RLSS: real-time, decentralized, cooperative, networkless multi-robot trajectory planning using linear spatial separations

Baskın Şenbaşlar[1] · **Wolfgang Hönig[2]** · **Nora Ayanian[3]**

Abstract

Trajectory planning for multiple robots in shared environments is a challenging problem especially when there is limited communication available or no central entity. In this article, we present Real-time planning using Linear Spatial Separations, or RLSS: a real-time decentralized trajectory planning algorithm for cooperative multi-robot teams in static environments. The algorithm requires relatively few robot capabilities, namely sensing the positions of robots and obstacles without higher-order derivatives and the ability of distinguishing robots from obstacles. There is no communication requirement and the robots' dynamic limits are taken into account. RLSS generates and solves convex quadratic optimization problems that are kinematically feasible and guarantees collision avoidance if the resulting problems are feasible. We demonstrate the algorithm's performance in real-time in simulations and on physical robots. We compare RLSS to two state-of-the-art planners and show empirically that RLSS does avoid deadlocks and collisions in forest-like and maze-like environments, significantly improving prior work, which result in collisions and deadlocks in such environments.

Keywords Trajectory planning for multiple mobile robots · Decentralized robot systems · Collision avoidance · Multi-robot systems

1 Introduction

Effective navigation of multiple robots in cluttered environments is key to emerging industries such as warehouse automation (Wurman et al., 2008), autonomous driving (Furda & Vlacic, 2011), and automated intersection management (Dresner & Stone, 2008). One of the core challenges of navigation systems in such domains is trajectory planning. Planning safe trajectories for multiple robots

is especially challenging when there is no central entity that plans all robots' trajectories a priori or re-plans mid-execution if there is a fault. In some cases, such a central entity is undesirable because of the communication link that must be maintained between each robot and the central entity. If the map is not known a priori, building and relaying the observed map of the environment to the central entity through the communication channel adds further challenges. In some cases, it is impractical to have such a central entity because it cannot react to robot trajectory tracking errors and map updates fast enough due to communication and computation delays. In practice, robots must operate safely even if there is limited communication available. This necessitates the ability for robots to plan in a decentralized fashion, where each robot plans a safe trajectory for itself while operating in environments with other robots and obstacles.

Decentralized algorithms delegate the computation of trajectories: each robot plans for itself and reacts to the environment by itself. In this paper, we introduce RLSS, a real-time decentralized trajectory planning algorithm for multiple robots in a shared environment that requires no communication between robots and requires relatively few sensing capabilities: each robot senses the relative positions

✉ Baskın Şenbaşlar
  baskin.senbaslar@usc.edu

  Wolfgang Hönig
  hoenig@tu-berlin.de

  Nora Ayanian
  nora_ayanian@brown.edu

[1] Department of Computer Science, University of Southern California, 941 Bloom Walk, Los Angeles, CA 90089, USA

[2] Department of Electrical Engineering and Computer Science, Technical University of Berlin, Marchstr. 23, 10587 Berlin, Germany

[3] Department of Computer Science and School of Engineering, Brown University, 115 Waterman Street, Providence, RI 02912, USA
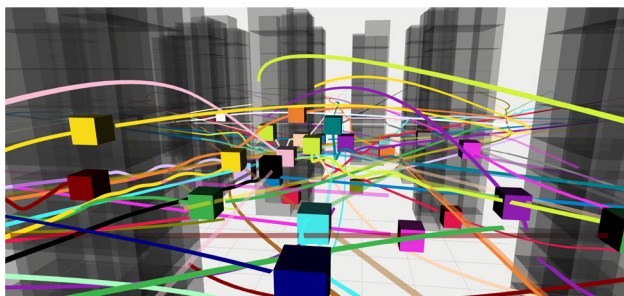
Fig. 1 RLSS runs in real-time in dense environments. Each robot plans a trajectory by itself using only the position information of other robots and obstacles

of other robots and obstacles along with their geometric shapes in the environment, and is able to distinguish robots from obstacles. RLSS requires relatively few robot capabilities than most state-of-the-art decentralized planners used for multi-robot navigation, which typically require communication (Luis et al., 2020; Tordesillas & How, 2021; Wang et al., 2021), higher order derivative estimates (Park & Kim, 2021; Wang et al., 2021), or predicted trajectories of objects (Park & Kim, 2021). However, the ability to distinguish robots from obstacles is not required by some state-of-the-art algorithms (Zhou et al., 2017), which require modelling obstacles as robots. RLSS is cooperative in the sense that we assume each robot stays within its current cell of a tessellation of the space until the next planning iteration. We assume obstacles are static, which is required to guarantee collision avoidance.

RLSS explicitly accounts for the dynamic limits of the robots and enforces safety with hard constraints, reporting when they cannot be enforced, thus guaranteeing collision avoidance when it succeeds. The planning algorithm can be guided with desired trajectories, thus it can be used in conjunction with centralized planners. If no centralized planner or plan is available, RLSS can be used on its own, without any central guidance, by setting the desired trajectories to line segments directly connecting robots' start positions to their goal positions.

There are 4 stages in RLSS.

1. Select a goal position to plan toward on the desired trajectory;
2. Plan a discrete path toward the selected goal position;
3. Formulate and solve a kinematically feasible convex optimization problem to compute a safe smooth trajectory guided by the discrete plan;
4. Check if the trajectory obeys the dynamic limits of the robot, and temporally rescale the trajectory if not.

RLSS works in the receding horizon fashion: it plans a long trajectory, executes it for a short duration, and replans at the next iteration. It utilizes separating hyperplanes, i.e., linear spatial separations, between robot shapes, obstacles, and sweep geometries (the subsets of space swept by robots while traversing straight line segments) to enforce safety during trajectory optimization.

We demonstrate, through simulation and experiments on physical robots, that RLSS works in dense environments in real-time (Fig. 1). We compare our approach to two state-of-the-art receding horizon decentralized multi-robot trajectory planning algorithms in 3D. In the first, introduced by Zhou et al. (2017), robots plan for actions using a model predictive control-style optimization formulation while enforcing that each robot stays inside its buffered Voronoi cell at every iteration. We refer to this method as BVC. The original BVC formulation works only for discrete single-integrator dynamics and environments without obstacles. We extend the BVC formulation to discrete time-invariant linear systems with position outputs and environments with obstacles, and call the extended version eBVC (short for extended BVC). RLSS and eBVC have similar properties: they require no communication between robots, and require position sensing of other objects in the environment. However, unlike RLSS, eBVC does not require that robots are able to distinguish robots from obstacles, as it treats each obstacle as a robot. In the second, introduced by Park and Kim (2021), robots plan for trajectories by utilizing relative safe navigation corridors, which they execute for a short duration, and replan. We refer to this method as RSFC. RSFC requires no communication between robots, and utilizes positions as well as velocities of the objects in the environment, thus requires more sensing capabilities than RLSS. We demonstrate empirically that RLSS results in no deadlocks or collisions in our experiments in forest-like and maze-like environments, while eBVC is prone to deadlocks and RSFC results in collisions in such environments. However, RLSS results in slightly longer navigation durations compared to both eBVC and RSFC.

The contribution of this work can be summarized as follows:

- A carefully designed, numerically stable, and effective real-time decentralized planning algorithm for multiple robots in shared environments with static obstacles with relatively few requirements: no communication between robots, position-only sensing of robots and obstacles, and ability to distinguish robots from obstacles.
- An extension (eBVC) of a baseline planner (BVC) to environments with obstacles and a richer set of dynamics than only single-integrators.
- The first comparison of more than two state-of-the-art communication-free decentralized multi-robot trajectory planning algorithms, namely, RSFC, eBVC, and RLSS, in complicated forest-like and maze-like environments, some of which containing more than 2000 obstacles.

2 Related work

The pipeline of RLSS contains three stages (discrete planning, safe navigation corridor construction, and trajectory optimization) that are employed by several existing single-robot trajectory planning algorithms. Richter et al. (2013) present a single-robot trajectory planning method for aggressive quadrotor flight which utilizes RRT* (Karaman & Frazzoli, 2010) to find a kinematically feasible discrete path and formulates an unconstrained quadratic program over polynomial splines guided by the discrete path. Collisions are checked after optimization, and additional decision points are added and optimization is re-run if there is collision. Chen et al. (2016) present a method that utilizes OcTree representation (Hornung et al., 2013) of the environment during discrete search. They find a discrete path using unoccupied grid cells of the OcTree. Then, they maximally inflate unoccupied grid cells to create a safe navigation corridor that they use as constraints in the subsequent polynomial spline optimization. Liu et al. (2017) uses Jump Point Search (JPS) (Harabor & Grastien, 2011) as the discrete planner, and construct safe navigation corridors that are used as constraints in the optimization stage. Our planning system uses these three stages (discrete planning, safe navigation corridor construction, and corridor-constrained optimization) and extends it to multi-robot scenarios in a decentralized way. We handle robot-robot safety by cooperatively computing a linear partitioning of the environment, and enforcing that each robot stays within its own cell of the partition.

We categorize multi-robot trajectory planning algorithms first on where computation is done, since the location of computation changes the properties of the algorithms. There are two main strategies to solve the multi-robot trajectory planning problem: centralized and decentralized. Centralized algorithms compute trajectories for all robots on a central entity with global information; these trajectories are communicated back to the robots for them to execute. In the decentralized strategy, each robot runs an algorithm onboard to compute its own trajectory; these algorithms may utilize direct communication between robots. Centralized algorithms often provide strong theoretical guarantees on completeness or global optimality because (i) they have the complete knowledge about the environment beforehand, (ii) they generally run on powerful centralized computers, and (iii) there are generally less restrictive time limits on their execution. Decentralized algorithms generally forgo such strong theoretical guarantees in favor of fast computation because (i) they are often used when complete information of the environment is not known beforehand, and (ii) they have to work in real-time on-board.

2.1 Centralized algorithms

We further categorize centralized algorithms into those that do not consider robot dynamics during planning and those that do.

Centralized without dynamics: If planning can be abstracted to agents moving along edges in a graph synchronously, we refer to the multi-agent path finding (MAPF) problem. Some variants of the MAPF problem are NP-Hard to solve optimally (Yu & LaValle, 2013). For NP-Hard variants, there are many optimal (Sharon et al., 2015; Lam et al., 2019), bounded suboptimal (Barer et al., 2014), and suboptimal (Solovey et al., 2013; Ma et al., 2019) algorithms that perform well in some environments. Trajectories generated by planning algorithms that do not model robot dynamics may not be followed perfectly by real robots, resulting in divergence from the plan.

Centralized with dynamics: These algorithms deliver smooth control or output trajectories that are executable by real robots. Trajectories are usually generated by formulating the problem under an optimization framework, and communicated to the robots for them to execute. Tang and Kumar (2016) propose an approach that combines a discrete motion planning algorithm with trajectory optimization that works only in obstacle-free environments. Another approach combines a MAPF solver with trajectory optimization to plan trajectories for hundreds of quadrotors in environments with obstacles, while ensuring that the resulting trajectories are executable by the robots (Hönig et al., 2018). Park et al. (2020) combine a MAPF solver with trajectory optimization and provide executability and feasibility guarantees. Desai and Michael (2020) propose an approach utilizing position-invariant geometric trees to find kinodynamically feasible trajectories.

Centralized algorithms can provide theoretical completeness and global optimality guarantees under the assumptions of perfect prior knowledge and execution of trajectories. However, centralized replanning is required when the environment is discovered during operation or the robots deviate from the planned trajectories. This necessitates continuous communication between the central computer and the individual robots. In some cases, solving the initial problem from scratch may be required. If the communication between the robots and the central computer cannot be maintained, or solving the initial problem from scratch cannot be done in real-time, the robots may collide with each other or obstacles.

2.2 Decentralized algorithms

When the requirements of the centralized strategies cannot be satisfied, decentralization of the trajectory planning among robots is required. In decentralized planning, robots assess the state of the environment and plan for themselves using

their on-board capabilities. We categorize decentralized algorithms into reactive and long horizon algorithms.

Decentralized reactive: In reactive algorithms, each robot computes the next action to execute based on the state of the environment without considering the actions that might follow it. One such approach is Optimal Reciprocal Collision Avoidance (ORCA) (Alonso-Mora et al., 2013). In ORCA, each robot chooses a velocity vector that is as close as possible to a desired velocity vector such that collisions between each pair of robots are provably avoided. It requires robots to (i) be cooperatively executing the same algorithm, (ii) obey single integrator dynamics, and (iii) either sense other robots' velocities or receive them through communication. While ORCA handles dynamic changes in the environment and is completely distributed, it fails to avoid deadlocks in environments with obstacles (Şenbaşlar et al., 2019). Safety barrier certificates guarantee collision avoidance by computing a safe control action that is as close as possible to a desired control action (Wang et al., 2017). Similarly to ORCA, the robots using safety barrier certificates may get stuck in deadlocks. Most learning-based approaches fall into the body of reactive strategies as well. Here, neural networks are trained to compute the next action to execute, given a robot's immediate neighborhood and goal.[1] For instance, PRIMAL (Sartoretti et al., 2019) is a computationally efficient suboptimal online MAPF solver, but it only works on grids, and does not consider robot dynamics. GLAS (Riviere et al., 2020), which considers robot dynamics, can guarantee safety for some robot dynamics by combining the network output with a safety term. It performs better than ORCA in terms of deadlocks, nevertheless results in deadlocks in dense environments. Another approach employs graph neural networks (GNNs) that allow communication between robots to avoid collisions (Li et al., 2020), but results in deadlocks in dense environments similar to GLAS. Batra et al. (2021) propose a method that outputs direct motor commands for a quadrotor using the positions and velocities of objects in the its immediate neighborhood, but results in collisions between robots.

Decentralized long horizon: In long horizon algorithms, robots generate a sequence of actions or long trajectories instead of computing a single action to execute. These approaches employ receding horizon planning: they plan long trajectories, execute them for a short duration, and replan. Zhou et al. (2017) present a model predictive control (MPC) based approach in which robots plan for a sequence of actions while enforcing each robot to stay within its buffered

Voronoi cell. It requires no inter-robot communication and depends on sensing of other robots' positions only. Another MPC-based approach approximates robots' controller behaviors under given desired states as a linear system (Luis et al., 2020). A smooth Bézier curve for each robot is computed by solving an optimization problem, in which samples of desired states are drawn from the curve and fed into the model of the system. The approach requires communication of future states for collision avoidance. Another MPC-based approach generates plans using motion primitives to compute time-optimal trajectories, then trains a neural network to approximate the behavior of the planner, which is used during operation for fast planning (Wang et al., 2021). It requires sensing or communicating the full states of robots. Tordesillas and How (2021) present a method for dynamic obstacle avoidance, handling asynchronous planning between agents in a principled way. However, the approach requires instant communication of planned trajectories between robots to ensure safety. Park and Kim (2021) plan a piecewise Bézier curve by formulating an optimization problem. It utilizes relative safe flight corridors (RSFCs) for collision avoidance and requires no communication between robots, but requires position and velocity sensing. In Peterson et al. (2021)'s approach, robot tasks are defined as time window temporal logic specifications. Robots plan trajectories to achieve their tasks while avoiding each other. The method provably avoids deadlocks, but requires communication of planned trajectories between robots.

RLSS falls into this body of algorithms. It requires sensing only the positions of other robots and obstacles, and does not require any communication between robots. The work of Zhou et al. (2017) (BVC) is the only other approach with these properties. Different from their approach, RLSS uses discrete planning to avoid local minima, plans piecewise Bézier trajectories instead of MPC-style input planning, and does not constrain the full trajectory to be inside a constraining polytope. These result in better deadlock and collision avoidance as we show in the evaluation section.

In terms of inter-robot collision avoidance, RLSS belongs to a family of algorithms that ensure multi-robot collision avoidance without inter-robot communication by using the fact that each robot in the team runs the same algorithm. In these algorithms, robots share the responsibility of collision avoidance by computing feasible action sets that would not result in collisions with others when others compute their own feasible action sets using the same algorithm. Examples include BVC (Zhou et al., 2017), which partitions position space between robots and makes sure that each robot stays within its own cell in the partition, ORCA (Alonso-Mora et al., 2013), which computes feasible velocity sets for robots such that there can be no collisions between robots as long as each robot chooses a velocity command from their corre-

---

[1] Note that some learning-based approaches train neural networks to mimic the behavior of a global long horizon planner. Because of this, even if they output a single action, they abstractly consider a longer horizon evolution of the system. We place these approaches to the reactive algorithms category because they output a single action at each iteration.

sponding sets, and SBC (Wang et al., 2017), which does the same with accelerations.

BVC and RLSS also belong to a family of decentralized multi-robot trajectory planning algorithms that utilize mutually computable separating hyperplanes for inter-robot safety as defined by Şenbaşlar and Sukhatme (2022). These algorithms do not require any inter-robot communication and utilize only position/geometry sensing between robots. The inter-robot safety is enforced by making sure that each pair of robots can compute the *same separating hyperplane* and enforce themselves to be in the different sides of this hyperplane at every planning iteration. Separating hyperplanes that can be mutually computed by robots using only position/geometry sensing and no communication are defined as mutually computable. BVC uses Voronoi hyperplanes and RLSS uses support vector machine hyperplanes, both of which are mutually computable.

RLSS is an extension to our previous conference paper (Şenbaşlar et al., 2019). We extend it conceptually by

- supporting robots with any convex geometry instead of only spherical robots; and
- decreasing the failure rate of the algorithm from 3% in complex environments to 0.01% by providing important modifications to the algorithm. These modifications include (i) changing the discrete planning to best-effort A^* search, thus allowing robots to plan towards their goals even when goals are not currently reachable; (ii) increasing the numerical stability of the algorithm by running discrete planning at each iteration instead of using the trajectory of the previous iteration when it is collision free to define the homotopy class, and adding a *preferred distance cost term* to the optimization problem in order to create a safety zone between objects when possible; and (iii) ensuring the kinematic feasibility of the optimization problem generated at the trajectory optimization stage.

We extend our previous work empirically by

- applying our algorithm to a heterogeneous team of differential drive robots in 2D;
- applying our algorithm to quadrotors in 3D; and
- comparing the performance of our algorithm to stronger baselines.

A preliminary version of RLSS previously appeared in a workshop (Şenbaşlar et al., 2021). In the workshop version, the RLSS optimization stage fails about 3% of the time, and switches to a soft optimization formulation when that happens. In the current version, the optimization rarely (close to 1 in every 10000 iterations in our experiments) fails thanks to the newly added preferred distance cost term, changed goal selection stage, and changed discrete planning stage that

ensures kinematic feasibility of the optimization problem. Hence, there is no longer a different soft optimization formulation. In the workshop version, we compare RLSS against Luis et al. (2020)'s work, which is based on distributed MPC, which requires communication of planned trajectories between robots. Herein, we compare our approach to two planners that do not require communication between robots as this provides a more comparable baseline.

3 Problem statement

We first define the multi-robot trajectory planning problem, then, we indicate the specific case of the problem we solve.

Consider a team of N robots for which trajectories must be computed at time $t = 0$. The team can be heterogeneous, i.e., each robot might be of a different type or shape. Let $\mathcal{R}_i$ be the collision shape function of robot i such that $\mathcal{R}_i(\mathbf{p})$ is the collision shape of robot i located at position $\mathbf{p} \in \mathbb{R}^d$, $d \in \{2, 3\}$, i.e., the subset of space occupied by robot i when placed at position $\mathbf{p}$. We define $\mathcal{R}_i(\mathbf{p}) = \{\mathbf{p} + \mathbf{x} \mid \mathbf{x} \in \mathcal{R}_{i,0}\}$ where $\mathcal{R}_{i,0} \subset \mathbb{R}^d$ is the the space occupied by robot i when placed at the origin. Note that we do not model robot orientation. If a robot can rotate, the collision shape should contain the union of spaces occupied by the robot for each possible orientation at a given position; which is minimally a hypersphere if all orientations are possible.

We assume the robots are differentially flat (Murray et al., 1995), i.e., the robots' states and inputs can be expressed in terms of output trajectories and a finite number of derivatives thereof. Differential flatness is common for many kinds of mobile robots, including differential drive robots (Campion et al., 1996), car-like robots (Murray & Sastry, 1993), omnidirectional robots (Jiang & Song, 2013), and quadrotors (Mellinger & Kumar, 2011). When robots are differentially-flat, their dynamics can be accounted for by (i) imposing C^c continuity on the trajectories for the required c (i.e. continuity up to c^{th} degree of derivative), and (ii) imposing constraints on the maximum k^{th} derivative magnitude of trajectories for any required k. For example, quadrotor states and inputs can be expressed in terms of the output trajectory, and its first (velocity), second (acceleration) and third (jerk) derivatives (Mellinger & Kumar, 2011). Hence, to account for quadrotor dynamics, continuity up to jerk should be enforced by setting $c = 3$, and velocity, acceleration, and jerk of the output trajectory should be limited appropriately by setting upper bounds for $k = 1, 2,$ and 3.

Let $\mathcal{O}(t) = \{\mathcal{Q} \subset \mathbb{R}^d\}$ be the set of obstacles in the environment at time t. The union $\mathcal{Q}(t) = \cup_{\mathcal{Q} \in \mathcal{O}(t)} \mathcal{Q}$ denotes the occupied space at time t. Let $\mathcal{W}_i \subseteq \mathbb{R}^d$ be the workspace that robot i should stay inside of.

Let $\mathbf{d}_i(t) : [0, T_i] \to \mathbb{R}^d$ be the desired trajectory of duration T_i that robot i should follow as closely as possible.

Define $\mathbf{d}_i(t) = \mathbf{d}_i(T_i) \; \forall t \geq T_i$. We do not require that this trajectory is collision-free or even dynamically feasible to track by the robot. If no such desired trajectory is known, it can be initialized, for example, with a straight line to a goal location.

The intent is that each robot i tracks a Euclidean trajectory $\mathbf{f}_i(t) : [0, T] \to \mathbb{R}^d$ such that $\mathbf{f}_i(t)$ is collision-free, executable according to the robot's dynamics, is as close as possible to the desired trajectory $\mathbf{d}_i(t)$, and ends at $\mathbf{d}_i(T_i)$. Here, T is the navigation duration of the team. We generically define the multi-robot trajectory planning problem as the problem of finding trajectories $\mathbf{f}_1, \ldots, \mathbf{f}_N$ and navigation duration T that optimizes the following formulation:

$$\min_{\mathbf{f}_1,\ldots,\mathbf{f}_N,T} \sum_{i=1}^{N} \int_0^T \|\mathbf{f}_i(t) - \mathbf{d}_i(t)\|_2 \qquad dt \; s.t. \quad (1)$$

$$\mathbf{f}_i(t) \in C^{c_i} \qquad\qquad \forall i \in \{1, \ldots, N\} \quad (2)$$

$$\mathbf{f}_i(T) = \mathbf{d}_i(T_i) \qquad\qquad \forall i \in \{1, \ldots, N\} \quad (3)$$

$$\frac{d^c \mathbf{f}_i(0)}{dt^c} = \frac{d^c \mathbf{p}_i^0}{dt^c} \qquad\qquad \forall i \forall c \in \{0, \ldots, c_i\} \quad (4)$$

$$\mathcal{R}_i(\mathbf{f}_i(t)) \cap \mathcal{Q}(t) = \emptyset \qquad\qquad \forall i \forall t \in [0, T] \quad (5)$$

$$\mathcal{R}_i(\mathbf{f}_i(t)) \cap \mathcal{R}_j(\mathbf{f}_j(t)) = \emptyset \qquad \forall j \neq i \forall t \in [0, T] \quad (6)$$

$$\mathcal{R}_i(\mathbf{f}_i(t)) \in \mathcal{W}_i \qquad\qquad \forall i \forall t \in [0, T] \quad (7)$$

$$\max_{t \in [0,T]} \left\| \frac{d^k \mathbf{f}_i(t)}{dt^k} \right\|_2 \leq \gamma_i^k \qquad \forall i \forall k \in \{1, \ldots, K_i\} \quad (8)$$

where c_i is the order of derivative up to which the trajectory of the ith robot must be continuous, $\mathbf{p}_i^0$ is the initial position of robot i (derivatives of which are the initial higher order state components, e.g., velocity, acceleration, etc.), γ_i^k is the maximum k^{th} derivative magnitude that the ith robot can execute, and K_i is the maximum derivative degree that robot i has a derivative magnitude limit on.

The cost (1) of the optimization problem is a metric for the total deviation from the desired trajectories; it is the sum of position distances between the planned and the desired trajectories. (2) enforces that the trajectory of each robot is continuous up to the required degree of derivatives. (3) enforces that planned trajectories end at the endpoints of desired trajectories. (4) enforces that the planned trajectories have the same initial position and higher order derivatives as the initial states of the robots. (5) and (6) enforce robot-obstacle and robot-robot collision avoidance, respectively. (7) enforces that each robot stays within its defined workspace. Lastly, (8) enforces that the dynamic limits of the robot are obeyed.

Note that only constraint (6) stems from multiple robots. However, this seemingly simple constraint couples robots' trajectories both spatially and temporally, making the problem much harder. As discussed in Sect. 2.1, solving the multi-agent path finding problem optimally is NP-Hard even for the discrete case while the discrete single-agent path find-

ing problem can be solved optimally with classical search methods in polynomial time. This curse of dimensionality affects continuous motion planning as well, where even geometric variants are known to be PSPACE-Hard (Hopcroft et al., 1984).

A centralized planner can be used to solve the generic multi-robot trajectory planning problem in one-shot provided that: the current and future obstacle positions are known a priori, computed trajectories can be sent to robots over a communication link, and robots can track these trajectories well enough that they do not violate the spatio-temporal safety of the computed trajectories. In the present work, we aim to approximately solve this problem in the case where obstacles are static, but not necessarily known a priori by a central entity, and there is no communication channel between robots or between robots and a central entity. Each robot plans its own trajectory in real-time. They plan at a high frequency to compensate for trajectory tracking errors.

4 Preliminaries

We now introduce essential mathematical concepts used herein.

4.1 Parametric curves and splines

Trajectories are curves $\mathbf{f} : [0, T] \to \mathbb{R}^d$ that are parametrized by time, with duration T. Mathematically, we choose to use splines, i.e. piecewise polynomials, where each piece is a Bézier curve defined by a set of control points and a duration.

A Bézier curve $\mathbf{f} : [0, T] \to \mathbb{R}^d$ of degree h is defined by $h + 1$ control points $\mathbf{P}_0, \ldots, \mathbf{P}_h \in \mathbb{R}^d$ as follows:

$$\mathbf{f}(t) = \sum_{i=0}^{h} \mathbf{P}_i \binom{h}{i} \left(\frac{t}{T}\right)^i \left(1 - \frac{t}{T}\right)^{(h-i)}.$$

Any Bézier curve $\mathbf{f}$ satisfies $\mathbf{f}(0) = \mathbf{P}_0$ and $\mathbf{f}(T) = \mathbf{P}_h$. Other points guide the curve from $\mathbf{P}_0$ to $\mathbf{P}_h$. Since any Bézier curve $\mathbf{f}$ is a polynomial of degree h, it is smooth, meaning $\mathbf{f} \in C^\infty$.

We choose Bézier curves as pieces because of their *convex hull property*: the curves themselves lie inside the convex hull of their control points, i.e., $\mathbf{f}(t) \in Convex\,Hull\{\mathbf{P}_0, \ldots, \mathbf{P}_h\}$ $\forall t \in [0, T]$ (Farouki, 2012). Using the convex hull property, we can constrain a curve to be inside a convex region by constraining its control points to be inside the same convex region. Tordesillas and How (2020) discuss the conservativeness of the convex hulls of control points of Bézier curves and show that the convex hulls are considerably less conservative than those of B-Splines (Piegl & Tiller, 1995), which are another type of curve with the convex hull property. Yet, they also show that convex hulls of the control points of the Bézier

curves can be considerably more conservative compared to the smallest possible convex sets containing the curves.

4.2 Linear spatial separations: half-spaces, convex polytopes, and support vector machines

A hyperplane $\mathcal{H}$ in $\mathbb{R}^d$ can be defined by a normal vector $\mathcal{H}_\mathbf{n} \in \mathbb{R}^d$ and an offset $\mathcal{H}_a$ as $\mathcal{H} = \{\mathbf{x} \in \mathbb{R}^d \mid \mathcal{H}_\mathbf{n}^\top \mathbf{x} + \mathcal{H}_a = 0\}$. A half-space $\tilde{\mathcal{H}}$ in $\mathbb{R}^d$ is a subset of $\mathbb{R}^d$ that is bounded by a hyperplane such that $\tilde{\mathcal{H}} = \{\mathbf{x} \in \mathbb{R}^d \mid \mathcal{H}_\mathbf{n}^\top \mathbf{x} + \mathcal{H}_a \leq 0\}$. A convex polytope is an intersection of a finite number of half-spaces.

Our approach relies heavily on computing safe convex polytopes and constraining spline pieces to be inside these polytopes. Specifically, we compute hard-margin support vector machine (SVM) (Cortes & Vapnik, 1995) hyperplanes between spaces swept by robots along line segments and the obstacles/robots in the environment, and use these hyperplanes to create safe convex polytopes for robots to navigate in.

5 Assumptions

Here, we list our additional assumptions about the problem formulation defined in Sect. 3 and the capabilities of robots.

We assume that obstacles in the environment are static, i.e., $\mathcal{O}(t) = \mathcal{O} \; \forall t \in [0, \infty]$, and convex.[2] Many existing, efficient, and widely-used mapping tools, including occupancy grids (Homm et al., 2010) and octomaps (Hornung et al., 2013), internally store obstacles as convex shapes; such maps can be updated in real-time using visual or RGBD sensors, and use unions of convex axis-aligned boxes to approximate the obstacles in the environment.

Similarly, we assume that the shapes of the robots are convex.

We assume that the workspace $\mathcal{W}_i \subseteq \mathbb{R}^d$ is a convex polytope. It can be set to a bounding box that defines a room that a ground robot must not leave, a half-space that contains vectors with positive z coordinates so that a quadrotor does not hit the ground or simply be set to $\mathbb{R}^d$. A non-convex workspace $\tilde{\mathcal{W}}_i$ can be modeled by a convex workspace $\mathcal{W}_i$ such that $\tilde{\mathcal{W}}_i \subseteq \mathcal{W}_i$ and a static set of convex obstacles $\tilde{\mathcal{O}}$ that block portions of the convex workspace so that $\tilde{\mathcal{W}}_i = \mathcal{W}_i \setminus \left(\cup_{\mathcal{Q} \in \tilde{\mathcal{O}}} \mathcal{Q} \right)$.

To provide a guarantee of collision avoidance, we assume that robots can *perfectly* sense the relative positions of obsta-

cles and robots as well as their shapes in the environment. The approach we propose does not require sensing of higher order state components (e.g., velocity, acceleration, etc.) or planned/estimated trajectories of objects, as the former is generally a noisy signal which cannot be expected to be sensed perfectly and the latter would require either communication or a potentially noisy trajectory estimation.

RLSS treats robots and obstacles differently. It enforces that each robot stays within a spatial cell that is disjoint from the cells of other robots until the next planning iteration to ensure robot-robot collision avoidance. To compute the spatial cell for each robot, RLSS uses positions and shapes of nearby robots, but not obstacles, in the environment. Therefore, robots must be able to distinguish other robots from obstacles. However, we do not require individual identification of robots.

We assume that the team is cooperative in that each robot runs the same algorithm using the same replanning period.

Lastly, we assume that planning is synchronized between robots, meaning that each robot plans at the same time instant. The synchronization assumption is needed for ensuring robot-robot collision avoidance when planning succeeds.[3]

6 Approach

Under the given assumptions, we solve the generic multi-robot trajectory planning problem approximately using decentralized receding horizon planning. Each robot plans a trajectory, executes it for a short period of time, and repeats this cycle until it reaches its goal. We call each plan-execute cycle an iteration.

We refer to the planning robot as the ego robot, and henceforth drop indices for the ego robot from our notation as the same algorithm is executed by each robot. Workspace $\mathcal{W}$ is the convex polytope in which the ego robot must remain. $\mathcal{R}$ is the collision shape function of the ego robot such that $\mathcal{R}(\mathbf{p}) \subset \mathbb{R}^d$ is the space occupied by the ego robot when it is placed at position $\mathbf{p}$. The ego robot is given a desired trajectory $\mathbf{d}(t) : [0, T] \to \mathbb{R}^d$ that it should follow. The dynamic limits of the robot are modeled using required derivative degree c up to which the trajectory must be continuous, and maximum derivative magnitudes γ^k for required degrees $k \in \{1, \ldots, K\}$.

At every iteration, the ego robot computes a piecewise trajectory $\mathbf{f}(t)$, which is dynamically feasible and safe up to the safety duration s, that it executes for the replanning

---

[2] For the purposes of our algorithm, concave obstacles can be described or approximated by a union of finite number of convex shapes provided that the union contains the original obstacle. Using our algorithm with approximations of concave obstacles results in trajectories that avoid the approximations.

[3] In physical deployments, asynchronous planning can cause collisions between robots when they are in close proximity to each other. To handle collisions stemming from asynchronous planning, robot shapes can be artificially inflated according to the maximum possible planning lag between robots (an empirical value) at the expense of conservativeness.

period $\delta t \leq s$, and fully re-plans, i.e., runs the full planning pipeline, for the next iteration. *The only parameter that is shared by all robots is the replanning period δt.*

Without loss of generality, we assume that navigation begins at $t = 0$, and at the start of planning iteration u, the current timestamp is $\tilde{T} = u\delta t$.

RLSS fits into the planning part of the classical robotics pipeline using perception, planning, and control. The inputs from perception for the ego robot are:

- $\mathcal{S}$: Shapes of other robots.[4] $\mathcal{S}_j \in \mathcal{S}$ where $j \in \{1, \ldots, i-1, i+1, \ldots, N\}$ is the collision shape of robot j sensed by the ego robot such that $\mathcal{S}_j \subseteq \mathbb{R}^d$.
- $\mathcal{O}$: The set of obstacles in the environment, where each obstacle $\mathcal{Q} \in \mathcal{O}$ is a subset of $\mathbb{R}^d$.
- $\mathbf{p}$: Current position of the ego robot, from which derivatives up to required degree of continuity can be computed.[5]

We define $\hat{\mathcal{O}} = \cup_{\mathcal{Q} \in \mathcal{O}} \mathcal{Q}$ as the space occupied by the obstacles, and $\hat{\mathcal{S}} = \cup_{\mathcal{S}' \in \mathcal{S}} \mathcal{S}'$ as the space occupied by the robot shapes. Robots sense the set of obstacles and the set of robot shapes and use those sets in practice. We use spaces occupied by obstacles and robots for brevity in notation.

There are 4 main stages of RLSS: (1) goal selection, (2) discrete planning, (3) trajectory optimization, and (4) temporal rescaling. The planning pipeline is summarized in Fig. 2. At each planning iteration, the ego robot executes the four stages to plan the next trajectory. In the goal selection stage, a goal position and the corresponding timestamp of the goal position on the desired trajectory $\mathbf{d}(t)$ is selected. In the discrete planning stage, a discrete path from robot's current position toward the selected goal position is computed and durations are assigned to each segment. In the trajectory optimization stage, discrete segments are smoothed to a piecewise trajectory. In the temporal rescaling stage, the dynamic limits of the robot are checked and duration rescaling is applied if necessary.

Next, we describe each stage in detail. Each stage has access to the workspace $\mathcal{W}$, the collision shape function $\mathcal{R}$, the desired trajectory $\mathbf{d}(t)$ and its duration T, the maximum derivative magnitudes γ^k, and the derivative degree c up to which the trajectory must be continuous. We call these task inputs. They describe the robot shape, robot dynamics, and

---

[4] Practically, if the ego robot cannot sense a particular robot j because it is not within the sensing range of the ego robot, robot j can be omitted by the ego robot. As long as the sensing range of the ego robot is more than the maximum distance that can be travelled by the ego robot and robot j in duration δt, omitting robot j does not affect the behavior of the algorithm.

[5] In reality, all required derivatives of the position would be estimated using a state estimation method. We use this definition of $\mathbf{p}$ for notational convenience.

🖄 Springer

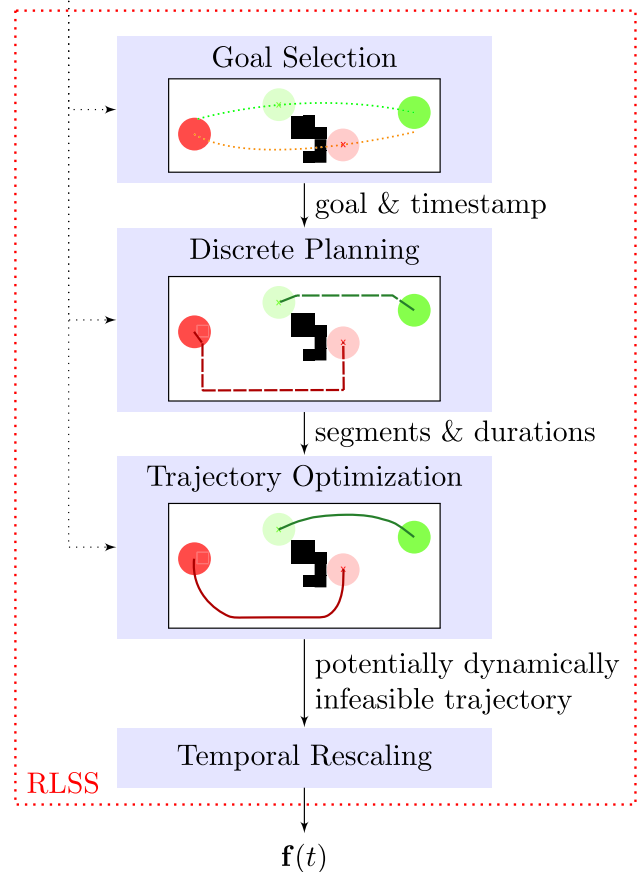$\mathcal{S}, \mathcal{O}, \mathbf{p}$ (inputs from sensing)

$\mathbf{f}(t)$

Fig. 2 RLSS planning pipeline. Based on the sensed robots $\mathcal{S}$, sensed obstacles $\mathcal{O}$, and current position $\mathbf{p}$, the ego robot computes the trajectory $\mathbf{f}(t)$ that is dynamically feasible and safe up to time s

the task at hand; these are not parameters that can be tuned freely and they do not change during navigation.

6.1 Goal selection

At the goal selection stage (Algorithm 1), we find a goal position $\mathbf{g}$ on the desired trajectory $\mathbf{d}(t)$ and a timestamp by which it should be reached. These are required in the subsequent discrete planning stage, which is a goal-oriented search algorithm.

Goal selection has two parameters: the desired planning horizon τ and safety distance D. It uses the robot collision shape function $\mathcal{R}$, desired trajectory $\mathbf{d}(t)$ and its duration T, and workspace $\mathcal{W}$ from the task inputs. The inputs of goal selection are the shapes of other robots $\mathcal{S}$, obstacles in the environment $\mathcal{O}$, current position $\mathbf{p}$, and the current timestamp $\tilde{T}$.

At the goal selection stage, the algorithm finds the timestamp T' that is closest to $\tilde{T} + \tau$ (i.e., the timestamp that is one planning horizon away from the current timestamp)

Algorithm 1 GOAL-SELECTION

Input	: $\mathcal{S}$: Set of robot shapes
Input	: $\mathcal{O}$: Set of obstacles
Input	: $\mathbf{p}$: Current position
Input	: $\tilde{T}$: Current timestamp
TaskInput	: $\mathcal{R}$: Collision shape function
TaskInput	: $\mathbf{d}(t)$, T : Desired trajectory and its duration
TaskInput	: $\mathcal{W}$: Workspace
Parameter	: τ : Desired planning horizon
Parameter	: D : Safety distance
Return	: Goal and timestamp by which it should be reached

1   $T' \leftarrow$ Solve (9) with linear search
2   **if** (9) *is infeasible* **then**
3   **return** $(\mathbf{p}, \tilde{T})$
4   **else**
5     $\mathbf{g} \leftarrow \mathbf{d}(T')$
6   **return** $(\mathbf{g}, T')$
7   **end**

(a) Desired Trajectories   (b) Red Goal Selection

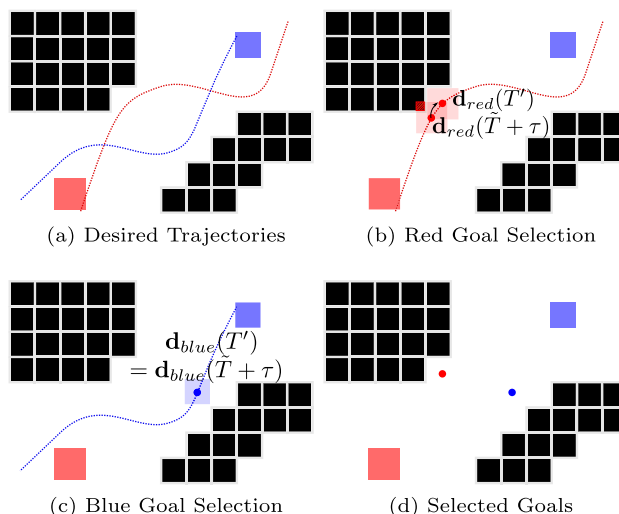(c) Blue Goal Selection   (d) Selected Goals

Fig. 3 Goal Selection. **a** Blue and red squares are robots, while obstacles are black boxes. The desired trajectories $\mathbf{d}_{red}(t)$ and $\mathbf{d}_{blue}(t)$ of each robot are given as dotted lines. Safety distance D is set to 0 for clarity. **b** The desired trajectory of the red robot is not collision-free at timestamp $\tilde{T} + \tau$. It selects its goal timestamp T' (and hence its goal position) by solving (9), which is the closest timestamp to $\tilde{T} + \tau$ when the robot, when placed on the desired trajectory, would be collision free. **c** Since the desired trajectory of blue robot is collision free at timestamp $\tilde{T} + \tau$, it selects its goal timestamp $T' = \tilde{T} + \tau$ after solving (9). **d** Selected goal positions are shown

when the robot, if placed on the desired trajectory at T', is at least safety distance D away from all objects in the environment. We use the safety distance D as a heuristic to choose goal positions that have free volume around them in order not to command robots into tight goal positions. Note that goal selection only chooses a single point on the desired trajectory that satisfies the safety distance; the actual trajectory the robot follows will be planned by the rest of the algorithm. Formally, the problem we solve in the goal selection stage is given as follows:

$$T' = \arg\min_{t} |t - (\tilde{T} + \tau)| \; s.t.$$
$$t \in [0, T] \tag{9}$$
$$\text{min-dist}(\mathcal{R}(\mathbf{d}(t)), \hat{\mathcal{O}} \cup \hat{\mathcal{S}} \cup \partial\mathcal{W}) \geq D$$

where $\partial\mathcal{W}$ is the boundary of workspace $\mathcal{W}$, and min-dist returns the minimum distance between two sets.

We solve (9) using linear search on timestamps starting from $\tilde{T} + \tau$ with small increments and decrements.

Figure 3, demonstrates the goal selection procedure for a particular instance.

If there is no safe point on the desired trajectory, i.e. if the robot is closer than D to objects when it is placed on any point on the desired trajectory, we return the current position and timestamp. This allows us to plan a safe stopping trajectory.

Note that while the selected goal position has free volume around it, it may not be reachable by the ego robot. For example, the goal position may be encircled by obstacles or other robots. Therefore, we use a best-effort search method during discrete planning (as described in Sect. 6.2) that plans a path *towards* the goal position.

The goal and the timestamp are used in the subsequent discrete planning stage as suggestions.

6.2 Discrete planning

Discrete planning (Algorithm 2) performs two main tasks: (i) it finds a collision-free discrete path from the current position $\mathbf{p}$ towards the goal position $\mathbf{g}$, and (ii) it assigns durations to each segment of the discrete path. The discrete path found at the discrete planning stage represents the homotopy class of the final trajectory. Trajectories in the same homotopy class can be smoothly deformed into one another without intersecting obstacles (Bhattacharya et al., 2010). The subsequent trajectory optimization stage computes a smooth trajectory within the homotopy class. The trajectory optimization stage utilizes the discrete path to (i) generate obstacle avoidance constraints and (ii) guide the computed trajectory by adding distance cost terms between the discrete path and the computed trajectory. It uses the durations assigned by the discrete planning stage as the piece durations of the piecewise trajectory it computes.

Finding a discrete path from the start position $\mathbf{p}$ to the goal position $\mathbf{g}$ is done with best-effort A^* search (Line 2, Algorithm 2), which we define as follows. If there are valid paths from $\mathbf{p}$ to $\mathbf{g}$, we find the least-cost path among such paths. If no such path exists, we choose the least-cost path to the position that has the lowest heuristic value (i.e., the position that is heuristically closest to $\mathbf{g}$). This modification of A^* search is done due to the fact that the goal position may

Algorithm 2 DISCRETE-PLANNING

Input	: $\mathbf{g}$: Goal position
Input	: T' : The timestamp that goal position should be (or should have been) reached at
Input	: $\mathcal{S}$: Set of robot shapes
Input	: $\mathcal{O}$: Set of obstacles
Input	: $\mathbf{p}$: Current position
Input	: $\tilde{T}$: Current timestamp
TaskInput	: $\mathcal{R}$: Collision shape function
TaskInput	: $\mathcal{W}$: Workspace
TaskInput	: γ^1 : Maximum velocity the robot can execute
Parameter	: σ : Step size of search grid
Parameter	: s : Duration up to which computed trajectory must be safe. $s \geq \delta t$ must hold.
Return	: Discrete path and duration assignments to segments

1   $\mathcal{F} \leftarrow \mathcal{W} \backslash (\hat{\mathcal{O}} \cup \hat{\mathcal{S}})$
2   $actions \leftarrow$ BEST-EFFORT-$A^*(\mathbf{p}, \mathbf{g}, \mathcal{R}, \mathcal{F}, \sigma)$
3   $\{\mathbf{e}_1, \ldots, \mathbf{e}_L\} \leftarrow$ EXTRACT-SEGMENTS($actions$)
4   Prepend $\mathbf{e}_1$ to $\{\mathbf{e}_1, \ldots, \mathbf{e}_L\}$ so that $\mathbf{e}_0 = \mathbf{e}_1$.
5   $totalLength \leftarrow \sum_{i=1}^{L} \|\mathbf{e}_i - \mathbf{e}_{i-1}\|_2$
6   $fDuration \leftarrow \max \left(T' - \tilde{T}, \frac{totalLength}{\gamma^1} \right)$
7   $T_1 \leftarrow s$
8   **for** $i = 2 \rightarrow L$ **do**
9     $T_i \leftarrow fDuration \frac{\|\mathbf{e}_i - \mathbf{e}_{i-1}\|_2}{totalLength}$
10 **end**
11 **return** $\{\mathbf{e}_0, \ldots, \mathbf{e}_L\}, \{T_1, \ldots, T_L\}$

not always be reachable, since the goal selection stage does not enforce reachability.

The ego robot plans its path in a search grid where the grid has hypercubic cells (i.e. square in 2D, cube in 3D) with edge size σ, which we call the step size of the search. The grid shifts in the environment with the robot in the sense that the robot's current position always coincides with a grid center. Let $\mathcal{F} = \mathcal{W} \backslash (\hat{\mathcal{O}} \cup \hat{\mathcal{S}})$ be the free space within the workspace (Line 1, Algorithm 2). We do not map free space $\mathcal{F}$ to the grid. Instead, we check if the robot shape swept along any segment on the grid is contained in $\mathcal{F}$ or not, to decide if a movement is valid. This allows us to (i) model obstacles and robot shapes independently from the grid's step size σ, and (ii) shift the grid with no additional computation since we do not store occupancy information within the grid.

The states during the search have two components: position $\boldsymbol{\pi}$ and direction $\boldsymbol{\Delta}$. Robots are allowed to move perpendicular or diagonal to the grid. This translates to 8 directions in 2-dimension, 26 directions in 3-dimension. Goal states are states that have position $\mathbf{g}$ and any direction. We model directions using vectors $\boldsymbol{\Delta}$ of d components where each component is in $\{-1, 0, 1\}$. When the robot moves 1 step along direction $\boldsymbol{\Delta}$, its position changes by $\sigma \boldsymbol{\Delta}$. The initial state of the search is the ego robot's current position $\mathbf{p}$ and direction $\mathbf{0}$.

There are 3 actions in the search formulation: ROTATE, FORWARD, and REACHGOAL, summarized in Table 1. ROTATE action has a cost of 1. The cost of the FORWARD

action is the distance travelled divided by the step size (i.e. cells travelled), which is equal to the size $\|\boldsymbol{\Delta}\|_2$ of the direction vector. REACHGOAL has the cost of one rotation plus cells travelled from $\boldsymbol{\pi}$ to goal position $\mathbf{g}$: $1 + \frac{\|\boldsymbol{\pi} - \mathbf{g}\|_2}{\sigma}$. One rotation cost is there because it is almost surely required to do one rotation before going to goal from a cell. ROTATE actions in all directions are always valid whenever the current state is valid. FORWARD and REACHGOAL actions are valid whenever the robot shape $\mathcal{R}$ swept along the movement is contained in free space $\mathcal{F}$.

For any state $(\boldsymbol{\pi}, \boldsymbol{\Delta})$, we use the Euclidean distance from position $\boldsymbol{\pi}$ to the goal position $\mathbf{g}$ divided by step size σ (i.e. cells travelled when $\boldsymbol{\pi}$ is connected to $\mathbf{g}$ with a straight line) as the admissible heuristic.

Lemma 1 *In the action sequence of the resulting plan*

1. *Each ROTATE action must be followed by at least one FORWARD action,*
2. *The first action cannot be a FORWARD action,*
3. *And no action can appear after REACHGOAL action.*

Proof Sketch 1. After each ROTATE action, a FORWARD action must be executed in a least-cost plan because (i) a ROTATE action cannot be the last action since goal states accept any direction and removing any ROTATE action from the end would result in a valid lower cost plan, (ii) the REACHGOAL action cannot appear after ROTATE action because REACHGOAL internally assumes robot rotation and removing the ROTATE action would result in a valid lower cost plan, and iii) there cannot be consecutive ROTATE actions in a least-cost path as each rotation has the cost of 1 and removing consecutive rotations and replacing them with a single rotation would result in a valid lower cost plan.

2. The first action cannot be a FORWARD action since initial direction is set to $\mathbf{0}$ and FORWARD action is available only when $\boldsymbol{\Delta} \neq \mathbf{0}$.

3. No action after a REACHGOAL action can appear in a least cost plan because REACHGOAL connects to the goal position, which is a goal state regardless of the direction. Removing any action after REACHGOAL action would result in a valid lower cost plan. $\square$

By Lemma 1, the action sequence can be described by the following regular expression in POSIX-Extended Regular Expression Syntax:

$$((\text{ROTATE})(\text{FORWARD})^+)^*(\text{REACHGOAL})^{\{0,1\}}$$

We collapse consecutive FORWARD actions in the resulting plan and extract discrete segments (Line 3, Algorithm 2). Each (ROTATE)(FORWARD)$^+$ sequence and REACHGOAL becomes a separate discrete segment. Let $\{\mathbf{e}_1, \ldots, \mathbf{e}_L\}$ be the endpoints of discrete segments. We

Table 1 Discrete search actions and their costs

Action	Description	Cost
ROTATE	Change current direction to a new direction	1
FORWARD	Move forward from the current position $\boldsymbol{\pi}$ along current direction $\boldsymbol{\Delta}$ by the step size σ. It is only available when $\boldsymbol{\Delta} \neq \mathbf{0} \|\boldsymbol{\Delta}\|_2$	
REACHGOAL	Connect the current position $\boldsymbol{\pi}$ to the goal position $\mathbf{g}$ where step size of the grid is σ	$1 + \frac{\|\boldsymbol{\pi} - \mathbf{g}\|_2}{\sigma}$

(a) Goal Positions

(b) Red Discrete Planning

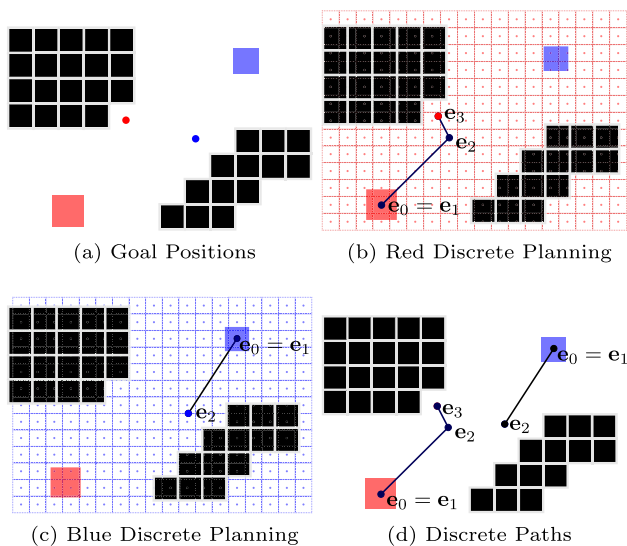(c) Blue Discrete Planning

(d) Discrete Paths

Fig. 4 Discrete Planning. **a** Goal positions of two robots computed at the goal selection stage are given. **b** Red robot plans a discrete path from its current position to its goal position on a search grid that is aligned on its current position. **c** Blue robot plans a discrete path from its current position to its goals position on a search grid that is aligned on its current position. Computed discrete paths are prepended with robot start positions to have 0-length first segments. **d** The resulting discrete paths are given

prepend the first endpoint to the endpoint sequence in order to have a 0-length first segment for reasons that we will explain in Sect. 6.3 (Line 4, Algorithm 2). The resulting L segments are described by $L+1$ endpoints $\{\mathbf{e}_0, \ldots, \mathbf{e}_L\}$ where $\mathbf{e}_0 = \mathbf{e}_1$. Example discrete paths for two robots are shown in Fig. 4.

Next, we assign durations to each segment. The total duration of the segments is computed using the ego robot's maximum velocity γ^1, the timestamp T' that the goal position should be reached by, and the current timestamp $\tilde{T}$. We use $T' - \tilde{T}$ as the desired duration of the plan. However, if $T' - \tilde{T}$ is negative (i.e. $T' < \tilde{T}$, meaning that the goal position should been reached in the past), or $T' - \tilde{T}$ is small such that the robot cannot traverse the discrete segments even with its maximum velocity γ^1, we adjust the desired duration to a feasible one, specifically the total length of segments divided by the maximum velocity (Line 6, Algorithm 2). We distribute the feasible duration to the segments except for the first one, proportional to their lengths (Loop at line 8, Algorithm 2). We set the duration of the first segment, which has

zero length, to the safety duration s (Line 7, Algorithm 2); the reason for this will be explained in Sect. 6.3.

The outputs of discrete planning are segments described by endpoints $\{\mathbf{e}_0, \ldots, \mathbf{e}_L\}$ with assigned durations $\{T_1, \ldots, T_L\}$ that are used in the trajectory optimization stage.

6.3 Trajectory optimization

In the trajectory optimization stage (Algorithm 3), we formulate a convex quadratic optimization problem (QP) to compute a piecewise trajectory $\mathbf{f}(t)$ by smoothing discrete segments. The computed trajectory is collision-free and continuous up to the desired degree of derivative. However, it may be dynamically infeasible (i.e., derivative magnitudes may exceed the maximum allowed derivative magnitudes γ^k); this is resolved during the subsequent temporal rescaling stage.

The decision variables of the optimization problem are the control points of an L-piece spline where each piece is a Bèzier curve. The duration of each piece is assigned during discrete planning. Let $T = \sum_{i=1}^{L} T_i$ denote the total duration of the planned trajectory. The degree of the Bézier curves is tuned with the parameter h. Let $\mathbf{P}_{i,j} \in \mathbb{R}^d$ be the jth control point of the ith piece where $i \in \{1, \ldots, L\}, j \in \{0, \ldots, h\}$. Let $\mathcal{P} = \{\mathbf{P}_{i,j} \mid i \in \{1, \ldots, L\}, j \in \{0, \ldots, h\}\}$ be the set of all control points.

6.3.1 Constraints

There are 4 types of constraints on the trajectory; all are linear in the decision variables $\mathcal{P}$.

1) Workspace constraints: We shift each bounding hyperplane of workspace $\mathcal{W}$ (there are a finite number of such hyperplanes as $\mathcal{W}$ is a convex polytope) to account for the robot's collision shape $\mathcal{R}$, such that when the robot's position is on the safe side of the shifted hyperplane, the entire robot is on the safe side of the original hyperplane (Line 2, Algorithm 3).

Algorithm 3 TRAJECTORY-OPTIMIZATION

Input	: $\{\mathbf{e}_0, \ldots, \mathbf{e}_L\}$: Endpoints of discrete segments
Input	: $\{T_1, \ldots, T_L\}$: Durations of discrete segments
Input	: $\mathcal{S}$: Set of robot shapes
Input	: $\mathcal{O}$: Set of obstacles
Input	: $\mathbf{p}$: Current position
TaskInput	: $\mathcal{R}$: Collision shape function
TaskInput	: $\mathcal{W}$: Workspace
TaskInput	: c : Degree of derivative up to which resulting trajectory must be continuous
Parameter	: h : Degree of Bézier curves
Parameter	: $\tilde{o}$: Obstacle check distance
Parameter	: $\tilde{r}$: Robot check distance
Parameter	: $\tilde{p}$: Preferred distance to objects
Parameter	: α : Preferred distance cost weight
Parameter	: $\boldsymbol{\theta}$: Endpoint cost weights
Parameter	: $\boldsymbol{\lambda}$: Integrated derivative cost weights
Return	: Potentially dynamically infeasible trajectory

1 QP is a quadratic program with variables $\mathcal{P}$
2 $\Upsilon_{\mathcal{W}} \leftarrow$ BUFFER-WORKSPACE$(\mathcal{W}, \mathcal{R})$
3 addWorkspaceConstraints$(QP, \Upsilon_{\mathcal{W}})$
4 $\Upsilon_i \leftarrow \emptyset \ \forall i \in \{1, \ldots, L\}$
5 **for** $\forall \mathcal{S}_j \in \mathcal{S} \ min\text{-}dist(\mathcal{S}_j, \mathcal{R}(\mathbf{p})) \leq \tilde{r}$ **do**
6    $\Upsilon_1 \leftarrow \Upsilon_1 \cup \{$BUFFERED-SVM$(\mathcal{S}_j, \mathcal{R}(\mathbf{p}), \mathcal{R})\}$
7 **end**
8 **for** $i = 1 \rightarrow L$ **do**
9    $\zeta_i \leftarrow$ region swept by $\mathcal{R}$ from $\mathbf{e}_{i-1}$ to $\mathbf{e}_i$
10   **for** $\forall \mathcal{Q} \in \mathcal{O} \ min\text{-}dist(\mathcal{Q}, \zeta_i) \leq \tilde{o}$ **do**
11       $\Upsilon_i \leftarrow \Upsilon_i \cup \{$BUFFERED-SVM$(\mathcal{Q}, \zeta_i, \mathcal{R})\}$
12   **end**
13 **end**
14 addCollAvoidanceConstraints$(QP, \Upsilon_1, \ldots, \Upsilon_L)$
15 addContinuityConstraints$(QP, c, \mathbf{p}, T_1, \ldots, T_L)$
16 addEnergyCostTerm$(QP, \boldsymbol{\lambda}, T_1, \ldots, T_L)$
17 addDeviationCostTerm$(QP, \boldsymbol{\theta}, \mathbf{e}_1, \ldots, \mathbf{e}_L)$
18 $\overline{\Upsilon}_1 \leftarrow$ SHIFT-HYPERPLANES$(\Upsilon_1, \tilde{p})$
19 addPreferredDistanceCostTerm$(QP, \overline{\Upsilon}_1, \alpha)$
20 $\mathbf{f}(t) \leftarrow$ SOLVE-QP(QP)
21 **return** $\mathbf{f}(t)$

Let $\Upsilon_{\mathcal{W}}$ be the set of shifted hyperplanes of $\mathcal{W}$. We constrain each control point of the trajectory to be on the valid sides of the shifted hyperplanes (Line 3, Algorithm 3). Since Bézier curves are contained in the convex hulls of their control points, constraining control points to be in a convex set automatically constrains the Bézier curves to stay within the same convex set.

2) Robot-robot collision avoidance constraints: For robot-robot collision avoidance, recall that each robot replans with the same period δt and planning is synchronized between robots.

At each iteration, the ego robot computes its SVM cell within the SVM tessellation of robots using hard-margin SVMs. SVM tessellation is similar to Voronoi tesselation, the only difference is that pairwise SVM hyperplanes are computed between collision shapes instead of Voronoi hyperplanes. We choose SVM tessellation because (i) hard-margin SVM is convex, hence pairs of robots can compute the same exact hyperplane under the assumption of perfect sensing,

(ii) it allows for a richer set of collision shapes than basic Voronoi cells, which is valid only for hyperspherical objects, and (iii) SVM cells are always convex unlike generalized Voronoi cells.

We buffer the ego robot's SVM cell to account for its collision shape $\mathcal{R}$, and constrain the first piece of the trajectory to stay inside the buffered SVM cell (BSVM) (loop at Line 5, Algorithm 3). Only the first piece of the trajectory is constrained to remain in the buffered SVM cell, since the entire planning pipeline is run after δt, which is smaller than the duration s of the first piece. At that time, planning begins at a new location, generating a new first piece that must remain in the new buffered SVM cell.

Buffering is achieved by changing the offset of the hyperplane. $\mathcal{R}(\mathbf{x})$ is the shape of the robot when placed at $\mathbf{x}$, defined as $\mathcal{R}(\mathbf{x}) = \{\mathbf{x}\} \oplus \mathcal{R}_0$, where $\mathcal{R}_0$ is the shape of the robot when placed the origin and $\oplus$ is the Minkowski sum operator. Given a hyperplane with normal $\mathcal{H}_{\mathbf{n}}$ and offset $\mathcal{H}_a$, we find $\mathcal{H}_{a'}$ that ensures $\mathcal{R}(\mathbf{x})$ is on the negative side of the hyperplane with normal $\mathcal{H}_{\mathbf{n}}$ and offset $\mathcal{H}_a$ whenever $\mathbf{x}$ is on the negative side of the hyperplane with normal $\mathcal{H}_{\mathbf{n}}$ and offset $\mathcal{H}_{a'}$, and vice versa, by setting $\mathcal{H}_{a'} = \mathcal{H}_a + \max_{\mathbf{y} \in \mathcal{R}_0} \mathcal{H}_{\mathbf{n}} \cdot \mathbf{y}$. The following shows that whenever $\mathcal{R}(\mathbf{x})$ is on the negative side of the hyperplane $(\mathcal{H}_{\mathbf{n}}, \mathcal{H}_a)$, $\mathbf{x}$ is on the negative side of the hyperplane $(\mathcal{H}_{\mathbf{n}}, \mathcal{H}_{a'})$. The converse can be shown by following the steps backwards.

$$\forall \mathbf{y} \in \mathcal{R}(\mathbf{x}) \ \mathcal{H}_{\mathbf{n}} \cdot \mathbf{y} + \mathcal{H}_a \leq 0$$
$$\implies \max_{\mathbf{y} \in \mathcal{R}(\mathbf{x})} \mathcal{H}_{\mathbf{n}} \cdot \mathbf{y} + \mathcal{H}_a \leq 0$$
$$\implies \max_{\mathbf{y} \in \{\mathbf{x}\} \oplus \mathcal{R}_0} \mathcal{H}_{\mathbf{n}} \cdot \mathbf{y} + \mathcal{H}_a \leq 0$$
$$\implies \max_{\mathbf{y} \in \mathcal{R}_0} \mathcal{H}_{\mathbf{n}} \cdot (\mathbf{y} + \mathbf{x}) + \mathcal{H}_a \leq 0$$
$$\implies \mathcal{H}_{\mathbf{n}} \cdot \mathbf{x} + \mathcal{H}_a + \max_{\mathbf{y} \in \mathcal{R}_0} \mathcal{H}_{\mathbf{n}} \cdot \mathbf{y} \leq 0$$

Since the duration of the first piece was set to the safety duration $s \geq \delta t$ in discrete planning, the robot stays within its buffered SVM cell for at least δt. Moreover, since planning is synchronized across all robots, the pairwise SVM hyperplanes they compute will match, thus the buffered SVM cells of robots are disjoint. This ensures robot-robot collision avoidance until the next planning iteration.

Computed SVMs and BSVMs are shown in Fig. 5b for a two robot case.

To ensure that the number of constraints of the optimization problem does not grow indefinitely, SVM hyperplanes are only computed against those robots that are at most $\tilde{r}$ away from the ego robot. This does not result in unsafe trajectories so long as $\tilde{r}$ is more than the total maximum distance that can be traversed by two robots while following the first pieces of their trajectories, for which an upper bound is $\max_{i,j \in \{1,\ldots,N\}} (\gamma_i^1 s_i + \gamma_j^1 s_j)$, where s_i and s_j are
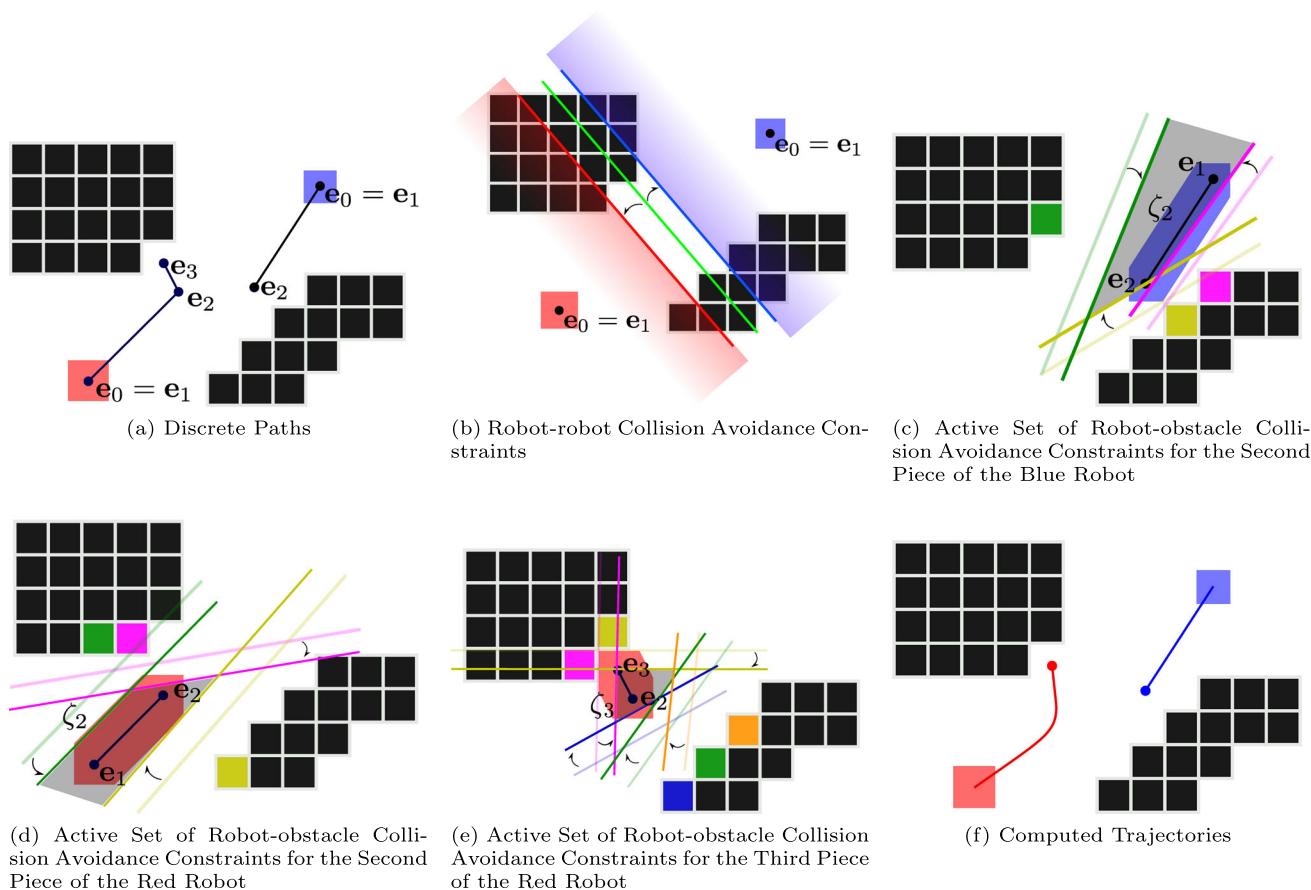
(a) Discrete Paths

(b) Robot-robot Collision Avoidance Constraints

(c) Active Set of Robot-obstacle Collision Avoidance Constraints for the Second Piece of the Blue Robot

(d) Active Set of Robot-obstacle Collision Avoidance Constraints for the Second Piece of the Red Robot

(e) Active Set of Robot-obstacle Collision Avoidance Constraints for the Third Piece of the Red Robot

(f) Computed Trajectories

Fig. 5 Trajectory Optimization. **a** Discrete segments of two robots computed at the discrete planning stage. **b** Robot-robot collision avoidance constraints are computed using BSVMs. The green hyperplane is the SVM hyperplane between two robots. Each robot shifts the SVM hyperplane to account for its geometry, and constrains the first piece of the trajectory with the resulting BSVM. **c**–**e** Active set of robot-obstacle collision avoidance constraints for three different pieces (one belonging to the blue robot, two belonging to the red robot). In each figure, the region swept by the robot while traversing the segment is shown in the robot's color. SVM hyperplanes between the swept region and the obstacles are given as light-colored lines. SVM hyperplanes are buffered to account for the robot's collision shape and shown as dark-colored lines (BSVMs). The shift operations are shown as arrows. Obstacles and constraints generated from them are colored using the same colors. For each piece, the feasible region that is induced by the robot-obstacle collision avoidance constraints is colored in gray. **f** Trajectories computed by the trajectory optimization stage are given

the durations of the first pieces of the trajectories of robots i and j respectively.

3) Robot-obstacle collision avoidance constraints: Buffered SVM hyperplanes are used for robot-obstacle collision avoidance as well. Let $\zeta_i \subset \mathbb{R}^d$ be the region swept by the ego robot while traversing the ith segment from $\mathbf{e}_{i-1}$ to $\mathbf{e}_i$. We compute the SVM hyperplane between ζ_i and each object in $\mathcal{O}$, buffer it as explained before to account for robot's collision shape, and constrain the ith piece by the resulting buffered SVM (Loop at line 8, Algorithm 3). This ensures that trajectory pieces do not cause collisions with obstacles.

The use of SVMs for collision avoidance against obstacles is a choice of convenience, as we already use them for robot-robot collision avoidance. For robot-obstacle collision avoidance, one can use any separating hyperplane between ζ_i and objects in $\mathcal{O}$, while in the case of robot-robot collision avoidance, pairs of robots must compute matching hyperplanes using the same algorithm.

Elements of the active set of SVM and BSVM hyperplanes for robot-obstacle collision avoidance are shown in an example scenario in Fig. 5c–e for three different pieces.

Similar to robot-robot collision avoidance, to ensure that the number of constraints of the optimization problem does not grow indefinitely, we only compute SVM hyperplanes between ζ_i and obstacles that are not more than $\tilde{o}$ away from ζ_i.

Let Υ_i be the set of buffered SVM hyperplanes that constrain the ith piece $\forall i \in \{1, \ldots, L\}$. Υ_1 contains both robot-robot and robot-obstacle collision avoidance hyperplanes while $\Upsilon_j \forall j \in \{2, \ldots, L\}$ contain only robot-obstacle collision avoidance hyperplanes. This is because the first piece is the only piece that we constrain with robot-robot

collision avoidance hyperplanes because it is the only piece that will be executed until the next planning iteration.

4) Continuity constraints: We add two types of continuity constraints: (i) continuity constraints between planning iterations, and (ii) continuity constraints between trajectory pieces (Line 15, Algorithm 3).

To enforce continuity between planning iterations, we add constraints that enforce

$$\frac{d^j \mathbf{f}(0)}{dt^j} = \frac{d^j \mathbf{p}}{dt^j} \; \forall j \in \{0, \ldots, c\}$$

where c is the task input denoting the continuity degree up to which the resulting trajectory must be continuous and $\mathbf{p}$ is the current position.

To enforce continuity between trajectory pieces, we add constraints that enforce

$$\frac{d^j \mathbf{f}_i(T_i)}{dt^j} = \frac{d^j \mathbf{f}_{i+1}(0)}{dt^j}$$
$$\forall i \in \{1, \ldots, L-1\} \; \forall j \in \{0, \ldots, c\}$$

where $\mathbf{f}_i(t)$ is the ith piece of the trajectory.

Remark 1 discusses that the SVM problems generated during trajectory optimization are feasible, i.e., the trajectory optimization stage will always succeed constructing the QP.

Remark 1 All SVM problems generated for robot-robot and robot-obstacle collision avoidance from the discrete path outputted by the discrete planning stage are feasible.

Reasoning. The discrete planning stage outputs a discrete path such that a robot with collision shape $\mathcal{R}$ following the path does not collide with any obstacles in $\mathcal{O}$ or any other robots in $\mathcal{S}$. It also ensures that $\mathbf{p} = \mathbf{e}_0$ since the search starts from the robot's current position $\mathbf{p}$. Hence, $\mathcal{R}(\mathbf{p})$ does not intersect with any $\mathcal{S}_j \in \mathcal{S}$. Since each robot is assumed to be convex, there exists at least one hyperplane that separates $\mathcal{R}(\mathbf{p})$ from $\mathcal{S}_j$ for each j by the separating hyperplane theorem. SVM is an optimal separating hyperplane according to a cost function. Therefore, SVM problems between $\mathcal{R}(\mathbf{p})$ and $\mathcal{S}_j \in \mathcal{S}$ for robot-robot collision avoidance are feasible.

Since the robot moving along the discrete segments is collision free, ζ_i is collision free for all i. Also, since it is a sweep of a convex shape along a line segment, ζ_i is convex as shown in Lemma 2 in Appendix A. Similar to the previous argument, since ζ_i is collision-free and convex and all obstacles in the environment are convex, each SVM problem between ζ_i and $\mathcal{Q} \in \mathcal{O}$ for robot-obstacle collision avoidance is feasible by the separating hyperplane theorem.

Workspace, robot-robot collision avoidance, robot-obstacle collision avoidance, and position continuity constraints are kinematic constraints. Higher-order continuity constraints are dynamic constraints. Remark 2 discusses the ensured feasibility of the kinematic constraints. The feasibility of the dynamic constraints cannot be ensured for arbitrary degrees of continuity.

Remark 2 Kinematic constraints of the optimization problem generated from a discrete path output from the discrete planning stage are feasible for the same path when the degree of Bézier curves $h \geq 1$.

Reasoning. Any Bézier curve with degree $h \geq 1$ can represent a discrete segment by setting half of the points to the start of the segment and other half to the end of the segment. Hence, we will only show that a discrete path output from discrete planning stage by itself satisfies the kinematic constraints generated. Remember that discrete path output from the discrete planning stage has the property $\mathbf{p} = \mathbf{e}_0 = \mathbf{e}_1$.

Each robot-robot SVM problem is feasible (see Remark 1). Let $\mathcal{H}_\mathbf{n}$ be the normal and $\mathcal{H}_a$ be the offset of any of the robot-robot SVMs. It is shifted by setting the offset to $\mathcal{H}_{a'} = \mathcal{H}_a + \max_{\mathbf{y} \in \mathcal{R}_0} \mathcal{H}_\mathbf{n} \cdot \mathbf{y}$. The point $\mathbf{p}$ satisfies $\mathcal{H}_\mathbf{n} \cdot \mathbf{p} + \mathcal{H}_{a'} \leq 0$ because $\mathcal{R}(\mathbf{p})$ is on the negative side of the SVM. The robot-robot BSVM hyperplanes are used to constrain the first piece of the trajectory, and setting the first piece as a 0-length segment with $\mathbf{p} = \mathbf{e}_0 = \mathbf{e}_1$ satisfies the robot-robot collision avoidance constraints.

Each robot-obstacle SVM problem is feasible (see Remark 1). Let $\mathcal{H}_\mathbf{n}$ be the normal and $\mathcal{H}_a$ be the offset of any of the robot-obstacle SVMs that is between ζ_i and an obstacle. It is shifted by setting the offset to $\mathcal{H}_{a'} = \mathcal{H}_a + \max_{\mathbf{y} \in \mathcal{R}_0} \mathcal{H}_\mathbf{n} \cdot \mathbf{y}$. All points on the line segment $\mathbf{p}_i(t) = \mathbf{e}_{i-1} + t(\mathbf{e}_i - \mathbf{e}_{i-1})$, $t \in [0, 1]$ from $\mathbf{e}_{i-1}$ to $\mathbf{e}_i$ satisfy the BSVM constraint because $\mathcal{R}(\mathbf{p}_i(t))$ is on the negative side of the SVM hyperplane $\forall t \in [0, 1]$. Since each SVM hyperplane between ζ_i and obstacles is only used to constrain the ith piece, constraints generated by it are feasible for the segment from $\mathbf{e}_{i-1}$ to $\mathbf{e}_i$.

The feasibility of the workspace constraints are trivial since the robot moving along discrete segments is contained in the workspace, and we shift bounding hyperplanes of the workspace in the same way as SVM hyperplanes. Hence, the discrete path satisfies the workspace constraints.

Initial point position continuity of the robot is satisfied by the given discrete segments since $\mathbf{p} = \mathbf{e}_0$. Position continuity between segments are trivially satisfied by the given discrete segments, since the discrete path is position continuous by its definition. □

6.3.2 Cost function

The cost function of the optimization problem has 3 terms: (i) energy usage, (ii) deviation from the discrete path, and (iii) preferred distance to objects.

We use the sum of integrated squared derivative magnitudes as a metric for energy usage (Line 16, Algortihm 3), similar to the works of Richter et al. (2013) and Hönig et al.

(2018). Parameters $\boldsymbol{\lambda} = \{\lambda_j\}$ are the weights for integrated squared derivative magnitudes, where λ_j is the weight for j^{th} degree of derivative. The energy term $\mathcal{J}_{energy}(\mathcal{P})$ is given as

$$\mathcal{J}_{energy}(\mathcal{P}) = \sum_{\lambda_j \in \boldsymbol{\lambda}} \lambda_j \int_0^T \left\| \frac{d^j \mathbf{f}(t)}{dt^j} \right\|_2^2 dt.$$

We use squared Euclidean distances between trajectory piece endpoints and discrete segment endpoints as a metric for deviation from the discrete path (Line 17, Algorithm 3). Remember that each Bézier curve piece i ends at its last control point $\mathbf{P}_{i,h}$. Parameters $\boldsymbol{\theta} = \{\theta_i\} \in \mathbb{R}^L$ are the weights for each segment endpoint. The deviation term $\mathcal{J}_{dev}(\mathcal{P})$ is given as

$$\mathcal{J}_{dev}(\mathcal{P}) = \sum_{i \in \{1,\dots,L\}} \theta_i \left\| \mathbf{P}_{i,h} - \mathbf{e}_i \right\|_2^2.$$

The last term of the cost function models the preferred distance to objects. We use this term to discourage robots from getting closer to other objects in the environment; this increases the numerical stability of the algorithm by driving robots away from tight states.

We shift each hyperplane in Υ_1 (i.e., buffered SVM hyperplanes constraining the first piece) by the preferred distance to objects, $\tilde{p}$, to create the preferred hyperplanes $\overline{\Upsilon}_1$ (Line 18, Algorithm 3). Since each robot replans with the period δt, we add a cost term that drives the robot closer to the preferred hyperplanes at the replanning period (Line 19, Algorithm 3). We take the sum of squared distances between $\mathbf{f}(\delta t)$ and hyperplanes in $\overline{\Upsilon}_1$:

$$\mathcal{J}_{pref}(\mathcal{P}) = \alpha \sum_{\mathcal{H} \in \overline{\Upsilon}_1} \left\| \mathcal{H}_\mathbf{n}^\top \mathbf{f}(\delta t) + \mathcal{H}_a \right\|_2^2$$

where $\mathcal{H}_\mathbf{n}$ is the normal and $\mathcal{H}_a$ is the offset of hyperplane $\mathcal{H}$, and α is the weight of $\mathcal{J}_{pref}$ term. Notice that this term is defined over the control points of first piece since $\delta t \leq s = T_1$, supporting the utilization of Υ_1 only.

The overall trajectory optimization problem is:

$$\min_{\mathcal{P}} \mathcal{J}_{energy}(\mathcal{P}) + \mathcal{J}_{dev}(\mathcal{P}) + \mathcal{J}_{pref}(\mathcal{P}) \text{ s.t.}$$

$$\mathcal{H}_\mathbf{n}^\top \mathbf{P}_{i,j} + \mathcal{H}_a \leq 0 \quad \forall i \in \{1, \dots, L\}$$
$$\forall j \in \{0, \dots, h\}$$
$$\forall \mathcal{H} \in \Upsilon_i \cup \Upsilon_\mathcal{W}$$
$$\frac{d^j \mathbf{f}(0)}{dt^j} = \frac{d^j \mathbf{p}}{dt^j} \quad \forall j \in \{0, \dots, c\}$$
$$\frac{d^j \mathbf{f}_i(T_i)}{dt^j} = \frac{d^j \mathbf{f}_{i+1}(0)}{dt^j} \quad \forall i \in \{1, \dots, L-1\}$$
$$\forall j \in \{0, \dots, c\}.$$

Notice that we formulate continuity and the safety of the trajectory using hard constraints. This ensures that the resulting trajectory is kinematically safe and continuous up to degree c if the optimization succeeds for all robots and planning is synchronized.

6.4 Temporal rescaling

At the temporal rescaling stage, we check whether the dynamic limits of the robot are violated. If the resulting trajectory is valid, i.e. derivative magnitudes of the trajectory are less than or equal to the maximum derivative magnitudes $\gamma^k \; \forall k \in \{1, \dots, K\}$, the trajectory is returned as the output of RLSS. If not, temporal rescaling is applied to the trajectory similar to Hönig et al. (2018) and Park et al. (2020) by increasing the durations of the pieces so that the trajectory is valid. We scale the durations of pieces by multiplying them with a constant parameter greater than 1 until the dynamic limits are obeyed.

We choose to enforce dynamic feasibility outside of the trajectory optimization problem as a post processing step because (i) the output of the trajectory optimization stage is often dynamically feasible, hence rescaling is rarely needed, and (ii) adding piece durations as variables to the optimization problem would make it a non quadratic program, potentially decreasing performance.

7 Evaluation

Here, using synchronized simulations, we first evaluate our algorithm's performance when different parameters are used in Sect. 7.1. Second, we conduct an ablation study to show the effects of two important steps, namely the prepend operation of the discrete planning stage and the preferred distance cost term of the trajectory optimization stage, to the performance of the algorithm in Sect. 7.2. Third, we compare our algorithm to two state-of-the-art baseline planners in Sect. 7.3. Finally, we show our algorithm's applicability to real robots by using it on quadrotors and differential drive robots in Sect. 7.4.

We conduct our simulations on a laptop computer with Intel i7-8565U @ $1.80GHz$ running Ubuntu 20.04. We implement our algorithm for a single core because of implementation simplicity and fairness to the baseline planners, which are not parallelized. The memory usage of each simulation is $30MB$ on average for our algorithm.

In synchronized simulations, we compute trajectories for each robot using the same snapshot of the environment, move robots perfectly according to computed trajectories for the re-planning period, and replan. The effects of planning iterations taking longer than the re-planning period are not modeled in the synchronized simulations. We show that all algorithms (RLSS and the baselines) can work in $1Hz - 10Hz$ on

the hardware we use, and assert that more powerful computers can be used to shorten planning times. In addition, parallelization of the A* search on GPUs is possible. For example, Zhou and Zeng (2015) show the possibility of 6–7 x speedup in A* search for pathfinding problems on GPUs. Moreover, parallelization of quadratic program solving is possible through (i) running multiple competing solvers in multiple cores and returning the answer from the first one that solves the problem, or (ii) parallelizing individual solvers. For instance, IBM ILOG CPLEX Optimizer[6] and Gurobi Optimizer[7] support running multiple competing solvers concurrently. IBM ILOG CPLEX Optimizer supports a parallel barrier optimizer,[8] which parallelizes a barrier algorithm for solving QPs. Gondzio and Grothey (2009) propose a parallel interior point method solver that exploits nested block structure of problems. Performance improvements of these approaches are problem dependent, and we do not investigate how much the performance of our trajectory optimization stage could be improved with such methods.

In all experiments, 32 robots are placed in a circle formation of radius $20m$ in 3D, and the task is to swap all robots to the antipodal points on the circle. The workspace $\mathcal{W}$ is set to an axis aligned bounding box from $\begin{bmatrix} -25m & -25m & 0m \end{bmatrix}^\top$ to $\begin{bmatrix} 25m & 25m & 5m \end{bmatrix}^\top$. Robot collision shapes are modeled as axis aligned cubes with $0.2m$ edge lengths. The desired planning horizon τ is set to $5s$. The safety distance D of goal selection is set to $0.2m$. Robots have velocity limit $\gamma^1 = 3.67\frac{m}{s}$, and acceleration limit $\gamma^2 = 4.88\frac{m}{s^2}$, which are chosen arbitrarily. The safety duration s is set to $0.11s$ and re-planning period δt is set to $0.1s$. We set integrated derivative cost weights $\lambda_1 = 2.0$ for velocity and $\lambda_2 = 2.8$ for acceleration. We set endpoint cost weights to $\theta_1 = 0$, $\theta_2 = 150$, $\theta_3 = 240$, $\theta_i = 300 \ \forall i \geq 4$. Setting $\theta_1 = 0$ allows the optimization to stretch the first 0-length segment freely, and setting other θs incrementally increases the importance of tail segments. We set the preferred distance to objects to $\tilde{p} = 0.6m$ and the preferred distance cost weight $\alpha = 0.3$. We use the OcTree data structure from the octomap library (Hornung et al., 2013) to represent the environment. Each leaf-level occupied axis aligned box of the OcTree is used as a separate obstacle in all algorithms. OcTree allows fast axis aligned box queries which return obstacles that intersects with a given axis aligned box, an operation we use extensively in our implementation. For example, we use axis aligned box queries in discrete planning as broadphase collision checkers to find the obstacles that are close to the

volume swept by the ego robot while traversing a given segment, and check collisions between the robot traversing the segment and only the obstacles returned from the query. Also, to generate robot-obstacle collision avoidance constraints, we execute axis aligned box queries around the segments to retrieve nearby obstacles. We generate BSVM constraints only against nearby obstacles that are no more than $\tilde{o}$ away from the robot traversing the segment. Other popular approaches for environment representations include (i) Euclidean signed distance fields (ESDFs) (Oleynikova et al., 2017), which support fast distance queries to nearest obstacles, (ii) 3D circular buffers (Usenko et al., 2017), which aim to limit memory usage of maps and supports fast occupancy checks, and (iii) Gaussian mixture models (O'Meadhra et al., 2019), which continuously represent occupancy instead of discretizing the environment as the former approaches do. None of these representations are as suitable as OcTrees for RLSS since they do not allow fast querying of obstacles in the vicinity of segments. We use the IBM ILOG CPLEX Optimizer[9] to solve our optimization problems. In all experiments, robots that are closer than $0.25m$ to their goal positions are considered as goal reaching robots. If a robot that has not reached its goal does not change its position more than $1cm$ in the last $1s$ of the simulation, it is considered as a deadlocked robot. At any point of the simulation, if each robot is either at the goal or deadlocked, we conclude the simulation.

7.1 Effects of selected parameters

We evaluate the performance of our algorithm when 4 important parameters are changed: step size σ of the search grid, degree h of Bézier curves, obstacle check distance $\tilde{o}$, and robot check distance $\tilde{r}$. Step size σ of the search grid is the parameter that affects discrete planning performance most because it determines the amount of movement at each step during the A* search. The degree h of Bézier curves is important in trajectory optimization because it determines the number of decision variables. The obstacle check distance $\tilde{o}$ and robot check distance $\tilde{r}$ determine the number of collision avoidance constraints in the optimization problem.

In all of the parameter evaluations, we use a random 3D forest environment with OcTree resolution $0.5m$ in which 10% of the environment is occupied. There are 2332 leaf-level boxes in OcTree, translating to 2332 obstacles in total. We set the desired trajectory of each robot to the straight line segment that connects the robot's start and goal positions. We set the duration of the segment to the length of the line segment divided by the maximum velocity of the robot. We enforce continuity up to velocity, hence set $c = 1$.

In our experiments, our algorithm does not result in any collisions or deadlocks.

[6] https://www.ibm.com/docs/en/icos/22.1.0?topic=optimizers-concurrent-optimizer-in-parallel.

[7] https://www.gurobi.com/documentation/9.5/refman/concurrent_environments.html.

[8] https://www.ibm.com/docs/en/icos/22.1.0?topic=optimizers-using-parallel-barrier-optimizer.

[9] https://www.ibm.com/analytics/cplex-optimizer.

We report average computation time per iteration (Fig. 6) and average navigation duration of robots (Fig. 7). Average navigation duration is the summed total navigation time for all robots divided by the number of robots.

7.1.1 Step size σ of discrete search

We evaluate our algorithm's performance when step size σ is changed. We set σ to values between $0.25m$ to $1.5m$ with $0.25m$ increments. We set obstacle check distance $\tilde{o} = 1.0m$, robot check distance $\tilde{r} = 2.0m$, and degree of Bézier curves $h = 12$ in all cases, which are determined by premiliminary experiments and the results of the experiments in Sects. 7.1.2, 7.1.3, and 7.1.4. The results are summarized in Figs. 6a and 7a.

As the step size gets smaller, discrete search takes more time; but the algorithm can still work in about 2Hz even when $\sigma = 0.25m$. The average navigation duration of robots are close to $22.5s$ in each case, suggesting the robustness of the algorithm to the changes in this parameter. At $\sigma \geq 0.75$, the time discrete planning takes is less than 6% of the time trajectory optimization takes.

We also run the algorithm with $\sigma = 3m$, $\sigma = 6m$, and $\sigma = 12m$, which decreases the flexibility of the discrete search considerably. In all of those cases, discrete search results in fluctuations, and some robots get stuck in livelocks, in which they move between same set of positions without reaching to their goal positions.

7.1.2 Degree h of Bézier curves

Next, we evaluate our algorithm's performance when the degree h of Bézier curves is changed. We set h to values in $\{5, \ldots, 12\}$. We set step size $\sigma = 0.77m$, obstacle check distance $\tilde{o} = 1.0m$, and robot check distance $\tilde{r} = 2.0m$, which are determined by premiliminary experiments and the results of the experiments in Sects. 7.1.1, 7.1.3, and 7.1.4. The results are summarized in Figs. 6b and 7b.

Even if the degree of the Bézier curves determine the number of decision variables of the trajectory optimization, the computation time increase of the trajectory optimization stage is not more than 10% between degree 5 Bézier curves and degree 12 Bézier curves. Also, average navigation duration of robots are close to $22.5m$ in each case, suggesting the robustness of the algorithm to the changes in this parameter as well.

7.1.3 Obstacle check distance $\tilde{o}$

Next, we evaluate our algorithm's performance when the obstacle check distance $\tilde{o}$ is changed. We set $\tilde{o}$ to values between $0.5m$ and $3m$ with $0.5m$ increments. Since the replanning period $\delta t = 0.1s$, and maximum velocity

$\gamma^1 = 3.67\frac{m}{s}$, the maximum amount of distance that can be traversed by a robot until the next planning iteration is $0.367m$. The obstacle check distance must be more than this value for safety. We set step size $\sigma = 0.77m$, robot check distance $\tilde{r} = 2.0m$, and degree of Bézier curves $h = 12$, which are determined by premiliminary experiments and the results of the experiments in Sects. 7.1.1, 7.1.2, and 7.1.4. The results are summarized in Figs. 6c and 7c.

The obstacle check distance is the most important parameter that determines the speed of trajectory optimization, and hence the planning pipeline. As $\tilde{o}$ increases, the number of SVM computations and the number of constraints in the optimization problem increases, which results in increased computation time. Average navigation durations of the robots are close to $22.5m$ in all cases, suggesting the robustness of the algorithm to this parameter. We explain the reason of this robustness as follows. All obstacles are considered during discrete search and $\tilde{o}$ determines the obstacles that are considered during trajectory optimization. Therefore, the path suggested by the discrete search is already very good, and obstacle avoidance behavior of trajectory optimization is only important when the discrete path is close to obstacles. In those cases, all obstacle check distances capture the obstacles in the vicinity of the path. Therefore, the quality of the planned trajectories does not increase as $\tilde{o}$ increases.

7.1.4 Robot check distance $\tilde{r}$

Last, we evaluate our algorithm's performance when the robot check distance $\tilde{r}$ is changed. We set $\tilde{r}$ to values between $1m$ and $3.5m$ with $0.5m$ increments. Robot check distance must be at least twice the amount of distance that can be traversed by the robot in one planning iteration, i.e. $0.734m$, because two robots may be travelling towards each other with their maximum speed in the worst case. We set step size $\sigma = 0.77m$, obstacle check distance $\tilde{o} = 1.0m$, and degree of Bézier curves $h = 12$, which are determined by premiliminary experiments and the results of the experiments in Sects. 7.1.1, 7.1.2, and 7.1.3.

The speed of the algorithm is not affected by the robot check distance considerably, because there are 32 robots in the environment, and from the perspective of the ego robot, there are at most 31 constraints generated from other robots. Since there are more than 2000 obstacles in the environment, effects of the obstacle check distance are more drastic than robot check distance. Similar to other cases, average navigation duration of the robots is not affected by the choice of $\tilde{r}$ because constraints generated by the robots far away do not actually constrain the trajectory of the ego robot since the ego robot cannot move faster than its maximum speed and hence the first piece of the trajectory is never affected by those constraints.
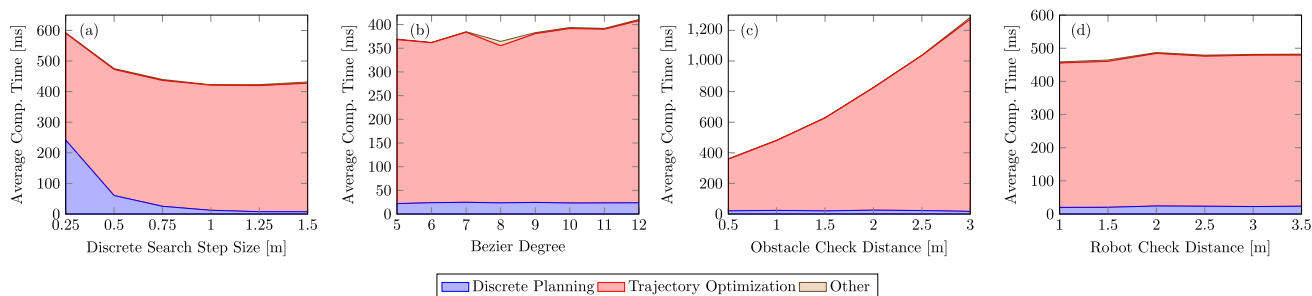
Fig. 6 Average computation time per stage are given when different parameters are used. Goal selection and temporal rescaling steps are given as "other" since they take a small amount time compared to other two stages. All experiments are done in a 3D random forest environment with 10% occupancy
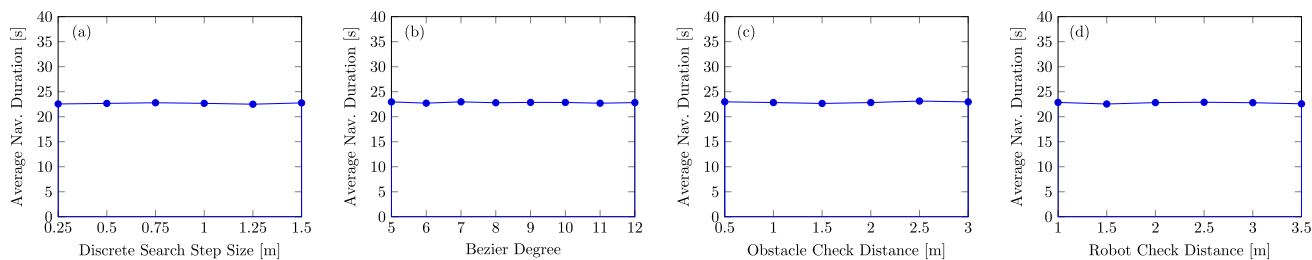

Fig. 7 Average navigation duration of robots from their start positions to their goal positions are given when the selected parameters are changed. In all cases, average navigation duration is not affected by the changes in the selected parameters in the chosen ranges

Overall, RLSS does not result in collisions or deadlocks when parameters are not set to extreme values. In addition, changes in parameters, outside of extreme ranges, do not result in significant changes in the average navigation durations. These suggest that RLSS does not need extensive parameter tuning.

7.2 Ablation study

We investigate the effects of (i) the prepend operation of the discrete planning stage (Line 4, Algorithm 2) and (ii) the preferred distance cost term $\mathcal{J}_{pref}$ of the trajectory optimization stage to the performance of the algorithm. The prepend operation enables the kinematic feasibility of the generated optimization problem as shown in Remark 2. The preferred distance cost term increases the numerical stability of the algorithm by encouraging robots to create a gap between themselves and other objects.

We consider four versions our algorithm: RLSS, RLSS without the prepend operation (RLSS w/o prepend), RLSS without the preferred distance cost term (RLSS w/o pref. dist.) and RLSS with neither. We set step size $\sigma = 0.77m$, obstacle check distance $\tilde{o} = 1.0m$, robot check distance $\tilde{r} = 2.0m$, degree of Bézier curves $h = 12$, and $c = 1$ (continuity up to velocity). Robots navigate in 3D maze like environments. The desired trajectories are set to straight line segments connecting robot start positions to goal positions and the durations of the segments are set to the length of the

line segments divided by the maximum speed of the robots. During simulation, robots continue using their existing plans when planning fails. Also, colliding robots continue navigating and are not removed from the experiment.

We generate ten 3D-maze like environments and list the average and standard deviation values for our metrics in Table 2. We report the failure rate of all algorithms in the form of the ratio of number of failures to the number of planning iterations (*Fail Rate* column in Table 2), the number of robots that are involved in at least one collision during navigation (*# Coll.* column in Table 2) and the average navigation duration of all robots (*Avg. Nav. Dur.* column in Table 2).

The failure rate of RLSS is 0.01% on average. RLSS w/o pref. dist. fails 0.06% of the time. RLSS w/o prepend fails 12.73% of the time. This is drastically more than RLSS w/o pref. dist because our prepend operation ensures the kinematic feasibility of the optimization problem while our preferred distance cost term tackles numerical instabilities only. RLSS with neither results in a failure rate of 10.62%. Interestingly, RLSS with the preferred distance cost term but without the prepend operation (RLSS w/o prepend) results in a higher failure rate than RLSS with neither. We do not investigate the root cause of this since failure rates in both cases are a lot higher than of RLSS.

The effects of failures are seen in the next two metrics. RLSS results in no collisions. The number of colliding robots increase to 0.67 in RLSS w/o pref. dist, 6.78 in RLSS w/o prepend and 8.67 in RLSS with neither on average. The

Table 2 The results of the ablation study

	Fail Rate	# Coll	Avg. Nav. Dur [s]
RLSS	**0.89 (1.45) / 7904 (297)**	**0 (0)**	**24.68 (0.93)**
RLSS w/o pref. dist	4.78 (9.81) / 8194 (489)	0.67 (1)	25.58 (1.53)
RLSS w/o prepend	1556 (120) / 12,220 (538)	6.78 (3.73)	38.17 (1.68)
RLSS with neither	1569 (101) / 14,778 (412)	8.67 (2.87)	46.16 (1.29)

We compare four different versions of RLSS. We ablate (i) the prepend operation of the discrete planning stage and (ii) the preferred distance cost term of the trajectory optimization stage. The details of the metrics are given in Sect. 7.2. The reported values are means and standard deviations (given in parentheses) of 10 experiments in random maze-like environments. Both prepend operation and preferred distance cost term are important for the effectiveness of the algorithm

average navigation duration of robots is lowest in RLSS. It increases by 3.65% in RLSS w/o pref. dis, 54.66% in RLSS w/o prepend and 87.03% in RLSS with neither compared to RLSS.

These results show that the prepend operation is more important than the preferred distance cost term for the success of the algorithm. Nevertheless, RLSS needs both to be safe and effective.

7.3 Comparisons with baseline planners

We compare the performance of our planner to two baseline planners that do not require communication in 3D experiments. We set step size $\sigma = 0.77m$, obstacle check distance $\tilde{o} = 1.0m$, robot check distance $\tilde{r} = 2.0m$, and degree of Bézier curves $h = 12$ in RLSS in all cases.

7.3.1 Extended buffered Voronoi cell (eBVC) planner

The first baseling planner is a MPC-style planner based on buffered Voronoi cells introduced by Zhou et al. (2017), which we call BVC. In the BVC approach, each robot computes its Voronoi cell within the Voronoi tesselation of the environment. This is done by using the position information of other robots. Robots buffer their Voronoi cells to account for their collision shapes and plan their trajectories within their corresponding buffered Voronoi cells. Similar to RLSS, BVC does not require any communication between robots, requires perfect sensing of robot positions in the environment, and the resulting trajectories are safe only when planning is synchronized between robots. They also require that each robot stays within its buffered Voronoi cell until the next planning iterations. Unlike RLSS, BVC is based on buffered Voronoi cells and it cannot work with arbitrary convex objects. Instead, robots are modeled as hyperspheres in BVC.

The original formulation presented in the BVC article only allows position state for the robots, which cannot model a rich set of dynamics, including double, triple, or higher order integrators. We extend their formulation to all discrete linear time invariant systems with position output. We define the systems with three matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$. Since the output of the system is the position of the robot, it cannot depend on the current input, hence $\mathbf{D} = \mathbf{0}$. This allows us to formulate the problem for a richer set of dynamics and have constraints on higher order derivatives than robot velocity.

BVC as published also does not consider obstacles in the environment. We extend their formulation to static obstacles by modeling obstacles as robots. Since obstacles are static, they stay within their buffered Voronoi cells at all times. Since we model obstacles as robots, extended BVC does not require the ability of distinguishing robots from obstacles.

Our extended BVC formulation, which we call eBVC, is as follows:

$$\min_{\mathbf{u}_0, \ldots, \mathbf{u}_{M-1}} \sum_{i=1}^{M} \lambda_i \left\| \mathbf{p}_i - \mathbf{d}(\tilde{T} + M\Delta t) \right\|_2^2 + \sum_{i=0}^{M-1} \theta_i \left\| \mathbf{u}_i \right\|_2^2 \, s.t.$$

$$\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i + \mathbf{B}\mathbf{u}_i \quad \forall i \in \{0, \ldots, M-1\}$$

$$\mathbf{p}_i = \mathbf{C}\mathbf{x}_i \quad \forall i \in \{0, \ldots, M\}$$

$$\mathbf{p}_i \in \mathcal{V} \quad \forall i \in \{0, \ldots, M\}$$

$$\mathbf{p}_i \in \mathcal{W} \quad \forall i \in \{0, \ldots, M\}$$

$$\mathbf{u}_{min} \preceq \mathbf{u}_i \preceq \mathbf{u}_{max} \quad \forall i \in \{0, \ldots, M-1\}$$

$$\mathbf{x}_{min} \preceq \mathbf{x}_i \preceq \mathbf{x}_{max} \quad \forall i \in \{0, \ldots, M\}$$

where $\tilde{T}$ is the current timestamp, $\mathbf{d}(t)$ is the desired trajectory for the robot, Δt is the discretization timestep of the system, M is the number of steps to plan for, $\mathbf{u}_i$ is the input to apply from timestep i to $i + 1$, $\mathbf{x}_i$ is the state at timestep i, $\mathbf{p}_i$ is the position at timestep i, $\mathcal{V}$ is the buffered Voronoi cell of the robot, $\mathbf{u}_{min}$ and $\mathbf{u}_{max}$ are the limits for the inputs, $\mathbf{x}_{min}$ and $\mathbf{x}_{max}$ are the limits for the states (which can be used to bound velocity in a double integrator system, or velocity and acceleration in a triple integrator system for example), and $\mathcal{W}$ is the workspace of the robot. $M\Delta t$ is the planning horizon. A robot plans toward the position $\mathbf{d}(\tilde{T} + M\Delta t)$, which is the position of the robot after the planning horizon if it could follow the desired trajectory perfectly. $\mathbf{x}_0$ is the current state of the robot. The first term of the cost function penalizes deviation from the goal position for the final and each inter-

mediate position with different weights λ_i. The second term of the cost function is the input cost that penalizes input magnitudes with different weights θ_i. We apply the first input of the solution for duration Δt, and replan at the next timestep.

We use our own implementation of eBVC as explained above during the comparisons.

7.3.2 Relative safe flight corridor (RSFC) planner

The second planner we compare against is presented by Park and Kim (2021), in which piecewise Bézier curves are computed, executed for a short duration, and replanning is done at the next iteration similar to our work. It utilizes the fact that the difference of two Bézier curves is another Bézier curve by constraining these relative Bézier curves to be inside safe regions (relative safe flight corridors, or RSFCs) defined according to robot collision shapes. We call this algorithm RSFC for short. RSFC does not require any communication between robots. It utilizes both positions and velocities of other objects in the environment, hence requires more sensing information than our algorithm. Velocities are used to predict the trajectories of other robots as piecewise Bézier curves. While it can handle dynamic obstacles as well, we use it in static environments in our comparisons, since RLSS does not handle dynamic obstacles explicitly.

We use the authors' implementation of RSFC during our comparisons.

7.3.3 Experiments and results

We compare RLSS against eBVC and RSFC in 10 different experiments differing in required degree of continuity, desired trajectories, and map of the environment. In all experiments, 32 robots are placed in a circle formation with radius $20m$ in 3D. The task is to swap the positions of robots to the antipodal points on the circle. The results of the experiments are summarized in Table 3.

There are 3 maps we use: empty, forest (Fig. 8a), and maze (Fig. 8b), listed in the map column of Table 3. In the empty map, there are no obstacles in the environment. The forest map is a random forest with 10% occupancy; it has a radius of $15m$ and each tree is a cylinder with radius $0.5m$. The maze map is a maze-like environment with choke regions.

We compute the desired trajectories of the robots by running a single-agent shortest path using the discrete planning stage of RLSS. We run single-agent shortest path on a prior map, which is set to either a full map of the environment or to an empty map; this is listed in the prior map column of Table 3. When the prior map is empty, single-agent shortest paths are straight line segments connecting robot start positions to robot goal positions.

We set two different continuity requirements: velocity and acceleration, listed in the continuity column of Table 3. When
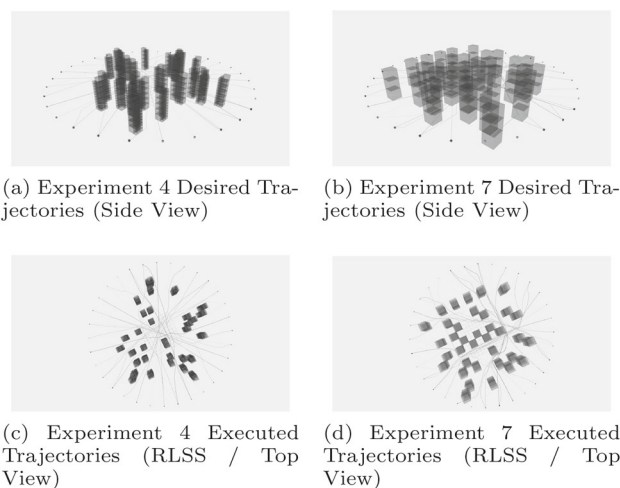

(a) Experiment 4 Desired Trajectories (Side View)

(b) Experiment 7 Desired Trajectories (Side View)

(c) Experiment 4 Executed Trajectories (RLSS / Top View)

(d) Experiment 7 Executed Trajectories (RLSS / Top View)

Fig. 8 Desired trajectories and the used forest map of experiment 4 is given in (**a**). Same forest map is used in each forest experiment. Desired trajectories and the used maze map of experiment 7 is given in (**b**). Same maze map is used in each maze experiment. **c** shows the executed trajectories of robots running RLSS from top in experiment 4. **d** shows the executed trajectories of robots running RLSS from top in experiment 7

velocity continuity is required, the system of eBVC is a double integrator with position output. When acceleration continuity is required, the system of eBVC is a triple integrator with position output.

We compare RSFC to RLSS and eBVC only with acceleration continuity requirement because the authors' implementation hard codes the acceleration continuity requirement, while in theory it can work with any degree. Also, we compare RSFC only in the case of an empty prior map in order not to change RSFC's source code, while in theory it can be guided with arbitrary trajectories.

We plan for $5s$ long trajectories in every $0.1s$ with all algorithms. In eBVC, we plan for $M = 50$ steps with discretization timestep $\Delta t = 0.1s$. We set state and input upper and lower bounds in eBVC in order to obey the dynamic limits of the robots. We set distance to goal weights $\lambda_1 = 120$, $\lambda_i = 20 \ \forall i \geq 2$ in eBVC, putting more importance on the position of the robot after 1 timestep. We set $\theta_i = 1 \ \forall i$ in eBVC.

Both eBVC and RSFC require spherical obstacles. We use the smallest spheres containing each leaf-level box of the OcTree structure as obstacles in eBVC and RSFC. Similar to RLSS, we use robot and obstacle check distance to limit the number of obstacles considered at each iteration. We set both obstacle and robot check distance $\tilde{o} = \tilde{r} = 2.0m$ in eBVC, and set $\tilde{o} = \tilde{r} = 5.0m$ in RSFC, since smaller values in RSFC result in a high number of collisions and higher values for the parameters do not improve the success of the algorithms. Robots are modeled as spheres in eBVC and RSFC as well. We set robot shapes to spheres with radius $0.173m$, which

Table 3 The results of the comparisons of RLSS, eBVC, and RSFC are summarized

Experiment	Map	Prior Map	Continuity	Algorithm	# Deadlocks	# Coll. Robots	Avg. Collision Dur. [s]	Avg. Nav. Dur. [s]	Avg. Comp. Time [ms]
1	Empty	Empty	Velocity	RLSS	0	0	NA	22.37	107
				eBVC	0	0	NA	**18.50**	**106**
2			Acceleration	RLSS	0	0	NA	22.12	110
				eBVC	0	0	NA	20.65	161
				RSFC	0	0	NA	**19.83**	**47**
3	Forest	Forest	Velocity	RLSS	**0**	**0**	**NA**	23.11	**182**
				eBVC	10	7	0.53	**18.94**	387
4			Acceleration	RLSS	**0**	**0**	**NA**	23.06	**147**
				eBVC	8	13	0.83	**21.24**	763
5	Forest	Empty	Velocity	RLSS	**0**	**0**	**NA**	22.62	**192**
				eBVC	8	24	1.35	**18.54**	186
6			Acceleration	RLSS	**0**	**0**	**NA**	22.72	244
				eBVC	11	15	0.71	**21.28**	970
				RSFC	**0**	6	0.42	21.82	**201**
7	Maze	Maze	Velocity	RLSS	**0**	**0**	**NA**	25.09	160
				eBVC	21	9	0.67	**20.12**	**145**
8			Acceleration	RLSS	**0**	**0**	**NA**	25.32	**170**
				eBVC	17	13	1.49	**23.80**	264
9	Maze	Empty	Velocity	RLSS	**0**	**0**	**NA**	**27.98**	410
				eBVC	32	0	NA	NA	**176**
10			Acceleration	RLSS	**0**	**0**	**NA**	32.04	386
				eBVC	30	2	3.26	**21.75**	407
				RSFC	**0**	22	0.34	28.23	**80**

Each experiment differ in the map used during navigation, prior map used during desired trajectory computation, and the required degree of continuity. RLSS results in longer navigation durations than eBVC and RSFC on average; but eBVC suffers from deadlocks in environments with obstacles and RSFC results in collisions in environments with obstacles

are the smallest spheres containing the actual robot shapes. We count collisions only when contained leaf-level OcTree boxes and contained robot shapes intersect. This gives both eBVC and RSFC buffer zones for collisions.

We report the number of deadlocking robots (*# Deadlocks* column in Table 3), number of robots that are involved in at least one collision (*# Coll. Robots* column in Table 3), collision duration of robots that are involved in collisions averaged over robots (*Avg. Collision Dur.* column in Table 3), average navigation duration of non-deadlocking robots from their start positions to goal positions (*Avg. Nav. Dur.* column in Table 3), and computation time per iteration averaged over each planning iteration of each robot (*Avg. Comp. Time* column in Table 3). We continue the navigation of colliding robots and do not remove them from the experiment.

Executed trajectories of robots running RLSS are shown in Fig. 8c for experiment 4, and in Fig. 8d for experiment 7 as examples.

RLSS does not result in any deadlocks or collisions in all cases. eBVC has a significant number of deadlocks and RSFC results in collisions in experiments with obstacles.

When there are no obstacles in the environment, e.g., in experiments 1 and 2, no algorithm results in deadlocks or collisions. When velocity continuity is required, e.g., in experiment 1, the average navigation duration of robots running eBVC is 18% lower than those that run RLSS. Both eBVC and RLSS run close to $9Hz$ on average. When acceleration continuity is required, e.g., in experiment 2, the average navigation duration of robots running RSFC is 10% lower than those that run RLSS; and the average navigation duration for eBVC is 7% lower than that for RLSS. RLSS runs at about $9Hz$ on average, eBVC runs close to $6Hz$ on average, and RSFC runs close to $21Hz$ on average. When there are no obstacles in the environment, the discrete search of RLSS results in unnecessary avoidance movements, which is the main reason for average navigation duration differences.

When there are obstacles in the environment, performance of both eBVC and RSFC degrades in terms of the number of deadlocks and collisions.

In experiment 3, even if the full prior map of the environment is given during initial discrete search with only velocity continuity, 10 out of 32 robots deadlock, and 7 out of the remaining 22 get involved in at least one collision when eBVC is used. When acceleration continuity is required (experiment 4), 8 robots deadlock and 13 other robots get involved in collisions, resulting in only 11 robot reaching their goal volumes without collisions in eBVC. RLSS both works faster than eBVC and results in no deadlocks or collisions in those cases.

When the prior map is not known in the forest environment (experiments 5 and 6), eBVC results in a lot of deadlocks and collisions. All robots in experiment 5 either deadlock or collide when eBVC is used. In experiment 6, RSFC does a lot better than eBVC. RSFC results in no deadlocks while eBVC results in 11 deadlocks. 15 out of remaining 21 robots get involved in at least one collision when eBVC is used with an average collision duration of $0.71s$. 6 out of 32 robots running RSFC collide at least once with average collision duration of $0.42s$. RLSS does not result in any deadlocks or collisions.

When the environment is a complicated maze, the performance of both eBVC and RSFC degrades more. With full prior map and velocity continuity (experiment 7), 21 out of 32 robots deadlock, 9 out of remaining 11 are involved in collisions, leaving only 2 reaching to their goal volumes without an incident when eBVC is used. With full prior map and acceleration continuity (experiment 8), 17 out of 32 robots deadlock, 13 out of remaining 15 are involved in collisions, again leaving only 2 that reach their goal volumes without incident when eBVC is used. RLSS does not result in any deadlocks or collisions in these scenarios. When the prior map is empty, i.e. the desired trajectories are straight line segments, the performance of eBVC degrades even more. All robots deadlock in the case with velocity continuity (experiment 9), while 30 out of 32 robots deadlock and the rest get involved in collisions in the case with acceleration continuity (experiment 10). No robot running RSFC deadlocks but 22 out of 32 get involved in collisions at least once with $0.34s$ average collision duration in experiment 10 with acceleration continuity. RLSS does not result in any deadlocks or collisions in those cases.

Overall, robots running RLSS have higher navigation durations than those that use eBVC or RSFC in all experiments. While the higher navigation duration is not an important metric when other algorithms cause deadlocks or collisions, it is an important metric when they do not (experiments 1 and 2 with no obstacles in particular). eBVC does not have an integrated discrete planner, which is the reason for its good performance in terms of average navigation durations. When robots are close to each other, RLSS uses the free space less effectively since discrete planning has a step size $0.77m$. eBVC does not rely on discrete planning and hence avoids other robots by executing small direction changes. RSFC utilizes velocities of other robots on top of positions, which allows it to estimate the intents of robots more effectively, resulting in a better usage of the free space, and hence results in better navigation durations on average. Since RLSS does not utilize communication and uses only positions of other robots, it cannot deduce the intents of other robots. This results in fluctuations of plans between planning iterations, which increases the average navigation durations of robots. Fluctuations of plans increase when the environment is dense as seen in the supplemental video.[10] If the environment becomes overly constraining, e.g. tens of robots

---

[10] https://youtu.be/Jrdvf2qyzrg.

(a) Heterogeneous team of differential drive robots navigating through an environment without obstacles. A person changes the positions of the robots while RLSS is running.

(b) 6 Crazyflie 2.0s are navigating through an environment without obstacles.

(c) 6 Crazyflie 2.0s are navigating through an environment with obstacles.

(d) 6 Crazyflie 2.0s are navigating through an environment with obstacles. A person changes the positions of the robots while RLSS is running.

Fig. 9 Physical robot experiments using Turtlebot2s, Turtlebot3s, iRobot Create2s, and Crazyflie 2.0s. RLSS works in real-time under external disturbances

trying to pass through a narrow tube, these fluctuations may turn into livelocks.

However, when obstacles are introduced in the environment, the performance of RLSS is better than other algorithms. eBVC suffers greatly from deadlocks. RSFC does not result in deadlocks but results in collisions, even while using more information than RLSS (velocity and position instead of position only).

A note about the statistical significance of the results. In each experiment, we run each algorithm on single randomly generated forest-like or maze-like environment. To show that the results are consistent for environment types, we generate 10 forest-like environments with the same parameters for experiment 3 and run RLSS and eBVC. RLSS does not result in deadlocks or collisions in any of the cases. The average navigation duration of robots averaged over 10 environments is 19.15*s* with standard deviation 0.24*s* for eBVC and 23.59*s* with standard deviation 0.25*s* for RLSS. The ratio between average navigation durations when eBVC or RLSS is used is consistent with the reported values given in Table 3 for experiment 3.

7.4 Physical robots

We implement and run RLSS on physical robots using iRobot Create2s, Turtlebot2s and Turtlebot 3s in 2D; and Crazyflie 2.0s in 3D. We use a VICON motion tracking system for localization. Robots do not sense, but receive the position of others using the VICON system. iRobot Create2s,

Turtlebot3s, and Turtlebot2s are equipped with ODROID XU4 and ODROID C1+ single board computers running ROS Kinetic on the Ubuntu 16.04 operating system. In all cases the algorithm is run on a centralized base station computer using separate processes for each robot. Therefore, unlike simulations, planning is not synchronized between robots on real robot implementations. RLSS does not result in any deadlocks in collisions in these asynchronized deployments as well. Commands are sent to 2D robots over a WiFi network, and to Crazyflie 2.0s directly over their custom radio.

We conduct external disturbance experiments with 2 iRobot Create2s, 3 Turtlebot3s, and 2 Turtlebot2s (Fig. 9a). A human changes the positions of some robots by moving them arbitrarily during execution several times. In all cases, robots replan in real-time and avoid each other successfully.

We demonstrate the algorithm in 3D using 6 Crazyflie 2.0 s. We conduct an experiment without obstacles in which Crazyflies swap positions with straight lines as desired trajectories (Fig. 9b). In another experiment, we show that Crazyflies can navigate through an environment with obstacles (9c, d). In each case, we externally disturb the Crazyflies and show that they can replan in real-time.

The recordings for our physical robot experiments are included in the supplemental video.[11]

8 Conclusion

In this article, we present RLSS, a real-time decentralized long horizon trajectory planning algorithm for the navigation of multiple robots in shared environments with obstacles that provides guarantees on collision avoidance if the resulting problems are feasible. The generated optimization problem to compute a smooth trajectory is convex and kinematically feasible. It does not require any communication between robots, requires only position sensing of obstacles and robots in the environment, and robots to be able to distinguish other robots from obstacles. With its comparatively minimal sensing requirements and no reliance on communication, it presents a new baseline for algorithms that require communication and sensing/prediction of higher order state components of other robots. The algorithm considers the dynamic limits of the robots explicitly and enforces safety using hard constraints.

We show in synchronized simulation that RLSS performs better than two state-of-the-art planning algorithms (eBVC and RSFC), one of which requires velocity sensing on top of position sensing, in environments with obstacles in terms of

---

[11] Since we define robots' goals as single points, i.e. sets of measure zero, in physical experiments, robots keep missing their goals slightly. This results in a spinning behavior in 2D since robots continuously fix their positions by replanning.

number of deadlocks and number of colliding robots. In our experiments, RLSS does not result in any deadlocks or collisions, while eBVC suffers from deadlocks and RSFC results in collisions (while RLSS provides theoretical guarantees on collision avoidance when it succeeds, it does not provide theoretical guarantees on deadlock avoidance). When there are no obstacles in the environment, RSFC and eBVC outperform RLSS in terms of average navigation duration by 7–20%.

In future work, we would like to extend our planner to asynchronously planning robots where each robot starts and finishes planning at different unknown timestamps by utilizing communication. Also, we would like to incorporate the noise in the sensing systems within our algorithm in order to solve the cases with imperfect sensing in a principled way.

Declarations

Conflict of interest Baskın Şenbaşlar, the first author, has moved to Prof. Gaurav S. Sukhatme's group in January 2022 after the initial submission of this paper. Prof. Sukhatme is the editor-in-chief of Autonomous Robots journal and the advisor of Baskın Şenbaşlar at the time of writing. Prof. Sukhatme was not involved in ideation, writing and experiments of this paper in any way. This paper is submitted to *Robot Swarms in the Real World: from Design to Deployment* special issue of Autonomous Robots, which has Dr. Siddharth Mayya as the lead guest editor.

Appendix A: Sweep of a convex shape along a line segment is convex

Lemma 2 *Let* $\mathcal{R}(\mathbf{x}) = \{\mathbf{x}\} \oplus \mathcal{R}_0$ *be the Minkowski sum of* $\{\mathbf{x}\} \in \mathbb{R}^d$ *and* $\mathcal{R}_0 \subseteq \mathbb{R}^d$, *and let* $\mathcal{R}_0$ *be a convex set. Let* $\mathbf{p}(t) = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$ *where* $t \in [0, 1]$ *be the line segment from* $\mathbf{a} \in \mathbb{R}^d$ *to* $\mathbf{b} \in \mathbb{R}^d$. *Then, the swept volume of* $\mathcal{R}$ *along* $\mathbf{p}(t)$ *is convex.*

Proof The swept volume ζ of $\mathcal{R}$ from $\mathbf{a}$ to $\mathbf{b}$ can be defined as $\zeta = \cup_{t=0}^{1} \mathcal{R}(\mathbf{p}(t)) = \cup_{t=0}^{1}\{\mathbf{p}(t)\} \oplus \mathcal{R}_0$. Choose two points $\mathbf{q}_1, \mathbf{q}_2 \in \zeta$. $\exists t_1 \in [0, 1] \exists \tilde{\mathbf{q}}_1 \in \mathcal{R}_0 \; \mathbf{q}_1 = \mathbf{p}(t_1) + \tilde{\mathbf{q}}_1$ and $\exists t_2 \in [0, 1] \exists \tilde{\mathbf{q}}_2 \in \mathcal{R}_0 \; \mathbf{q}_2 = \mathbf{p}(t_2) + \tilde{\mathbf{q}}_2$. Let $\mathbf{q}' = (1 - t')\mathbf{q}_1 + t'\mathbf{q}_2$ be a point on the line segment connecting $\mathbf{q}_1$ and $\mathbf{q}_2$ where $t' \in [0, 1]$.

$$\mathbf{q}' = (1 - t')(\mathbf{p}(t_1) + \tilde{\mathbf{q}}_1) + t'(\mathbf{p}(t_2) + \tilde{\mathbf{q}}_2)$$
$$= (1 - t')(\mathbf{a} + t_1(\mathbf{b} - \mathbf{a}) + \tilde{\mathbf{q}}_1)$$
$$+ t'(\mathbf{a} + t_2(\mathbf{b} - \mathbf{a}) + \tilde{\mathbf{q}}_2)$$
$$= (1 - t')\tilde{\mathbf{q}}_1 + t'\tilde{\mathbf{q}}_2$$
$$+ \mathbf{a} + (t_1(1 - t') + t_2t')(\mathbf{b} - \mathbf{a})$$

$(t_1(1 - t') + t_2t') \in [0, 1]$ because it is a convex combination of $t_1, t_2 \in [0, 1]$. Therefore $\mathbf{a} + (t_1(1 - t') + t_2t')(\mathbf{b} - \mathbf{a}) \in \mathbf{p}(t)$. Also, $(1 - t')\tilde{\mathbf{q}}_1 + t'\tilde{\mathbf{q}}_2 \in \mathcal{R}_0$, because it is a convex combination of $\tilde{\mathbf{q}}_1, \tilde{\mathbf{q}}_2 \in \mathcal{R}_0$ and $\mathcal{R}_0$ is convex. Therefore, $\mathbf{q}' \in \zeta$ since $\exists t \in [0, 1] \; \mathbf{q}' \in \{\mathbf{p}(t)\} \oplus \mathcal{R}_0$. Hence, ζ is convex. $\square$

References

Alonso-Mora, J., Breitenmoser, A., Rufli, M., Beardsley, P., Siegwart, R. (2013). Optimal reciprocal collision avoidance for multiple nonholonomic robots. In *Distributed autonomous robotic systems: The 10th international symposium* (pp. 203–216). https://doi.org/10.1007/978-3-642-32723-0.

Barer, M., Sharon, G., Stern, R., & Felner, A. (2014). Suboptimal variants of the conflict based search algorithm for the multiagent pathfinding problem. *Frontiers in Artificial Intelligence and Applications, 263*, 961–962. https://doi.org/10.3233/978-1-61499-419-0-961

Batra, S., Huang, Z., Petrenko, A., Kumar, T., Molchanov, A., Sukhatme, G. S. (2021). Decentralized control of quadrotor swarms with end-to-end deep reinforcement learning. In *5th Conference on robot learning*. CoRL 2021.

Bhattacharya, S., Kumar, V., Likhachev, M. (2010). Search-based path planning with homotopy class constraints. In *Proceedings of the twenty-fourth AAAI conference on artificial intelligence*, pp. 1230–1237.

Campion, G., Bastin, G., & Dandrea-Novel, B. (1996). Structural properties and classification of kinematic and dynamic models of wheeled mobile robots. *IEEE Transactions on Robotics and Automation, 12*(1), 47–62. https://doi.org/10.1109/70.481750

Chen, J., Liu, T., Shen, S. (2016). Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments. In *2016 IEEE international conference on robotics and automa-

tion (ICRA) (pp. 1476–1483). https://doi.org/10.1109/ICRA.2016. 7487283.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning, 20*(3), 273–297. https://doi.org/10.1007/BF00994018

Desai, A., & Michael, N. (2020). Online planning for quadrotor teams in 3-d workspaces via reachability analysis on invariant geometric trees. In *IEEE international conference on robotics and automation (ICRA)* (pp. 8769–8775). https://doi.org/10.1109/ICRA40945.2020.9197195.

Dresner, K., & Stone, P. (2008). A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research, 31*, 591–656. https://doi.org/10.1613/jair.2502.

Farouki, R. T. (2012). The Bernstein polynomial basis: A centennial retrospective. *Computer Aided Geometric Design, 29*(6), 379–419. https://doi.org/10.1016/j.cagd.2012.03.001

Furda, A., & Vlacic, L. (2011). Enabling safe autonomous driving in real-world city traffic using multiple criteria decision making. *IEEE Intelligent Transportation Systems Magazine, 3*(1), 4–17. https://doi.org/10.1109/MITS.2011.940472

Gondzio, J., & Grothey, A. (2009). Exploiting structure in parallel implementation of interior point methods for optimization. *Computational Management Science, 6*(2), 135–160. https://doi.org/10.1007/s10287-008-0090-3

Harabor, D., & Grastien, A. (2011). Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI conference on artificial intelligence*, vol. 25, pp. 1114–1119.

Hönig, W., Preiss, J. A., Kumar, T. K. S., Sukhatme, G. S., & Ayanian, N. (2018). Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics, 34*(4), 856–869. https://doi.org/10.1109/TRO.2018.2853613

Homm, F., Kaempchen, N., Ota, J., Burschka, D. (2010). Efficient occupancy grid computation on the gpu with lidar and radar for road boundary detection. In *IEEE intelligent vehicles symposium (IV)* (pp. 1006–1013). https://doi.org/10.1109/IVS.2010.5548091.

Hopcroft, J., Schwartz, J., & Sharir, M. (1984). On the complexity of motion planning for multiple independent objects; pspacehardness of the "warehouseman's problem". *The International Journal of Robotics Research, 3*(4), 76–88. https://doi.org/10.1177/027836498400300405

Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., & Burgard, W. (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots, 34*(3), 189–206. https://doi.org/10.1007/s10514-012-9321-0

Jiang, S., & Song, K. (2013). Differential flatness-based motion control of a steerand-drive omnidirectional mobile robot. In *IEEE international conference on mechatronics and automation (ICMA)* (pp. 1167–1172). https://doi.org/10.1109/ICMA.2013.6618079.

Karaman, S., & Frazzoli, E. (2010). Incremental sampling-based algorithms for optimal motion planning. *Robotics Science and Systems VI, 104*(2). https://doi.org/10.15607/RSS.2010.VI.034.

Lam, E., Le Bodic, P., Harabor, D.D., Stuckey, P.J. (2019). Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the twenty-eighth international joint conference on artificial intelligence, IJCAI-19* (pp. 1289–1296). https://doi.org/10.24963/ijcai.2019/179.

Li, Q., Gama, F., Ribeiro, A., Prorok, A. (2020). Graph neural networks for decentralized multi-robot path planning. *IEEE/RSJ international conference on intelligent robots and systems (IROS)* (pp. 11785–11792). https://doi.org/10.1109/iros45743.2020.9341668.

Liu, S., Watterson, M., Mohta, K., Sun, K., Bhattacharya, S., Taylor, C. J., & Kumar, V. (2017). Planning dynamically feasible trajectories for quadrotors using safe flight corridors in 3-D complex environments. *IEEE Robotics and Automation Letters, 2*(3), 1688–1695. https://doi.org/10.1109/LRA.2017.2663526

Luis, C., Vukosavljev, M., & Schoellig, A. (2020). Online trajectory generation with distributed model predictive control for multirobot

motion planning. *IEEE Robotics and Automation Letters*, pp. 1–1. https://doi.org/10.1109/LRA.2020.2964159.

Ma, H., Harabor, D., Stuckey, P.J., Li, J., Koenig, S. (2019). Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence, 33*(01), 7643–7650. https://doi.org/10.1609/AAAI.V33I01.33017643.

Mellinger, D., & Kumar, V. (2011). Minimum snap trajectory generation and control for quadrotors. In *IEEE international conference on robotics and automation (ICRA)* (pp. 2520–2525). https://doi.org/10.1109/ICRA.2011.5980409.

Murray, R. M., Rathinam, M., Sluis, W. (1995). Differential flatness of mechanical control systems: A catalog of prototype systems. In *ASME international mechanical engineering congress and exposition*.

Murray, R. M., & Sastry, S. S. (1993). Nonholonomic motion planning: steering using sinusoids. *IEEE Transactions on Automatic Control, 38*(5), 700–716. https://doi.org/10.1109/9.277235.

Oleynikova, H., Taylor, Z., Fehr, M., Siegwart, R., & Nieto, J. (2017). Voxblox: Incremental 3D euclidean signed distance fields for onboard mav planning. In *IEEE/RSJ international conference on intelligent robots and systems (IROS)*. https://doi.org/10.1109/IROS.2017.8202315.

O'Meadhra, C., Tabib, W., & Michael, N. (2019). Variable resolution occupancy mapping using gaussian mixture models. *IEEE Robotics and Automation Letters, 4*(2), 2015–2022. https://doi.org/10.1109/LRA.2018.2889348

Park, J., & Kim, H. J. (2021). Online trajectory planning for multiple quadrotors in dynamic environments using relative safe flight corridor. *IEEE Robotics and Automation Letters, 6*(2), 659–666. https://doi.org/10.1109/LRA.2020.3047786

Park, J., Kim, J., Jang, I., Kim, H. J. (2020). Efficient multi-agent trajectory planning with feasibility guarantee using relative Bernstein polynomial. In *IEEE international conference on robotics and automation (ICRA)* (pp. 434–440). https://doi.org/10.1109/ICRA40945.2020.9197162.

Peterson, R., Buyukkocak, A. T., Aksaray, D., & Yazicioğlu, Y. (2021). Distributed safe planning for satisfying minimal temporal relaxations of twtl specifications. *Robotics and Autonomous Systems, 142*, 103801. https://doi.org/10.1016/j.robot.2021.103801.

Piegl, L., & Tiller, W. (1995). *The NURBS book*. Springer. https://doi.org/10.1007/978-3-642-97385-7.

Richter, C., Bry, A., Roy, N. (2013). Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. In *International symposium of robotic research (ISRR)* (vol. 114, pp. 649–666). https://doi.org/10.1007/978-3-319-28872-7_37.

Riviere, B., Hönig, W., Yue, Y., Chung, S.-J. (2020). Glas: Globalto-local safe autonomy synthesis for multi-robot motion planning with end-to-end learning. *IEEE Robotics and Automation Letters*, pp. 1–1. https://doi.org/10.1109/LRA.2020.2994035.

Sartoretti, G., Kerr, J., Shi, Y., Wagner, G., Kumar, T. K., Koenig, S., & Choset, H. (2019). Primal: Pathfinding via reinforcement and imitation multi-agent learning. *IEEE Robotics and Automation Letters, 4*, 2378–2385. https://doi.org/10.1109/LRA.2019.2903261

Şenbaşlar, B., Hönig, W., Ayanian, N. (2019). Robust trajectory execution for multi-robot teams using distributed real-time replanning. In *Distributed Autonomous Robotic Systems (DARS)* (pp. 167–181). https://doi.org/10.1007/978-3-030-05816-6_12.

Şenbaşlar, B., Hönig, W., Ayanian, N. (2021). RLSS: Real-time multi-robot trajectory replanning using linear spatial separations. Retrieved from arXiv:2103.07588.

Şenbaşlar, B., & Sukhatme, G. (2022). Asynchronous real-time decentralized multirobot trajectory planning. In *IEEE/RSJ international conference on intelligent robots and systems (IROS 2022)*.

Sharon, G., Stern, R., Felner, A., & Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence, 219*, 40–66. https://doi.org/10.1016/j.artint.2014.11.006

Solovey, K., Salzman, O., & Halperin, D. (2013). Finding a needle in an exponential haystack: Discrete RRT for exploration of implicit roadmaps in multi-robot motion planning. In *The International Journal of Robotics Research*, *107*. https://doi.org/10.1007/978-3-319-16595-0_34.

Tang, S., & Kumar, V. (2016). Safe and complete trajectory generation for robot teams with higher-order dynamics. In *IEEE/RSJ international conference on intelligent robots and systems (IROS)* (pp. 1894–1901). https://doi.org/10.1109/IROS.2016.7759300.

Tordesillas, J., & How, J. P. (2020). MINVO basis: Finding simplexes with minimum volume enclosing polynomial curves. arXiv preprint arXiv:2010.10726.

Tordesillas, J., & How, J. P. (2021). MADER: Trajectory planner in multi-agent and dynamic environments. *IEEE Transactions on Robotics*. https://doi.org/10.1109/TRO.2021.3080235

Usenko, V., Von Stumberg, L., Pangercic, A., Cremers, D. (2017). Real-time trajectory replanning for mavs using uniform b-splines and a 3d circular buffer. In *IEEE/RSJ international conference on intelligent robots and systems (IROS)* (pp. 215–222). https://doi.org/10.1109/IROS.2017.8202160.

Wang, L., Ames, A.D., Egerstedt, M. (2017). Safety barrier certificates for collisions-free multirobot systems. *IEEE Transactions on Robotics, 33*(3), 661–674. https://doi.org/10.1109/TRO.2017.2659727.

Wang, X., Xi, L., Chen, Y., Lai, S., Lin, F., Chen, B.M. (2021). Decentralized mpc-based trajectory generation for multiple quadrotors in cluttered environments. *Guidance, Navigation and Control, 01*(02), 2150007. https://doi.org/10.1142/S2737480721500072.

Wurman, P., D'Andrea, R., & Mountz, M. (2008). Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine, 29*, 9–20.

Yu, J., & LaValle, S. M. (2013). Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the twenty-seventh AAAI conference on artificial intelligence* (pp. 1443–1449). https://doi.org/10.1609/aaai.v27i1.8541.

Zhou, D., Wang, Z., Bandyopadhyay, S., & Schwager, M. (2017). Fast, on-line collision avoidance for dynamic vehicles using buffered Voronoi cells. *IEEE Robotics and Automation Letters, 2*(2), 1047–1054. https://doi.org/10.1109/LRA.2017.2656241

Zhou, Y., & Zeng, J. (2015). Massively parallel A* search on a GPU. In *Proceedings of the AAAI conference on artificial intelligence (vol. 29)*. https://doi.org/10.1609/aaai.v29i1.9367.

Wolfgang Hönig is an independent junior research group leader at TU Berlin, Germany heading the Intelligent Multi-Robot Coordination Lab. Previously, he was a postdoctoral scholar at the California Institute of Technology, USA. He received the diploma in Computer Science from TU Dresden, Germany in 2012, and the M.S. and Ph.D. degrees from the University of Southern California (USC), USA in 2016 and 2019, respectively. His research focuses on enabling large teams of physical robots to collaboratively solve real-world tasks, using tools from informed search, optimization, and machine learning. Dr. Hönig has been the recipient of several awards, including Best Paper in Robotics Track for a paper at ICAPS 2016 and the 2019 Best Dissertation Award in Computer Science at USC.

Nora Ayanian received the B.S. degree in Mechanical Engineering and Mechanics in 2005 from Drexel University, Philadelphia, PA, and the M.S. and Ph.D. degrees in Mechanical Engineering and Applied Mechanics from the University of Pennsylvania, in 2008 and 2011, respectively. She was a Postdoctoral Associate at the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology from 2011–2013. In 2013, she joined the Department of Computer Science, University of Southern California, as an Assistant Professor, and in 2020, became Associate Professor. Since January 2022, she is an Associate Professor with the Department of Computer Science and the School of Engineering at Brown University. Her research interests include multi-robot planning, coordination, and control. Dr. Ayanian is a founder and advisory chair of the the IEEE Technical Committee on Multi-Robot Systems. She is the recipient of several awards, including a being named to the MIT TR35, Okawa Foundation Research Award, and the USC Hanna Reisler Mentorship award.

... tems Lab (RESL). His research interests include multi-robot navigation systems, multi-robot trajectory planning, and multi-robot aware control.

Baskın Şenbaşlar received his B.S. degree in Computer Engineering in 2017 from Middle East Technical University, Ankara, Turkey. He received his M.S. degree in Computer Science in 2019 from University of Southern California, Los Angeles, USA as a Fulbright Grantee. In 2019, he started his PhD. studies in University of Southern California as a part of Automatic Coordination of Teams (ACT) Lab. In 2022, he moved to the Robotics and Embedded Sys-