as possible to the pre-planned trajectories, as in Fig. 1(b). Our method is fully distributed and requires no communication. The robots only need to know their own trajectories and be able to sense other robots' positions and the obstacles around them.

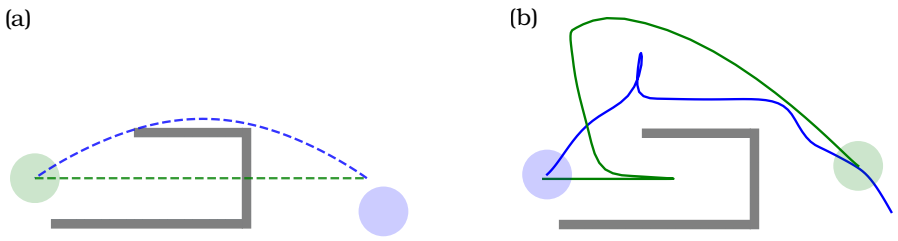

Fig. 1. (a) Two robots (green and blue circles) are tasked with following their pre-planned trajectories (green and blue dashed lines). The initial plans were created without the knowledge of the obstacle (gray) and the blue robot does not start at its planned initial position. (b) Our approach computes smooth trajectories in real-time, avoiding both the new obstacle and other robots while staying close to the pre-planned trajectory.

Robust trajectory execution is an extension of cooperative collision avoidance where the objective is to stay as close to the originally planned trajectories as possible. In contrast, traditional collision avoidance methods frequently only take a desired velocity, desired goal state, or desired action as input (we discuss these in more detail in Section 2). Our method relies on *Buffered Voronoi Cells* (BVC) [19] as the underlying cooperative collision avoidance strategy and retains the same theoretical guarantees. We employ a novel combination of trajectory optimization and discrete search-based planning using a dynamic receding horizon approach. The discrete search allows us to avoid local minima effectively even in difficult scenarios, while the trajectory optimization generates smooth trajectories that are collision-free.

The main contribution of this work is a novel distributed algorithm for robust trajectory execution that considers higher-order dynamic limits. It also compensates for a variety of dynamic changes, including imperfect motion execution of robots, newly appearing obstacles, robots breaking down, or external disturbances. We show in simulations that our method avoids deadlocks better than ORCA [2]. Furthermore, we implement and test our approach on a team of six differential drive robots with several dynamic environmental changes.

2 Related Work

Our method is closely related to cooperative collision avoidance such as reciprocal velocity obstacles, buffered Voronoi cells, and safety barrier certificates. Methods based on reciprocal velocity obstacles (RVO) [3] assume that robots continue with constant velocity and compute the safe configuration space such that no other robot might collide for the time horizon. Many extensions of the RVO method have been proposed, see [1] for an extensive overview. Buffered Voronoi Cells (BVC) [19] compute the safe configuration space for a robot by its Voronoi cell shifted by the physical extent of the robot. Safety barrier certificates achieve collision-free operation by modifying a user-specified controller such that no collision can occur [17]. Our robust trajectory execution approach uses cooperative collision avoidance at its core (specifically BVC), while extending it to minimize the difference to the original trajectories (rather than just a preferred velocity as in [1], preferred control input as in [17], or difference over a fixed time horizon as in [19].)

Our method is inspired by our previous work on offline planning for robotic teams [9] and uses the same optimization framework based on Bézier curves to generate trajectories, although with a different cost function. Similar to previous work we use discrete search to quickly get out of local minima, but do so in a distributed manner.

While our approach naturally works in multi-robot settings, some of the methods are inspired by single-robot optimization and collision avoidance. Local collision avoidance for single robots such as UAVs can be formulated as optimization problems [13,16]. In both cases collisions are considered as a soft constraint in the cost function using a Euclidean (Signed) Distance Field. In contrast, our formulation uses a hard constraint allowing us to easily detect infeasible trajectories. The optimization can use a discrete plan as an initial guess [13] or shift the existing trajectory based on newly appearing obstacles [16]. In contrast, our approach shifts the existing trajectory whenever possible, while falling back to an efficient discrete planner with dynamic receding horizon to avoid local minima.

3 Problem Formulation

The general problem we would like to solve can be formulated as follows. Consider a group of m robots. Each robot i is given the following:

$\mathbf{o}_i(t)$: original trajectory ($\mathbb{R} \to \mathbb{R}^n$) of i^{th} robot where time $t \in [0, T_i]$,

c : order of derivative up to which smoothness is required,

$R(\mathbf{p})$: convex collision shape of any robot at position $\mathbf{p}$,

γ_k : dynamic limit of the robot for the k^{th} derivative of its trajectory.

Each robot i can sense the positions $\{\mathbf{p}_1, \ldots, \mathbf{p}_m\}$ of other robots as well as the current occupied space $\mathcal{O}_i$ around it[1]. We represent $\mathcal{O}_i$ as a set of θ convex

[1] Since our approach can accommodate many sensing modalities, we do not provide a specific sensing capability in the general problem.

obstacles. Robots are unaware of the other robots' planned trajectories, and cannot communicate with each other. Each robot i must execute a trajectory $\mathbf{f}_i(t)$, where $\mathbf{f}_i(t)$ is a solution to the following optimization problem:

$$\begin{aligned} &\text{minimize } \int_0^{T_i} \|\mathbf{f}_i(t) - \mathbf{o}_i(t)\|^2 \, dt \\ &\text{subject to} \\ &\qquad \mathbf{f}_i(t) \text{ is continuous up to degree } c, \\ &\qquad \frac{d^j \mathbf{f}_i}{dt^j}(0) = \frac{d^j \mathbf{p}_i}{dt^j}(0) \text{ for } j \in \{0, 1, ..., c\} \\ &\qquad \mathbf{f}_i(t) \text{ is collision-free, and} \\ &\qquad \left\| \frac{d^k \mathbf{f}_i(t)}{dt^k} \right\| \leq \gamma_k \text{ for all desired } k, \\ &\text{where } t \in [0, T_i]. \end{aligned} \tag{1}$$

We solve this problem approximately, using a dynamic receding horizon approach iteratively. At every iteration K, robot i plans a trajectory $\mathbf{f}_i^K(t)$ that starts at the robots' current position and is safe to execute up to the user-provided period δt. We set $R(\mathbf{p})$ to a sphere with radius r_s centered at $\mathbf{p}$.

4 Preliminaries

This section introduces important mathematical concepts and notations that will be used throughout the paper.

4.1 Buffered Voronoi Cell

Given a set of m robots with positions $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m \in \mathbb{R}^n$ and radii $r_s \in \mathbb{R}$, the buffered Voronoi cell $\mathcal{V}_i$ of robot i is defined as [19]:

$$\mathcal{V}_i = \left\{ \mathbf{p} : \forall_{j \neq i} \frac{\mathbf{p}_j - \mathbf{p}_i}{\|\mathbf{p}_j - \mathbf{p}_i\|} \cdot \mathbf{p} - \frac{\mathbf{p}_j - \mathbf{p}_i}{\|\mathbf{p}_j - \mathbf{p}_i\|} \cdot \frac{\mathbf{p}_j + \mathbf{p}_i}{2} + r_s \leq 0 \right\}, \tag{2}$$

where $\|\mathbf{p}\|$ is the L^2-norm of the vector $\mathbf{p}$.

The inequality inside (2) defines a hyperspace $\mathcal{S}_i^j$ bounded by hyperplane $\mathcal{H}_i^j$ that separates point $\mathbf{p}_i$ from $\mathbf{p}_j$. $\mathcal{H}_i^j$ has normal $\alpha_i^j \in \mathbb{R}^n$ and distance $\beta_i^j \in \mathbb{R}$ along α_i^j such that

$$\alpha_i^j = \frac{\mathbf{p}_j - \mathbf{p}_i}{\|\mathbf{p}_j - \mathbf{p}_i\|} \text{ and } \beta_i^j = \alpha_i^j \cdot \left(\frac{\mathbf{p}_i + \mathbf{p}_j}{2} \right) - r_s. \tag{3}$$

For a given buffered Voronoi decomposition of the space, any point $\mathbf{p} \in \mathbb{R}^n$ can be inside of at most one of the buffered Voronoi cells. We use this property in order to avoid robot-to-robot collisions.

Using the hyperspaces $\mathcal{S}_i^j$ we can reformulate $\mathcal{V}_i$ as follows:

$$\mathcal{V}_i = \bigcap_{j \neq i} \mathcal{S}_i^j \text{ , where } \mathcal{S}_i^j = \left\{ \mathbf{p} : \alpha_i^j \cdot \mathbf{p} - \beta_i^j \leq 0 \right\}. \tag{4}$$

Thus, we can compute the buffered Voronoi cell of any robot i as the set of the hyperplanes $\mathcal{H}_i^j$ in $O(m)$ time.

4.2 Bézier Curve

A degree d Bézier curve $\mathbf{f}(t)$ implicitly parametrized by duration T is defined by $d+1$ control points $\mathbf{P}_0, \mathbf{P}_1, ..., \mathbf{P}_d \in \mathbb{R}^n$ such that

$$\mathbf{f}(t) = \sum_{i=0}^{d} \mathbf{P}_i \binom{d}{i} \left(\frac{t}{T}\right)^i \left(1 - \frac{t}{T}\right)^{d-i}, \quad 0 \leq t \leq T. \tag{5}$$

The curve starts at $\mathbf{P}_0$ and ends at $\mathbf{P}_d$, however does not interpolate other control points. A Bézier curve lies completely inside the convex hull of its control points [7]; we leverage this property to avoid robot-to-obstacle collisions.

We use splines as trajectories with user-specified number of pieces (l) and degree (d), where each piece is a degree d Bézier curve. Given a trajectory $\mathbf{f}_i^K(t)$ for robot i with l pieces and duration T_i^K at iteration K, $T_{i,j}^K$ denotes the duration of the j^{th} piece where $j \in \{1, 2, \ldots, l\}$. $\mathbf{f}_{i,j}^K(t)$ denotes the j^{th} piece of the trajectory with implicit duration parameter $T_{i,j}^K$ where $t \in [0, T_{i,j}^K]$. $\mathbf{P}_{i,j,\rho}^K$ denotes the ρ^{th} control point of the j^{th} piece where $\rho \in \{0, 1, \ldots, d\}$.

4.3 Trajectory Optimization using Quadratic Programming

Our replanning approach utilizes *quadratic programming* (QP) for trajectory optimization. The decision variables $\mathbf{x}$ are the concatenated Bézier curve control points. The overall structure of our quadratic optimization problem is as follows:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\mathbf{x}^T H \mathbf{x} + \mathbf{x}^T \mathbf{g} \\ \text{subject to} \quad & \mathbf{lbA} \leq A\mathbf{x} \leq \mathbf{ubA}. \end{aligned} \tag{6}$$

A quadratic cost function is represented using the matrix H and the vector $\mathbf{g}$. The quadratic cost function we use is described in Section 5.2.

The constraints are represented using the matrix A with vectors $\mathbf{lbA}$ and $\mathbf{ubA}$. Notice that all constraints should be linear in the decision variables. There are three types of constraints we impose on the curves: *initial point constraints*, *continuity constraints*, and *hyperspace constraints*. An initial point constraint on a Bézier curve requires the initial point of the curve to be equal to a given vector in a specified degree of differentiation. This translates to n linear constraints on control points, n being the dimension we are working in. A continuity constraint between curve j and curve $j+1$ requires the end of curve j to be equal to the

beginning of curve $j+1$ at any order of differentiation. We take the vector difference of those values and require it to be equal to $\mathbf{0}$. This translates to n linear constraints on control points. A hyperspace constraint requires all control points of a curve to be on a specific side of a hyperplane. If a curve has $d+1$ control points, this translates to $d+1$ constraints on control points. All three types of constraints are linear in control points. The exact construction is discussed in a previous work [9].

5 Approach

Replanning is done at a fixed period of δt. In each iteration K, we sense the other robots' positions to compute the buffered Voronoi cell $\mathcal{V}_i$, update our current representation of the occupied space ($\mathcal{O}_i$), and compute a trajectory $\mathbf{f}_i^K(t)$. The planning horizon τ' is automatically adjusted, but the desired planning horizon τ can be provided.

We execute the following three major components iteratively: *discrete planning* that is used to efficiently plan around new obstacles, *trajectory optimization* to generate smooth and collision-free trajectories, and *temporal rescaling* to enforce the dynamic limits of the robot (see Fig. 2).

In the beginning of each iteration, we check several conditions to decide if discrete planning is required. If discrete planning is required, we execute a discrete search that results in a discrete path that is collision-free but not smooth. We use this discrete path as an initial guess in trajectory optimization. If discrete planning is not required, we use the control points of the previous plan as the initial guess.

We construct a QP with hard constraints for trajectory optimization in a slightly different way depending on whether discrete planning was executed or not. In both cases, buffered Voronoi cells are used to ensure collision-free operation for time δt and collisions with static obstacles are avoided for the planning horizon.

Dynamic limits cannot be represented as linear constraints in our QP. Thus, we check dynamic limit violations in the temporal rescaling stage that runs after optimization. While dynamic limits are violated, we increase the durations of all trajectory pieces uniformly, and since the initial point constraints are violated when the durations are increased, we re-solve the QP.

At the end of each iteration, each robot has its trajectory $\mathbf{f}_i^K(t)$ that is guaranteed to be collision-free up to time δt; is continuous up to the c^{th} derivative; obeys the dynamic limits of the robot; tries to stay close to the original trajectory; and is a good starting point for the next iteration. We execute this trajectory for a period of δt and replan for the next iteration.

5.1 Discrete Planning

Robot i executes discrete planning if any of the following conditions are true, where $\psi = K\delta t$ is the current time:

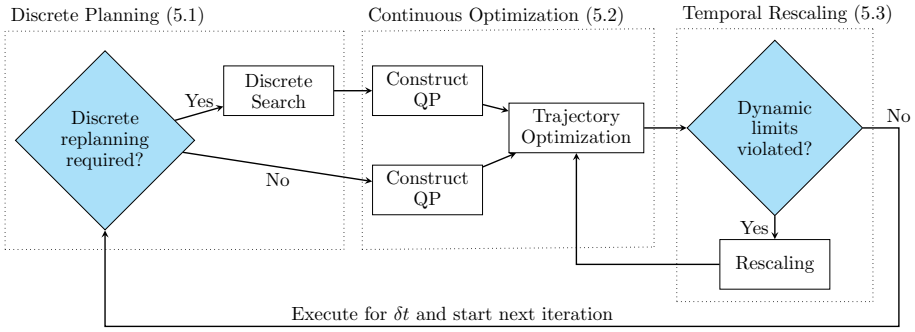

Fig. 2. Overview of the replanning pipeline.

1. The original trajectory is not collision-free for the desired time horizon τ, i.e., $\exists t \in [\psi, \psi + \tau] : R(\mathbf{o}_i(t)) \cap \mathcal{O}_i \neq \emptyset$,
2. The first piece of the previously planned trajectory is outside the robot's buffered Voronoi cell, i.e., $\exists t \in [0, T_{i,1}^{K-1}] : \mathbf{f}_{i,1}^{K-1}(t) \notin \mathcal{V}_i$, or
3. The previously planned trajectory is not collision-free for the desired time horizon τ, i.e., $\exists t \in [0, \tau] : R(\mathbf{f}_i^{K-1}(t)) \cap \mathcal{O}_i \neq \emptyset$.

The first condition handles cases where previously unknown obstacles block the pre-planned path of a robot. The second condition handles cases where previously unknown robots appear and cases where robots are close and moving towards each other. The third condition handles dynamic obstacles, and also infeasibilities and numerical issues that resulted in a trajectory with collisions in the previous iteration. Sections 5.4 and Section 6.1 detail reasons for infeasibilities and numerical issues, respectively.

Discrete planning uses a dynamic receding horizon approach. First, we find the earliest time $\tau' \in [\min(\tau, T_i - \psi), T_i - \psi]$ where the original trajectory is collision-free at time $\psi + \tau'$ with respect to both obstacles and other robots. Second, we use a discrete graph search to find a path from the robot's current location to $\mathbf{o}_i(\psi + \tau')$ that avoids both static obstacles and other robots. If τ' does not exist or no solution path exists, we skip the discrete planning stage and construct the QP as if discrete planning was not required. Third, we use the first l segments of the discrete path to uniformly place the new estimated control points on top of those segments. In case the discrete path has fewer than l segments, the last discrete segment is shared between multiple Bézier curves. Finally, we adjust $T_{i,j}^K$ relative to the segment lengths and scale by τ', such that we would arrive at time $\psi + \tau'$ at $\mathbf{o}_i(\psi + \tau')$ if we followed the discrete path with constant speed. To guarantee collision-free operation, we ensure that $T_{i,1}^K \geq \delta t$ in any case.

An example is shown in Fig. 3 (a), with parameters $l = 4$ and $d = 7$. Discrete planning is executed because the original trajectory (green dashed line) passes through an obstacle. We find the earliest time τ' such that $R(\mathbf{o}_g(\psi + \tau'))$ does not intersect with the obstacle and the blue robot where g is the green robot.

The discrete planner is then used to find a path (green dotted line) that avoids both the static obstacle and the other (blue) robot, with a total of six path segments. The first four segments ($l = 4$) are used to place new guesses of Bézier control points (blue, green, red, and cyan circles). Notice that since $d = 7$, each curve has eight control points, while some of the points are overlapping. The duration $T^K_{i,j}$ for each Bézier curve is adjusted linearly according to its segment length; for example, the duration of the segment with the red control points ($T^K_{i,3}$) is approximately twice as long as the duration of the segment with the green control points ($T^K_{i,2}$).

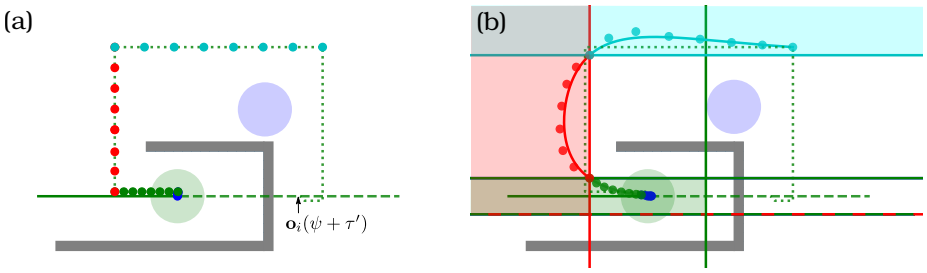

Fig. 3. Example at $t = 3.9\,\mathrm{s}$. (a) Discrete path around an obstacle and other robot back to the original trajectory. (b) Continuous trajectory split into four pieces and respective hyperspaces.

5.2 Continuous Optimization

We compute a new trajectory by formulating a quadratic program where the decision variables are the concatenated control points of the pieces. The parameters d and l of the pieces (see Section 4.2) are provided by the user. If discrete planning is not performed, initial guesses of the decision variables are copied from the previous iteration, and the durations $T^K_{i,j}$ are set uniformly to $\frac{\min(\tau, T_i - \psi)}{l}$. If discrete planning is performed, initial variables and durations are calculated from the discrete path as explained in Section 5.1. The objective that we minimize is defined

$$\sum_{e=1}^{\mathcal{E}} \lambda_e \int_0^{T_i^K} \left\| \frac{d^e \mathbf{f}_i^K(t)}{dt^e} \right\|^2 dt + \sum_{j=1}^{l} \theta_j \left\| \mathbf{P}^K_{i,j,d} - \chi_j \right\|^2, \tag{7}$$

where χ_j is the point where we want piece j to end. The first term of the objective minimizes the energy along the trajectory, and is the combination of integrated squared derivatives up to user provided degree $\mathcal{E}$ with weights λ_e [9,14]. The second term of the objective penalizes deviation from the given end points for each piece of the trajectory with different weights θ_j. In case discrete planning is performed, we attempt to get to the position of the last guessed control point,

i.e., we set θ_l to a positive value, enforcing $\mathbf{P}^K_{i,l,d} = \chi_l$, and $\theta_j = 0, \forall j < l$. If discrete planning is not performed, we attempt to stay as close as possible to the original trajectory, i.e., we set $\theta_1 = 0$, and $\theta_j, \forall j \geq 2$ to positive values (increasing with j) and $\chi_j = \mathbf{o}_i(\psi + \sum_{u=1}^{j} T^K_{i,u})$. The matrix H (see Section 4.3) for the first term of our objective can be constructed as in our previous work [9]. The second term is a quadratic function of the control points; hence it is straightforward to construct the H matrix and the $\mathbf{g}$ vector.

For robot-to-robot collision avoidance, the buffered Voronoi hyperplanes are computed according to (3) and $m-1$ hyperspace constraints are added for the first piece. These constraints ensure that the first piece stays inside $\mathcal{V}_i$ because of the convexity of $\mathcal{V}_i$ and the convex hull property of Bézier curves. As long as $T^K_{i,1} \geq \delta t$ and all other robots stay inside their Voronoi cells up to time δt, we can be sure that no robot-to-robot collision will occur up to time δt.

For robot-to-obstacle collision avoidance we compute separating hyperplanes between convex obstacles $\mathcal{O}_i$ for each curve piece j. Let M^b_j be the hyperplane that seperates the initially guessed control points of the j^{th} piece from the b^{th} convex obstacle obtained from $\mathcal{O}_i$ (these can be computed, e.g., using support vector machines [4]). We shift each hyperplane towards its obstacle and than shift it back using the radius r_s to account for the physical extent of the robot. We add hyperspace constraints as before, requiring control points of the j^{th} piece in the non-occupied side of each hyperplane M^b_j. These constraints ensure that no robot-to-obstacle collision will occur up to time τ'. In case discrete planning was executed, we additionally treat other robots as static obstacles. Fig. 3(b) shows the effective set of hyperspaces for our example.

Moreover, we add continuity constraints that enforce the continuity requirements between pieces and initial point constraints that enforce continuity requirements between iterations.

All constraints are linear and matrix A and its bounds can be constructed as in Section 4.3. The number of decision variables in our QP is $l(d+1)n$. Let $\mathsf{\theta}'$ describe the number of considered static obstacles, i.e., $\mathsf{\theta}'$ is equal to $\mathsf{\theta} + (m-1)$ if discrete planning was performed and $\mathsf{\theta}$ otherwise. We add $(m-1)(d+1) + \mathsf{\theta}' l(d+1) + (c+1)nl$ linear constraints, where the terms refer to the Voronoi hyperspace, obstacle hyperspace, and continuity constraints, respectively. For our example in Fig. 3, we have $n = 2$, $m = 2$, $d = 7$, $l = 4$, and $c = 2$. Thus, we have 64 decision variables and $8 + 128 + 24 = 160$ linear constraints.

5.3 Temporal Rescaling

Since we use fixed durations of the pieces and do not account for the dynamic limits of the robot during optimization, the resulting trajectory may violate the dynamic limits of the robot. After trajectory optimization, we calculate the maximum magnitudes $\mathit{\Gamma}_k$ of the k^{th} derivatives of the curve, and check if there exists a k such that $\mathit{\Gamma}_k > \gamma_k$, where γ_k is the dynamic limit of the robot in the k^{th} derivation degree. If that is the case, we uniformly scale the piece durations $T^K_{i,j}$, and re-run the trajectory optimization with the same exact constraints using

the previous result as the initial guess. If the dynamic limits are not violated, no temporal rescaling is needed and the trajectory is feasible.

5.4 Theoretical Guarantees

For robot-to-robot collision avoidance our approach uses buffered Voronoi cells which has the following theoretical guarantee: if robots start in a collision-free configuration (that is, $\|\mathbf{p}_i - \mathbf{p}_j\| \geq 2r_s, \forall i \neq j$), then all future configurations are collision-free. However, this guarantee has only been proven for the case of synchronous robot execution, if robots have first-order integrator dynamics ($c = 0$), and if they execute their trajectories perfectly [19]. The QP in our formulation has additional constraints that can cause it to be infeasible. However, in this case, one can simply fallback to the QP formulation of the BVC approach to retain the same theoretical guarantee.

Formal guarantees under arbitrary disturbances and higher order dynamics cannot be provided. In fact, our QP can fail if it is not feasible to satisfy all safety and continuity constraints under the given dynamic limits. However, our empirical evaluation presented in Section 6.1 shows that the QP rarely fails and even if it does, the robots do not collide with each other and the obstacles since the QP becomes feasible in the following iterations. In addition, our QP formulation allows us to easily detect failure cases because we model all safety-critical parts as hard constraints.

Similar to other work, there are no formal liveness guarantees and there might be deadlocks [19]. Nevertheless, our approach works in practice for robots with higher-order dynamics, if robot execution is asynchronous, or trajectories are not executed perfectly.

6 Evaluation

We implement our approach in C++. We use an occupancy grid as the environment representation, because previous work has shown that such data structures can be updated in real-time on robots that are equipped with a LIDAR sensor or an RGB-D camera. In particular, OctoMap [10] is an octree-based 3D occupancy grid that can be run on unmanned aerial vehicles with at least 4 Hz update rate [13]. OctoMaps are memory efficient, but update operations can show high execution time variance. For local replanning, occupancy grids using ring buffers as data structures have been shown to achieve near constant execution time [16]. Our implementation uses a simple pre-initialized 2D occupancy grid.

We use the CVXGEN-package [11] to generate small QPs to find separating hyperplanes between control points and obstacles. We test with qpOASES [8] and OSQP [15] as QP solvers; both are open source and have been shown to work well in model predictive control scenarios.

A supplemental video containing some of our simulations and physical experiments is available at https://youtu.be/LbWRvLfdwTA.

6.1 Simulation

We test our algorithm in a simulation running on a laptop computer (i7-4700MQ 2.4 GHz, 16 GB) with Ubuntu 16.04 as the operating system.

In the first set of experiments, we test the scalability of our method in terms of the number of pieces l we plan for, the number of occupied cells θ in the occupancy grid and the number of robots m. Our results are summarized in Tables 1, 2, and 3, where t_{avg} is the average time that qpOASES takes per iteration. Our algorithm scales well with the number of robots. In terms of number of curves, our algorithm has almost the same performance up to $l = 10$. For the simulations and physical experiments we did, we never needed more than $l = 4$. The bottleneck of our algorithm is the number of occupied cells in the occupancy grid. However, as it can be seen in Table 2, our algorithm still has real-time capability when considering hundreds of occupied cells, assuming a 10 Hz execution. When we use OSQP instead of qpOASES, our implementation takes significantly more time if we consider many obstacles. For example, when we do experiment 7 using OSQP, it takes 297 ms on average.

#	l	θ	m	t_{avg} [ms]
1	4	0	4	10
2	8	0	4	15
3	10	0	4	13
4	12	0	4	27
5	16	0	4	107

Table 1. Runtime with varying curve count l.

#	l	θ	m	t_{avg} [ms]
6	4	4	4	9
7	4	62	4	28
8	4	196	4	47
9	4	213	4	69
10	4	1250	4	253

Table 2. Runtime with varying occupied cells θ.

#	l	θ	m	t_{avg} [ms]
11	4	5	4	9
12	4	5	8	10
13	4	5	16	13
14	4	5	32	15
15	4	5	64	14

Table 3. Runtime with varying robot count m.

			ORCA		DS+ORCA		Our Method		
#	m	θ	t_{avg} [ms]	s	t_{avg} [ms]	s	t_{avg} [ms]	s	QP failures [%]
16	2	4	< 1	0	< 1	2	7	2	0.00
17	4	12	< 1	0	< 1	4	10	4	0.30
18	8	30	< 1	4	< 1	8	13	8	0.00
19	16	9	< 1	13	< 1	16	12	16	0.08
20	32	30	< 1	23	< 1	32	16	32	0.09

Table 4. Comparison of our method, ORCA, and DS+ORCA with respect to average computation time (t_{avg}), the number of robots that reach their destinations (s), and the percentage of time that our QP fails.

We also compare our method to two ORCA variants in the second set of experiments. In the first ORCA variant, we use the RVO2 library [2] and set the preferred velocities at time ψ to $\mathbf{o}'_i(\psi)$ if $\mathbf{p}_i \approx \mathbf{o}_i(\psi)$ or to $\frac{\mathbf{o}_i(\psi)-\mathbf{p}_i}{\delta t}$ otherwise. In the second ORCA variant, we combine ORCA and our discrete planning

method with a dynamic receding horizon approach (denoted as DS+ORCA). We demonstrate that this variant resolves deadlocks better than the first variant. For our method we use $\delta t = 0.1\,\text{s}$, $l = 4$, and $d = 7$ and for the ORCA variants we use $\delta t = 0.01\,\text{s}$. The results are summarized in Table 4. All robots using our method or DS+ORCA reach their destinations, while robots using ORCA can easily get stuck around obstacles. Our method takes more time in computation compared to the ORCA variants, but produces smooth curves up to a user-defined smoothness. We use $c = 2$ in our experiments meaning that the generated trajectories are continuous in position, velocity, and acceleration. The ORCA variants, on the other hand, provide smoothness guarantees up to $c = 0$ only, i.e., velocities can jump between iterations. Furthermore, the ORCA variants must sense the other robots' velocities and positions while our approach relies on positions only.

We also report the percentage of time our QP fails, which happens no more than 0.3 % of the time. Notice that even if our QP fails, robots do not collide with each other and the obstacles, because the QP becomes feasible in the following iteration after 100 ms. There are two reasons for QP failures: infeasibilities, which are explained in Section 5.4, and numerical issues. The numerical issues stem from separating hyperplane calculations between robots and obstacles. We use hard-margin SVMs to calculate separating hyperplanes. When robots get too close to obstacles, small epsilon values in SVM optimization may result in invalid hyperplanes, and hence QP fails. The original trajectories and the occupancy grids in some experiments are shown in Fig. 4 and the supplemental video contains selected simulations.

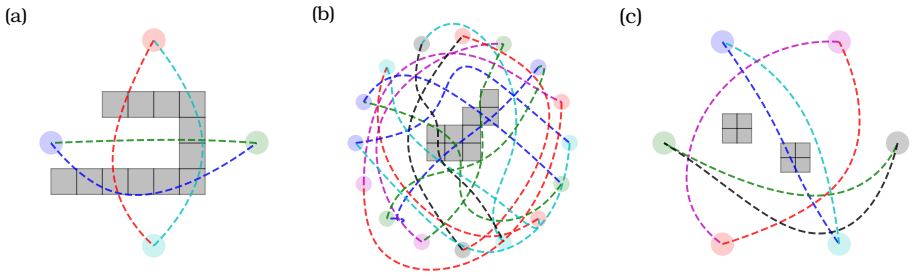

Fig. 4. The original trajectories and the occupancy grids in the simulation experiments 17 (a), 19 (b), and the physical experiment (c).

6.2 Physical Robots

We implement our approach on six differential drive robots (iRobot Create2) that are equipped with one of ODROID C1+ or ODROID XU4 single-board computers. Those computers run Ubuntu 16.04 with ROS Kinetic, but C1+ has

very limited computation capabilities (ARM Cortex-A5, max. 10 W). The robots are arranged in a circle (2 m radius) and are tasked with swapping sides (Fig. 4c). We plan the original trajectories with one static obstacle using a centralized planner [5]. Each robot receives the position information of all other robots using a motion capture system. A trajectory tracking controller and our algorithm run on-board at a frequency of 10 Hz.

We conduct several experiments and add an additional obstacle, change the robots initial position, disturb the robots during run-time, or artificially stop one of the robots. In all cases robots successfully avoid collisions and in many cases they reach their final destination within the originally planned durations. We also saw a few cases where robots got into a deadlock, which we attribute to the fact that the robots, unlike the simulation, cannot execute very low velocity commands. The supplemental video includes recordings of our experiment.

7 Conclusion

We present a method for robust trajectory execution that takes pre-planned trajectories as input and compensates for a variety of dynamic changes, including imperfect motion execution, newly appearing obstacles, robots breaking down, or external disturbances. Our approach does not require communication between the robots. We use a novel planning strategy employing both discrete planning and trajectory optimization with a dynamic receding horizon approach. We demonstrate in simulation and on physical robots that we can generate smooth trajectories in real-time, while avoiding deadlocks successfully. In comparison, ORCA neither generates smooth trajectories nor avoids deadlocks in our test cases.

In future work we would like to conduct additional experiments with robots using on-board perception and flying robots, handle dynamic obstacles, and consider communication between robots to improve their plans. We also would like to actively address numerical issues and QP infeasibilities.

Acknowledgements

This research was supported in part by Office of Naval Research grant N00014-14-1-073 and National Science Foundation grant 1724399. B. Şenbaşlar gratefully acknowledges the support from the Fulbright program sponsored by U.S. Department of State.

References

1. Alonso-Mora, J., Beardsley, P.A., Siegwart, R.: Cooperative collision avoidance for nonholonomic robots. IEEE Transactions on Robotics (T-RO) **34**(2), 404–420 (2018)
2. van den Berg, J., Guy, S.J., Lin, M.C., Manocha, D.: Reciprocal n-body collision avoidance. In: Int. Symposium of Robotic Research (ISRR), pp. 3–19 (2009). Software available at http://gamma.cs.unc.edu/RVO2/
3. van den Berg, J.P., Lin, M.C., Manocha, D.: Reciprocal velocity obstacles for real-time multi-agent navigation. In: IEEE International Conference on Robotics and Automation (ICRA), pp. 1928–1935 (2008)
4. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning **20**(3), 273–297 (1995)
5. Debord, M., Hönig, W., Ayanian, N.: Trajectory planning for heterogeneous robot teams. In: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS) (2018). Accepted. To appear.
6. Dresner, K.M., Stone, P.: A multiagent approach to autonomous intersection management. Journal of Artificial Intelligence Research (JAIR) **31**, 591–656 (2008)
7. Farouki, R.T.: The bernstein polynomial basis: A centennial retrospective. Computer Aided Geometric Design **29**(6), 379–419 (2012)
8. Ferreau, H.J., Kirches, C., Potschka, A., Bock, H.G., Diehl, M.: qpOASES: A parametric active-set algorithm for quadratic programming. Mathematical Programming Computation **6**(4), 327–363 (2014)
9. Hönig, W., Preiss, J.A., Kumar, T.K.S., Sukhatme, G.S., Ayanian, N.: Trajectory planning for quadrotor swarms. IEEE Transactions on Robotics (T-RO) **34**(4), 856–869 (2018)
10. Hornung, A., Wurm, K.M., Bennewitz, M., Stachniss, C., Burgard, W.: OctoMap: An efficient probabilistic 3D mapping framework based on octrees. Autonomous Robots **34**(3), 189–206 (2013). Software available at http://octomap.github.com
11. Mattingley, J., Boyd, S.: CVXGEN: a code generator for embedded convex optimization. Optimization and Engineering **13**(1), 1–27 (2012)
12. Morris, R., Pasareanu, C.S., Luckow, K.S., Malik, W., Ma, H., Kumar, T.K.S., Koenig, S.: Planning, scheduling and monitoring for airport surface operations. In: AAAI Workshop on Planning for Hybrid Systems, *AAAI Workshops*, vol. WS-16-12, pp. 608–614 (2016)
13. Oleynikova, H., Burri, M., Taylor, Z., Nieto, J.I., Siegwart, R., Galceran, E.: Continuous-time trajectory optimization for online UAV replanning. In: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), pp. 5332–5339 (2016)
14. Richter, C., Bry, A., Roy, N.: Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. In: Int. Symposium of Robotic Research (ISRR), pp. 649–666 (2013)
15. Stellato, B., Banjac, G., Goulart, P., Bemporad, A., Boyd, S.: OSQP: An operator splitting solver for quadratic programs. ArXiv e-prints (2018)
16. Usenko, V.C., von Stumberg, L., Pangercic, A., Cremers, D.: Real-time trajectory replanning for MAVs using uniform B-splines and a 3D circular buffer. In: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), pp. 215–222 (2017)
17. Wang, L., Ames, A.D., Egerstedt, M.: Safety barrier certificates for collisions-free multirobot systems. IEEE Transactions on Robotics (T-RO) **33**(3), 661–674 (2017)

18. Wurman, P.R., D'Andrea, R., Mountz, M.: Coordinating hundreds of cooperative, autonomous vehicles in warehouses. AI Magazine **29**(1), 9–20 (2008)
19. Zhou, D., Wang, Z., Bandyopadhyay, S., Schwager, M.: Fast, on-line collision avoidance for dynamic vehicles using buffered voronoi cells. IEEE Robotics and Automation Letters (RA-L) **2**(2), 1047–1054 (2017)