Received December 5, 2018, accepted December 19, 2018, date of publication December 24, 2018, date of current version January 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2889533

Fast Generation of Collision-Free Trajectories for Robot Swarms Using GPU Acceleration

MICHAEL HAMER, (Member, IEEE), LINO WIDMER, AND RAFFAELLO D'ANDREA, (Fellow, IEEE)
Institute for Dynamic Systems and Control, ETH Zurich, 8092 Zurich, Switzerland
Corresponding author: Michael Hamer (mike@mikehamer.info)

ABSTRACT As the capabilities of robots and their control systems improve, we see an increasing number of use cases where the simultaneous operation of robots within a space is advantageous. Although trajectories for individual robots can be computed quickly using the existing methods, when robots operate simultaneously and in close proximity, the requirement for collision avoidance introduces a coupling between robot trajectories and makes the trajectory generation problem difficult to solve quickly. In this paper, we propose a parallelizable formulation of such problems and a method for solving them quickly on modern graphics processing units, using momentum-based gradient descent. We demonstrate the proposed framework in simulation using two case studies: a swarm of 200 quadcopters traversing a maze and a fleet of 100 bicycle robots changing their formation. In both the cases, our method requires just seconds to generate feasible, collision-free trajectories for the entire swarm.

INDEX TERMS Collision avoidance, motion planning, robot control, trajectory optimization.

I. INTRODUCTION

We consider a swarm of robots working simultaneously and in close proximity, and where each robot in the swarm is tasked with transitioning from its given initial state to a given goal state without colliding and while satisfying other constraints placed on the trajectories. Using existing methods, feasible trajectories that transition each robot from its initial state to its goal state can quickly be generated. However, when robots operate simultaneously and in close proximity, the requirement for collision avoidance introduces a coupling between the trajectories of individual robots and makes the problem non-convex and difficult to solve in a time-efficient manner.

In this paper we propose a parallelizable formulation of such problems, as well as a method for solving such problems efficiently on modern tensor or graphics processing units (GPUs). We initialize the trajectory for each robot independently without considering inter-robot collisions, and then use momentum-based gradient descent to iteratively improve robot trajectories until feasibility. Given the non-convexity of the problem and the usage of gradient descent, our method yields solutions in the local neighborhood of the initialization, and is not guaranteed to find an optimal solution; however, our primary concern and the primary focus of this paper is on quickly generating feasible and objectively "good" trajectories, which is itself not an easy task given the number of robots interacting.

As we later demonstrate, our method requires just seconds to generate collision-free, dynamically feasible trajectories for hundreds of robots operating in close proximity and within a densely-cluttered environment. These trajectories are fully-defined state and input trajectories, and we assume that these nominal trajectories are provided as reference to and are tracked by a controller on each robot.

After reviewing related literature in Section II, we present a general formulation of our method in Section III, before discussing two case studies in Section IV:

1) In Section IV-A, we solve the "Sort 200" quadcopter maze-traversal benchmark problem [1] (see Fig. 1)
2) In Section IV-B, we address a ground robot transition problem using a fleet of ground robots with bicycle dynamics (see Fig. 2).

We conclude the paper in Section V with a discussion of the method's current limitations and possibilities for future research.

II. RELATED WORK

The method presented in this paper builds upon prior work in trajectory generation for individual robots, optimization of

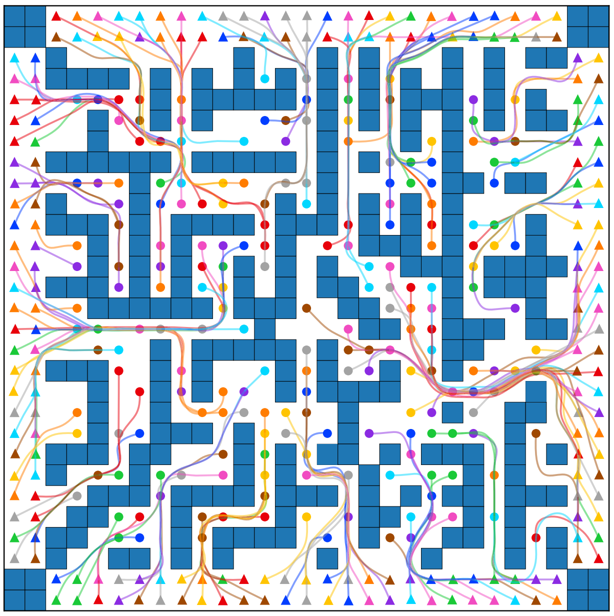


FIGURE 1. An example of the type of problem that we address in this paper. In this benchmark example, 200 quadcopters are tasked with finding their way out of a maze without colliding with each other or with the maze. Our method takes roughly 2.3 seconds to generate feasible, collision-free trajectories for all 200 quadcopters. Further details to this example are presented in Section IV-A.

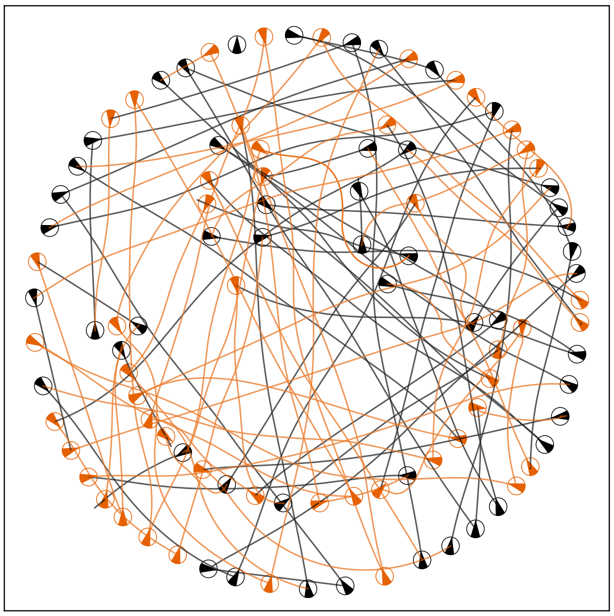


FIGURE 2. A further example of the type of problem addressed in this paper. In this example, 100 robots with bicycle dynamics (e.g. cars, warehouse robots, etc.) are tasked with exchanging positions in a "smiley face" formation without colliding. To exemplify the application of our method to heterogeneous fleets of robots, we constrain the steering angle of 50 robots (colored black) to 20°, and the steering angle of the other 50 robots (colored orange) to 70°. As elaborated upon in Section IV-B, our method takes roughly 1.6 seconds to generate feasible, collision-free trajectories for the 100 robots.

swarm trajectories, and the application of GPUs to the evaluation of robot trajectories and constraints.

A. ROBOT TRAJECTORY GENERATION

Trajectory generation for individual robots is a well established field of research, with high-performance algorithms existing for most classes of robots. Many methods of trajectory generation express trajectories as the piecewise connection of basis functions. In [2], for example, trajectories are described using a piecewise constant jerk and generated for time-optimality using a bang-bang approach. In [3]–[6], trajectories are represented as polynomials generated to yield minimum jerk or minimum snap motions.

In cluttered environments, collisions with obstacles often prevent the direct application of the above methods. In such situations, graph-search methods can be used to plan trajectories in a discretized state- or action-space [7], [8]; or sampling based methods, for example Rapidly-exploring Random Trees (RRT) [9], RRT* [10] or Probabilistic Roadmaps (PRM) [11], can be used to find a feasible path through a continuous space.

Such search-based methods scale poorly with the dimensionality of the space and are thus often used to find feasible paths through a lower-dimensional space, before smoothing trajectories (for example, piecewise polynomials or splines) are fit through the sequence of waypoints that define the path [12], [13].

Since our method uses gradient descent, the convergence speed and overall quality of the solution is very dependent on the quality of the initialization. Which approach is best used to initialize individual trajectories is highly problem specific. In the case study presented in Section IV-A, we show an example where graph search in a discretized environment is used to find a feasible path through a cluttered environment before splines are used to smooth this path and initialize the robots' trajectories. In the second case study presented in Section IV-B, we initialize robot trajectories using fifth-order polynomials optimized to minimize trajectory jerk.

B. SWARM TRAJECTORY GENERATION & COLLISION AVOIDANCE

Generating collision-free trajectories for robot swarms has historically proven to be a highly combinatorial and high dimensional problem. Previous approaches have used convex approximations or reformulations of the problem to improve computational tractability. Examples of such convex-optimization-based approaches include [14], who solve the problem by iteratively solving mixed-integer linear programs; [13] and [15]–[18], who employ sequential convex programming to iteratively refine the solution towards feasibility; [19], who formulate and solve the problem as a mixed-integer quadratic program; and [20], who solve the problem using a variation of the alternating direction method of multipliers.

The application of these aforementioned methods to large swarms is limited by their computational complexity. This is addressed in [21], who demonstrate that a significant increase in computation speed can be achieved by only including collision constraints for pairs of robots in each other's vicinity, and by formulating the problem such that robot trajectories can be optimized in parallel. A further improvement in computational speed can by achieved by optimizing both trajectory and robot assignment in parallel, leveraging the fact that an optimal assignment of robots to goals will require far fewer collisions to be avoided, than a non-optimal assignment. This property is exploited by Agarwal and Akella [22], who reformulate the optimization problem to be solved as a linear sum assignment problem and apply their method to swarms of hundreds of robots; and by Preiss *et al.* [1], Debord *et al.* [23], and Hönig *et al.* [24], who iterate a search-based roadmap planner with a trajectory smoothing step until feasible robot trajectories are found, and who demonstrate their method by planning collision-free trajectories for hundreds of robots through densely cluttered environments.

The approach we present in this paper builds upon this existing work, and allows trajectories to be initialized using a large range of existing methods, depending on what is suitable for the given problem domain. This paper advances the current state of the art by allowing the non-convex trajectory generation problem to be solved directly on the GPU using gradient descent, allowing our algorithm to leverage the computational power of modern GPUs.

C. GPU-BASED TRAJECTORY GENERATION

The use of GPUs for accelerating computation is widespread in many fields; however, the potential application to robot trajectory generation problems remains largely unexplored.

The majority of papers to date use the GPU to parallelize the search for feasible trajectories for a single robot. Approaches include, for example, variations of genetic algorithms [25]–[29]; parallel implementations of PRM [30], [31]; and a parallel implementation of R* search [32], [33].

More in line with the approach of this paper are [34] and [35], within which the GPU is used to parallelize the evaluation of dynamics and collision constraints. Both papers, however, only deal with trajectories for a single robot. The non-convex trajectory generation problem is solved in [34] directly; however in this case the GPU is used only for parallel constraint evaluation with results then transferred to a CPU-based solver.

Our paper advances the aforementioned in both the intended problem domain: swarms of hundreds of robots; and in our approach: we employ standard methods for the initialization of feasible trajectories for individual robots within the swarm, utilize the GPU for parallel evaluation of constraints, and iteratively solve the non-convex trajectory generation problem using gradient descent on the GPU.

III. PROBLEM FORMULATION

In this paper, we address the problem of generating feasible, collision-free trajectories for robots operating simultaneously and in close proximity. We initialize each robot's trajectory independently without considering inter-robot collisions, and we model constraints on the trajectories (e.g. state and collision constraints) as soft constraints and include them in an objective function. We then employ momentum-based gradient descent to iteratively improve robot trajectories until all constraints are satisfied. Given the non-convexity of the problem and the usage of gradient descent, our method is not guaranteed to find an optimal solution, rather yielding feasible solutions in the local neighborhood of the initialization. We assume that the reference state and input trajectories generated by our method are tracked by each robot's controller.

In this section, we formalize the above problem, introduce notation, outline assumptions and requirements, and present the method formally in Algorithm 1.

A. ROBOT STATES AND INPUTS

Consider a swarm of R robots. We number these robots sequentially from 1 to R, and use indices i and j to refer to robots within the swarm. For simplicity of the following explanation, we assume that all robots in the swarm are identical; however, this method can be trivially extended for swarms of different robots, an example of which is shown in the case study presented in Section IV-B. We let each of the R robots have an input space of dimension M, and a state space of dimension N. We assume inputs are issued to each robot at discrete times $k = 0, \dots, K-1$, and that inputs are held constant between time instants. We denote robot i's input trajectory by $U_i \in \mathbb{R}^{K\times M}$, and it's state trajectory by $S_i \in \mathbb{R}^{(K+1)\times N}$. We require that each robot's initial state $S_i[0]$ and desired goal state $G_i \in \mathbb{R}^N$ are known and feasible. Our method generates a feasible and collision-free trajectory for each robot, which transitions it from its initial state to within a user-defined threshold of its goal state (e.g. a 5 cm final position tolerance is used in the case studies in Section IV).

B. ROBOT DYNAMICS

Robot i's input trajectory is mapped to its state trajectory through the known robot dynamics $f(\cdot,\cdot)$ as

$$S_i[k+1] = f(U_i[k],\ S_i[k]) \quad \text{for all } k \in [0,\ K-1]. \tag{1}$$

Iterating f across the entire input trajectory, we write

$$S_i = \mathcal{F}(U_i,\ S_i[0]) \tag{2}$$

to express the relationship between robot i's input and state trajectory.

For notational simplicity, we stack the input and state trajectories of all robots into the tensors $\boldsymbol{U} \in \mathbb{R}^{K\times M\times R}$ and $\boldsymbol{S} \in \mathbb{R}^{(K+1)\times N\times R}$ respectively. This is not always possible in the case of heterogenous swarms; however, in such cases, operations can be performed on trajectories individually, while incurring minimal computational overhead. This stacking is therefore without loss of generality. We overload nota-

Algorithm 1 Our Method Uses Momentum-Based Gradient Descent to Iteratively Improve Robot Input Trajectories $\boldsymbol{U}$ Until the Corresponding State Trajectories $\boldsymbol{S}$ Are Collision-Free and Feasible With Respect to the Given Constraints

Require:

- Initialization of robot input trajectories $\boldsymbol{U}$, which are feasible for individual robots if robot collisions are ignored
- Initial state of each robot $\boldsymbol{S}[0]$
- Robot dynamics $\mathcal{F}(\cdot,\ \cdot)$ which maps the robots' inputs and initial states to their state trajectories as in (3)
- Set of constraints $\mathscr{C}$, where each constraint $c \in \mathscr{C}$ is defined by a loss function $\mathcal{L}_c(\cdot,\ \cdot)$ and satisfaction check $\mathcal{S}_c(\cdot,\ \cdot)$
- Weighting parameter w_c for each constraint $c \in \mathscr{C}$
- Gradient descent optimizer $\mathrm{OPT}(\cdot, \cdot \mid \eta)$, parameterized by a given learning rate η and which takes a gradient and momentum state, and returns a step and updated momentum.

Optimization Procedure:

```
 1: Reset optimizer momentum $\boldsymbol{\rho}$
 2: loop
        Calculate robot state trajectories as in (3)
 3:     $\boldsymbol{S} \leftarrow \mathcal{F}(\boldsymbol{U}, \boldsymbol{S}[0])$
        Calculate loss gradient as in (5)
 4:     $\nabla_{\boldsymbol{U}}\mathcal{L} \leftarrow \sum_c w_c \left(\frac{\partial \mathcal{L}_c}{\partial \boldsymbol{U}} + \frac{\partial \mathcal{L}_c}{\partial \boldsymbol{S}}\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{U}}\right)(\boldsymbol{U},\ \boldsymbol{S})$
        Check constraint satisfaction
 5:     if $\mathcal{S}_c(\boldsymbol{U},\ \boldsymbol{S})$ is false for any $c \in \mathscr{C}$ then
            Run optimizer
 6:         $\Delta\boldsymbol{U},\ \boldsymbol{\rho} \leftarrow \mathrm{OPT}(\nabla_{\boldsymbol{U}}\mathcal{L},\ \boldsymbol{\rho} \mid \eta)$
            Update robot input trajectories
 7:         $\boldsymbol{U} \leftarrow \boldsymbol{U} + \Delta\boldsymbol{U}$
 8:     else
 9:         return $\boldsymbol{U}$, $\boldsymbol{S}$
10:     end if
11: end loop
```

Returns: leftmargin=17pt, labelsep=7pt

- Input trajectories $\boldsymbol{U}$ that are feasible with respect to all input constraints
- State trajectories $\boldsymbol{S}$ that are collision free and feasible with respect to all state constraints

tion to write the relationship between the input and state trajectories of the swarm as

$$\boldsymbol{S} = \mathcal{F}(\boldsymbol{U},\ \boldsymbol{S}[0]), \tag{3}$$

and note that $\boldsymbol{S}$ is a function of both the robots' initial states $\boldsymbol{S}[0]$ and the robots' input trajectories $\boldsymbol{U}$. Hidden behind this notation are a number of performance caveats, which we discuss in Section III-F.

C. CONSTRAINTS & OPTIMIZATION OBJECTIVE

The robots' input and state trajectories are subject to constraints, which can include inter-robot and environmental collision constraints; constraints on the final state (e.g. goal constraints); bounds on the state trajectory (e.g. minimum and maximum velocity); as well as actuator limits (e.g. minimum and maximum input). Our general formulation also allows for coupled input and state constraints (e.g. state-dependent input bounds), or for constraints which are only active on specific robots.

We model these constraints as soft constraints and denote by $\mathscr{C}$ the set of all constraints. We model each constraint $c \in \mathscr{C}$ as consisting of two components:

1) A loss function $\mathcal{L}_c(\boldsymbol{U},\ \boldsymbol{S})$, which is included in the optimization objective function (detailed below) and whose negative gradient provides a direction towards a feasible solution.
2) A satisfaction function $\mathcal{S}_c(\boldsymbol{U},\ \boldsymbol{S})$, whose boolean output indicates whether the constraint c is satisfied. If $\mathcal{S}_c(\boldsymbol{U},\ \boldsymbol{S})$ is true for all $c \in \mathscr{C}$, trajectories are deemed feasible and the optimization terminates.

Weighting each loss by a parameter w_c, the optimization objective is to minimize

$$\mathcal{L}(\boldsymbol{U},\ \boldsymbol{S}) := \sum_c w_c \mathcal{L}_c(\boldsymbol{U},\ \boldsymbol{S}) \tag{4}$$

with respect to the input trajectories $\boldsymbol{U}$. We achieve this by descending the loss gradient

$$\nabla_{\boldsymbol{U}}\mathcal{L}(\boldsymbol{U},\ \boldsymbol{S}) := \sum_c w_c \left(\frac{\partial \mathcal{L}_c}{\partial \boldsymbol{U}} + \frac{\partial \mathcal{L}_c}{\partial \boldsymbol{S}} \cdot \frac{\partial \boldsymbol{S}}{\partial \boldsymbol{U}}\right)(\boldsymbol{U},\ \boldsymbol{S}), \tag{5}$$

where we recall that $\boldsymbol{S}$ is a function of $\boldsymbol{U}$. This gradient calculation and back-propagation can be automated using the auto-differentiation functionality of tensor arithmetic libraries such as Tensorflow [36] or PyTorch [37], or can be computed manually. In the case studies presented in Section IV, we show the implementation of a number of common constraints.

D. OPTIMIZATION

Since our method uses gradient descent, the speed of the algorithm and quality of the solution are dependent on the quality of the initialization. Robot input trajectories should therefore be initialized to transition the robot from its initial state to its goal state, while satisfying individual robot constraints, and in a manner that is appropriate for the problem (e.g. time optimal, minimum jerk, etc.). This initialization is performed for each robot independently and without considering inter-robot collisions.

The requirement for collision-avoidance between robots introduces a coupling between the trajectories of individual robots and makes the problem non-trivial to solve. Using a (potentially momentum-based) gradient descent optimizer, for example Adam [38], our method iteratively improves the input tensor $\boldsymbol{U}$ by descending the loss function's gradient (5). When trajectories are found to satisfy all constraints ($\mathcal{S}_c(\boldsymbol{U}, \boldsymbol{S})$ is true for all $c \in \mathscr{C}$) the algorithm terminates. This optimization procedure is formalized in Algorithm 1.

E. HYPERPARAMETER TUNING

With respect to (5), we observe that the gradient and therefore the direction and size of the gradient descent update are dependent on the set of constraint weighting factors $\{w_c \mid c \in \mathscr{C}\}$. In addition, the size of each update step is proportional to the so-called ''learning rate'' of the gradient descent optimizer. The learning rate, together with the constraint weights are the hyperparameters of our method.

The selection of appropriate hyperparameters is specific to the class of problem being addressed. Selecting appropriate hyperparameters is critical to achieving fast convergence to reasonable results. Hyperparameters should be chosen such that the number of collisions is quickly reduced, while ensuring that other constraints which may be violated during the optimization process are quickly reoptimized to feasibility. To select hyperparameters for the problem classes presented in Section IV, we employed an automated hyperparameter search [39]. This search involves running hundreds of thousands of simulations with different hyperparameter values, and selecting those hyperparameters that minimize the 90th-percentile convergence speed of the algorithm.

The effects of hyperparameter tuning are investigated in more detail in Section IV-B5.

F. PERFORMANCE CONSIDERATIONS

The speed of our method comes from the ability to parallelize many of the algorithm's steps and thus leverage the computational power of a modern GPU. With reference to Algorithm 1, the two major steps of the algorithm are the calculation of state trajectory (line 3), and the calculation of a loss gradient (line 4). It is important to ensure that these steps are implemented in such a way as to enable their effective parallelization.

1) CALCULATION OF STATE TRAJECTORY

When computing the state trajectory of each robot from its input trajectory, it is important to note that we define the dynamics $\mathcal{F}(\boldsymbol{U}, \boldsymbol{S}[0])$ to operate on the input trajectory as a whole, rather than iterating the dynamics across time (as in (1)).

Significant performance gains can be realized when the dynamics are implemented using cumulative sums or cumulative products, since these cumulative operations can be efficiently parallelized to run in logarithmic time [40]. In the case studies presented in Section IV, we show two examples of how robot dynamics expressed as standard difference equations can be implemented using cumulative sums.

2) CALCULATION OF LOSS GRADIENT

Losses are typically independent across time and across robots, and can be effectively implemented using tensor operations, for example as provided by libraries such as Tensorflow [36] or PyTorch [37]. Tensor operations are inherently parallelizable and thus enable the evaluation of losses in parallel. Likewise, the evaluation of the loss gradient follows through back-propagation, which is based on tensor multiplications and additions and is thus also inherently parallelizable.

IV. CASE STUDIES

In this section, we demonstrate the application of our method to two different scenarios. An example implementation of each scenario is available at http://mikehamer.info/swarm-trajectories.

Results presented in this section were generated using an Nvidia GeForce GTX 1080 Ti GPU, a widely-available consumer GPU costing roughly $700 at the time of writing. Implementations were programmed using Python 3.6, PyTorch version 0.4.1, CUDA version 9.2.148 and cuDNN version 7.1.4. Tensor arithmetic was performed using 32-bit floating point. We note that the throughput could be more than doubled if 16-bit floating point were used on a more recent GPU. We also note that the GPU utilization was between 20% and 40% in all scenarios, implying that further performance may be achievable through a more optimized implementation.

A. ''SORT200'' QUADCOPTER MAZE BENCHMARK

In the ''Sort 200'' benchmark scenario [1], 200 quadcopter robots begin at random (x, y) locations within a maze and must fly to a goal location outside the maze without colliding. Initial and goal positions all lie on the $z = 0$ plane. As in the original benchmark scenario, quadcopters are allowed to move in the z dimension if required to avoid collisions.

An example of the Sort 200 scenario is shown in Fig. 1, where the initial and final positions of 200 quadcopters are shown connected by their collision-free trajectories.

1) ROBOT MODELING

The dynamic model of a quadcopter is differentially flat [41], a property which is often exploited to allow trajectories to be planned in each of the inertial axes independently (see, e.g. [2]–[5]). We take a similar approach and plan jerk trajectories for each quadcopter in each axis independently. These trajectories can at a later stage be converted to nominal thrust and body-rate inputs, or could be provided to a trajectory-tracking controller.

Denoting the three inertial axes with subscripts x, y and z, we respectively denote the position, velocity, acceleration and jerk of the quadcopter in the inertial frame using

$$\mathrm{p} := (\mathrm{p}_x, \mathrm{p}_y, \mathrm{p}_z), \tag{6}$$

$$\mathrm{v} := (\mathrm{v}_x, \mathrm{v}_y, \mathrm{v}_z), \tag{7}$$

$$\mathrm{a} := (\mathrm{a}_x, \mathrm{a}_y, \mathrm{a}_z), \quad \text{and} \tag{8}$$

$$\mathrm{j} := (\mathrm{j}_x, \mathrm{j}_y, \mathrm{j}_z). \tag{9}$$

We define the $M = 3$ dimensional input space of a quadcopter as

$$U := (\mathrm{j}_x, \mathrm{j}_y, \mathrm{j}_z) \tag{10}$$

and its $N = 9$ dimensional state as

$$S := (\mathrm{p}_x, \mathrm{p}_y, \mathrm{p}_z, \mathrm{v}_x, \mathrm{v}_y, \mathrm{v}_z, \mathrm{a}_x, \mathrm{a}_y, \mathrm{a}_z). \tag{11}$$

Discretizing using the Euler forward method for a sampling period of $T = 50$ ms, the quadcopter's state in the inertial frame evolves as

$$\mathrm{a}[k+1] = \mathrm{a}[k] + T\,\mathrm{j}[k] \tag{12}$$
$$\mathrm{v}[k+1] = \mathrm{v}[k] + T\,\mathrm{a}[k] + \tfrac{1}{2}T^2\mathrm{j}[k] \tag{13}$$
$$\mathrm{p}[k+1] = \mathrm{p}[k] + T\,\mathrm{v}[k] + \tfrac{1}{2}T^2\mathrm{a}[k] + \tfrac{1}{6}T^3\mathrm{j}[k], \tag{14}$$

where we assume that the input $\mathrm{j}[k]$ is held constant during timestep k. These recursive difference equations can be rewritten to express the state at time $k + 1$ in terms of the input and state history:

$$\mathrm{a}[k+1] = \mathrm{a}[0] + T\sum_{\kappa=0}^{k}\mathrm{j}[\kappa] \tag{15}$$
$$\mathrm{v}[k+1] = \mathrm{v}[0] + T\sum_{\kappa=0}^{k}\mathrm{a}[\kappa] + \tfrac{1}{2}T^2\sum_{\kappa=0}^{k}\mathrm{j}[\kappa] \tag{16}$$
$$\mathrm{p}[k+1] = \mathrm{p}[0]+T\sum_{\kappa=0}^{k}\mathrm{v}[\kappa] + \tfrac{1}{2}T^2\sum_{\kappa=0}^{k}\mathrm{a}[\kappa]+\tfrac{1}{6}T^3\sum_{\kappa=0}^{k}\mathrm{j}[\kappa]. \tag{17}$$

Lifting the above in time, we observe that the state trajectory can be computed using cumulative summation (denoted CS):

$$\begin{aligned}\mathrm{a}[1{:}K] &= \mathrm{a}[0] + T\cdot\mathrm{CS}(\mathrm{j})\\ \mathrm{v}[1{:}K] &= \mathrm{v}[0] + T\cdot\mathrm{CS}(\mathrm{a}[0{:}K-1]) + \tfrac{1}{2}T^2\cdot\mathrm{CS}(\mathrm{j})\\ \mathrm{p}[1{:}K] &= \mathrm{p}[0] + T\cdot\mathrm{CS}(\mathrm{v}[0{:}K-1])\\ &\quad + \tfrac{1}{2}T^2\cdot\mathrm{CS}(\mathrm{a}[0{:}K-1]) + \tfrac{1}{6}T^3\cdot\mathrm{CS}(\mathrm{j}).\end{aligned} \tag{18}$$

This implementation can be effectively parallelized to run in logarithmic time, which in this case-study executes an order of magnitude faster than the linear-time, recursive implementation.

2) ASSIGNMENT OF ROBOTS TO GOALS

Prior to trajectory generation, the 200 quadcopters must be assigned one of 200 goal positions located on the perimeter of the maze. We begin by transforming the given 2D maze into a graph representation by discretizing the graph coordinates and connecting points with an unobstructed line of sight using an edge with weight equal to the points' Euclidean distance. The path length between all (discretized) positions within the maze can now be computed using, for example, Dijkstra's algorithm [42]. This step is only required if the maze changes.

Using these computed path lengths and the known initial quadcopter positions, we use the Hungarian method [43] to compute an allocation of quadcopters to goal positions that minimizes the sum of squared distance traveled by the swarm.

3) INITIALIZATION OF INDIVIDUAL TRAJECTORIES

Once the allocation of initial positions to goal positions is known, we fit splines to the shortest path connecting these positions (as given by Dijkstra's algorithm in the previous step). As in the original benchmark problem, we give quadcopters 10 seconds to exit the maze. Splines are scaled to have this duration, and are then sampled at the desired rate (in our case 20 Hz) to yield the initial trajectories for each quadcopter.

The results of this stage are 200 jerk trajectories in x and y, which quickly and smoothly transition robots from an initial to a goal state. We initialize the z component of each trajectory to zero. At this stage, quadcopters are treated independently and as such the initial trajectories will cause collisions between quadcopters. This and the previous step are computed on the CPU due to the availability of centralized solvers for Dijkstra's algorithm and the Hungarian method.

4) CONSTRAINTS

Our constraint set consists of constraints based on the quadcopters' physical capabilities, constraints on quadcopter collisions with each other and with the maze walls, and constraints on the final state of each quadcopter. Based on these constraints, we define the loss function to be

$$\begin{aligned}\mathcal{L}(\boldsymbol{U},\boldsymbol{S}) := {} & w_{\text{thrust}}\,\mathcal{L}_{\text{thrust}}(\boldsymbol{U},\boldsymbol{S})\\ &+ w_{\text{body-rate}}\,\mathcal{L}_{\text{body-rate}}(\boldsymbol{U},\boldsymbol{S})\\ &+ w_{\text{quad-collision}}\,\mathcal{L}_{\text{quad-collision}}(\boldsymbol{U},\boldsymbol{S})\\ &+ w_{\text{maze-collision}}\,\mathcal{L}_{\text{maze-collision}}(\boldsymbol{U},\boldsymbol{S})\\ &+ w_{\text{goal-pos}}\,\mathcal{L}_{\text{goal-pos}}(\boldsymbol{U},\boldsymbol{S})\\ &+ w_{\text{goal-vel}}\,\mathcal{L}_{\text{goal-vel}}(\boldsymbol{U},\boldsymbol{S}).\end{aligned} \tag{19}$$

In the following, we define and discuss the implementation of these constraints and their respective loss functions.

a: *DYNAMICS CONSTRAINTS*

In line with [1], we set the quadcopter dynamic limits based on a Crazyflie 2.0 quadcopter [44]. These limits include the minimum and maximum mass-normalized thrusts $\mathrm{f}_{\min} = 5\,\mathrm{m\,s^{-2}}$ and $\mathrm{f}_{\max} = 15\,\mathrm{m\,s^{-2}}$, and the maximum tilting body-rate $\omega_{\max} = 30\,\mathrm{rad\,s^{-1}}$ [45].

Letting g denote the vector of gravitational acceleration, quadcopter i's thrust at time step k is

$$\mathrm{f}_i[k] = \|\mathrm{a}_i[k] - g\|_2, \tag{20}$$

which we constrain to the feasible range using the loss function

$$\begin{aligned}\mathcal{L}_{\text{thrust}}(\boldsymbol{U},\boldsymbol{S}) := {} & \sum_{k=1}^{K}\sum_{i=1}^{R}\max\{0,\ \mathrm{f}_i[k]-\mathrm{f}_{\max}\}\\ &+ \sum_{k=1}^{K}\sum_{i=1}^{R}\max\{0,\ \mathrm{f}_{\min}-\mathrm{f}_i[k]\}.\end{aligned} \tag{21}$$

As shown in [5], the magnitude of quadcopter i's tilting body-rates $\omega_{x,i}[k]$ and $\omega_{y,i}[k]$ at timestep k can be upper-bounded by a function of its jerk and thrust as:

$$\sqrt{\omega_{x,i}[k]^2 + \omega_{y,i}[k]^2} \leq \frac{\|\mathrm{j}_i[k]\|_2}{\mathrm{f}_i[k]}. \tag{22}$$

We use this upper bound to constrain the quadcopters' tilting body-rates using the loss function

$$\mathcal{L}_{\text{body-rate}}(\boldsymbol{U}, \boldsymbol{S}) := \sum_{k=0}^{K-1} \sum_{i=1}^{R} \max\left\{0,\ \frac{\|\mathrm{j}_i[k]\|_2}{\mathrm{f}_i[k]} - \omega_{\max}\right\}. \tag{23}$$

A quadcopter's thrust and body-rates are defined to satisfy their respective constraints if the associated loss function is zero.

b: QUADROCOPER COLLISION CONSTRAINTS

We define two quadcopters as colliding if their centers are within a certain distance D, which we refer to as the collision distance and define as $D := 0.25$ m in line with [1]. We define the collision loss function as

$$\mathcal{L}_{\text{collision}}(\boldsymbol{U}, \boldsymbol{S}) := \sum_{k=1}^{K} \sum_{i=1}^{R} \sum_{j=i+1}^{R} 2 \cdot \max\left\{0,\ D - d_{ij}[k]\right\}, \tag{24}$$

where $d_{ij}[k] := \|\mathrm{p}_i[k] - \mathrm{p}_j[k]\|_2$ is the center-to-center distance between quadcopters i and j at timestep k. Note that due to the pairwise nature of collisions, we only check quadcopters with a higher index and we therefore add twice the collision loss.

Checking for robot-robot collisions is one of the most time consuming steps in the optimization pipeline. Directly computing the above loss requires that for each step in time, the positions of all robots are checked against the positions of all other robots to determine whether the robots collide. As noted in [21], the time required for this step can be significantly reduced if pairs of quadcopters are only checked for a collision if they are closer than a given threshold.

We take a similar approach in Algorithm 2 and leverage the time-efficiency of parallel sorting to order quadcopter positions according to their position $\mathrm{p}_{\alpha,i}$ in the axis $\alpha \in \{x, y, z\}$. It is then possible to compare each robot only with those robots having similar α coordinates. By choosing α to be an axis with a large position variance (e.g. in this case study x or y are good choices), we significantly reduce the number of checks required.

c: MAZE COLLISION CONSTRAINTS

We implement the maze collision loss function as a two-dimensional lookup. For every quadcopter and at every timestep we check the robot's x and y position to determine whether it collides with the maze boundaries. If a collision is found to occur, that is if the quadcopter's position is closer to a maze wall than its collision distance allows, a loss with a gradient normal to the boundary is added to the global loss.

Algorithm 2 This Algorithm Drastically Improves the Speed of Collision Checks by Leveraging the Efficiency of Parallel Sorting Algorithms. Robot Positions Are Sorted Along an Axis α With High Position Variance, Thus Enabling Collision Checks to be Performed in the Robot's Local Neighborhood Rather Than Across All Robots in the Swarm. This Algorithm Implements the Loss Function Given in (24)

Require:
- Robot positions $\mathrm{p}_i[k] := (\mathrm{p}_{x,i}[k],\ \mathrm{p}_{y,i}[k],\ \mathrm{p}_{z,i}[k])$
- Axis $\alpha \in \{x, y, z\}$ with large position variance
- Minimum allowed center-to-center distance D

Robot collision detection procedure:

```
 1: for $k \in [1, K]$ in parallel do
        Order robots at time k by their position in axis $\alpha$
 2:     $\mathrm{Idx}[k, :] \leftarrow \mathrm{ParallelArgsort}(\{\mathrm{p}_{\alpha,i}[k] \mid i \in [1, R]\})$
 3: end parallel for
    In parallel, check collisions for all robots at all timesteps
 4: for $i \in [1, R]$ and $k \in [1, K]$ in parallel do
        Sequentially (to allow early stopping), check for
        collisions for all pairs with the current robot i
 5:     for $j \in [i+1, R]$ sequentially do
            Indexes of robot pair
 6:         $r_i \leftarrow \mathrm{Idx}[k, i]$; $r_j \leftarrow \mathrm{Idx}[k, j]$
 7:         if $\mathrm{p}_{\alpha,r_j}[k] - \mathrm{p}_{\alpha,r_i}[k] > D$ then
                Since $p_{\alpha,r_{j+1}}[k] \geq p_{\alpha,r_j}[k]$ we can stop early
 8:             break
 9:         end if
            Compute pairwise distance
10:         $d \leftarrow \|\mathrm{p}_{r_j} - \mathrm{p}_{r_i}\|_2$
            Compute loss and loss gradients
11:         if $d \leq D$ then
12:             atomic $\{ \mathcal{L}_{\text{collision}} \leftarrow \mathcal{L}_{\text{collision}} + 2 \cdot (D - d) \}$
13:             atomic $\{ \nabla_{\mathrm{p}_{r_i}} \mathcal{L} \leftarrow \nabla_{\mathrm{p}_{r_i}} \mathcal{L} + (\mathrm{p}_{r_j} - \mathrm{p}_{r_i})/d \}$
14:             atomic $\{ \nabla_{\mathrm{p}_{r_j}} \mathcal{L} \leftarrow \nabla_{\mathrm{p}_{r_j}} \mathcal{L} - (\mathrm{p}_{r_j} - \mathrm{p}_{r_i})/d \}$
15:         end if
16:     end sequential for
17: end parallel for
18: return $\mathcal{L}_{\text{collision}}$, $\{\nabla_{\mathrm{p}_i} \mathcal{L} \mid i \in [1, R]\}$
```

Returns: leftmargin=17pt, labelsep=7pt
- Collision loss $\mathcal{L}_{\text{collision}}$
- Collision loss gradient $\nabla_{\mathrm{p}_i} \mathcal{L}$ for each robot i

We find it useful to distinguish between the case of a quadcopter's trajectory traveling too close to a wall, and the case of the trajectory moving into a wall. During the optimization process, quadcopter state trajectories are perturbed in order to avoid collisions and to satisfy other constraints. This is an iterative process, and satisfying one constraint may require temporarily violating another constraint during an intermediate iteration of the optimization. Such temporary constraint violations are resolved in later iterations. As an example, it is often unavoidable that in trying to avoid a collision with another quadcopter, a quadcopter's trajectory is temporarily perturbed to travel too close to a wall. This situation can be resolved in later iterations; however, if a quadcopter's trajectory is perturbed so significantly as to enter a wall, it is possible that the gradient descent optimization will force the trajectory through the wall, and will thus optimize towards an inescapable and infeasible local minimum. For this reason, we lightly penalize being too close to a wall, while applying significantly more penalty to trajectory steps that enter a wall.

d: GOAL CONSTRAINTS

As previously mentioned, during individual iterations of the optimization, satisfying one constraint might require temporarily violating another. As a further example of this, perturbing trajectories to avoid collisions often results in trajectories that no longer end at the desired goal state. In order to ensure the final feasibility of the trajectories the deviation of the final state from the goal state must be penalized. In this case study, we penalize the squared deviation of the final position and final velocity from the desired values ($G_{\mathrm{p},i}$ and 0 respectively) using the loss functions

$$\mathcal{L}_{\text{goal-pos}} := \sum_{i=1}^{R} \|\mathrm{p}_i[K] - G_{\mathrm{p},i}\|_2^2 \tag{25}$$

$$\mathcal{L}_{\text{goal-vel}} := \sum_{i=1}^{R} \|\mathrm{v}_i[K]\|_2^2; \tag{26}$$

we do not penalize the final acceleration.

We deem robot i's final state constraints to be satisfied if

$$\|\mathrm{p}_i[K] - G_{\mathrm{p},i}\|_2 \leq D_{\text{goal-pos}}, \text{ and}$$
$$\|\mathrm{v}_i[K]\|_2 \leq D_{\text{goal-vel}} \tag{27}$$

where $D_{\text{goal-pos}}$ and $D_{\text{goal-vel}}$ define an acceptable deviation of the final position and velocity around the desired goal states. In this case study we set $D_{\text{goal-pos}} = 0.05\,\mathrm{m}$ and $D_{\text{goal-vel}} = 0.05\,\mathrm{m\,s^{-1}}$, a deviation that is well within the ability of a hover controller to stabilize.

5) OPTIMIZATION & RESULTS

We used the Adam gradient descent method [38] for minimization of the loss function (19). Using an automatic hyperparameter search [39] yielded the hyperparameter values shown in Table 1.

Fig. 3 shows a histogram of the time required to generate feasible, collision-free trajectories for 200 quadcopters exiting the maze. These results are based on 1000 random initializations of this case study. These results show that 50% of initializations were solved in under 2.28 seconds, and 90% of initializations in under 3.38 seconds.

TABLE 1. Hyperparameters for the "Sort 200" case study, as optimized by a hyperparameter search aiming to minimize the algorithm's 90th-percentile convergence time.

learning-rate	1.46×10^{-3}
$w_{\text{quad-collision}}$	1.16×10^{4}
$w_{\text{maze-collision}}$	2.71×10^{2}
$w_{\text{maze-wall-entry}}$	2.71×10^{5}
$w_{\text{goal-pos}}$	1.86×10^{4}
$w_{\text{goal-vel}}$	2.11×10^{4}
w_{thrust}	1×10^{5}
$w_{\text{body-rate}}$	1×10^{5}

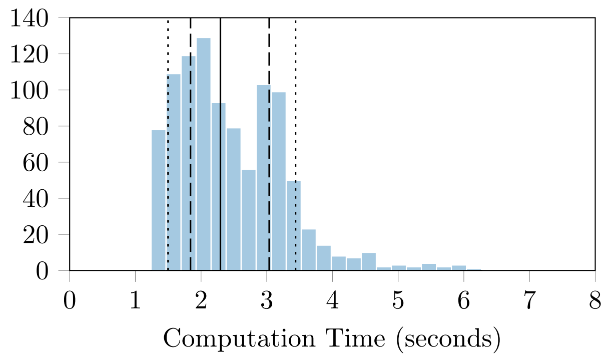


FIGURE 3. A histogram showing the time required for the generation of feasible, collision-free trajectories for a swarm of 200 quadcopters. This histogram summarizes the results of 1000 trials. The median calculation time of 2.28 seconds is shown as a solid line. The 10th and 90th percentiles (1.52 and 3.38 seconds respectively) are shown as dotted lines, and the 25th and 75th percentiles (1.83 and 3.03 seconds respectively) are shown as dashed lines.

B. GROUND ROBOT TRANSITIONS

In this scenario, a fleet of 100 ground robots with bicycle dynamics is tasked with changing its formation. We assume that robots begin and end in rest and require that robots arrive at their goal positions at the same time. As an example of our method's applicability to heterogenous swarms, we select 50 robots to have a maximum steering angle of 20°, and the other 50 robots to have a maximum steering angle of 70°. An example initialization of this problem is shown in Fig. 2, where each robot in the fleet changes its position within a "smiley face" formation.

1) ROBOT MODELING

We model each ground robot using the kinematic model of a bicycle [46], which is a common choice for modeling front-steer vehicles. We assume that dynamics not modeled by the kinematic model (e.g. inertia) are compensated for by a controller able to track the state and input trajectories generated by our method.

With reference to Fig. 4, we model each robot as a bicycle with a fixed rear wheel located $l_r = 0.5\,\mathrm{m}$ from its center and a steerable front wheel located $l_f = 0.5\,\mathrm{m}$ from its center. We denote the angle of its front wheel with respect to its longitudinal axis by δ. We denote the location of each robot's center in the two-dimensional inertial frame by p_x and p_y, the angle of its longitudinal axis relative to the inertial frame by φ, its forward velocity and acceleration by v_b and a_b, and the angle of its forward velocity relative to its longitudinal axis by β. The continuous-time kinematics of a bicycle robot

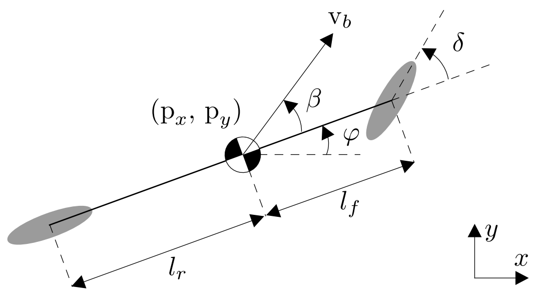


FIGURE 4. An illustration of a robot with bicycle dynamics. Such robots have a steerable front wheel and a fixed rear wheel located at distances l_f and l_r from the robot's center. We denote by δ the steering angle of the front wheel, by φ the angle of the robot with respect to the inertial frame, and by β the angle of the robot's velocity v_b with respect to its longitudinal axis. We use p_x and p_y to denote the location of the robot in the inertial frame. Adapted from [46].

are then

$$\dot{\mathrm{p}_x} = \mathrm{v}_b \cos(\varphi + \beta) \tag{28}$$
$$\dot{\mathrm{p}_y} = \mathrm{v}_b \sin(\varphi + \beta) \tag{29}$$
$$\dot{\varphi} = \frac{\mathrm{v}_b}{l_r} \sin(\beta) \tag{30}$$
$$\dot{\mathrm{v}_b} = \mathrm{a}_b, \tag{31}$$

where

$$\beta := \tan^{-1}\left(\tfrac{l_r}{l_f+l_r}\tan(\delta)\right). \tag{32}$$

We discretize the above using the Euler forward method with a sampling period of $T = 50\,\mathrm{ms}$ to arrive at the discrete-time model

$$\mathrm{p}_x[k+1] = \mathrm{p}_x[k] + T\,\mathrm{v}_b[k]\cos(\varphi[k] + \beta[k]) \tag{33}$$
$$\mathrm{p}_y[k+1] = \mathrm{p}_y[k] + T\,\mathrm{v}_b[k]\sin(\varphi[k] + \beta[k]) \tag{34}$$
$$\varphi[k+1] = \varphi[k] + T\,\frac{\mathrm{v}_b[k]}{l_r}\sin(\beta[k]) \tag{35}$$
$$\mathrm{v}_b[k+1] = \mathrm{v}_b[k] + T\,\mathrm{a}_b[k] \tag{36}$$

where

$$\beta[k] = \tan^{-1}\left(\frac{l_r}{l_f + l_r}\tan(\delta[k])\right). \tag{37}$$

Based on the above model, we define the $N = 4$ dimensional state space of a robot as

$$S := (\mathrm{p}_x,\ \mathrm{p}_y,\ \mathrm{v}_b,\ \varphi) \in \mathbb{R}^{K+1\times N}. \tag{38}$$

As in the previous case study, these recursive difference equations can be implemented using cumulative summation:

$$\begin{aligned}
\beta[0\!:\!K-1] &= \tan^{-1}\left(\frac{l_r}{l_f+l_r}\tan(\delta[0\!:\!K-1])\right)\\
\mathrm{v}_b[1\!:\!K] &= \mathrm{v}_b[0] + T\cdot\mathrm{CS}(\mathrm{a}_b[0\!:\!K-1])\\
\varphi[1\!:\!K] &= \varphi[0] + T\cdot\mathrm{CS}\left(\frac{\mathrm{v}_b[0\!:\!K-1]}{l_r}\sin(\beta[0\!:\!K-1])\right)\\
\mathrm{v}_x[0\!:\!K-1] &= \mathrm{v}_b[0\!:\!K-1]\cos(\varphi[0\!:\!K-1] + \beta[0\!:\!K-1])\\
\mathrm{v}_y[0\!:\!K-1] &= \mathrm{v}_b[0\!:\!K-1]\sin(\varphi[0\!:\!K-1] + \beta[0\!:\!K-1])\\
\mathrm{p}_x[1\!:\!K] &= \mathrm{p}_x[0] + T\cdot\mathrm{CS}(\mathrm{v}_x[0\!:\!K-1])\\
\mathrm{p}_y[1\!:\!K] &= \mathrm{p}_y[0] + T\cdot\mathrm{CS}(\mathrm{v}_y[0\!:\!K-1]),
\end{aligned} \tag{39}$$

where the ordering of equations indicates the necessary order of calculation.

2) INITIAL AND FINAL CONDITIONS

We assume that robot i begins at the known position

$$\mathrm{p}_i[0] := (\mathrm{p}_{x,i}[0],\ \mathrm{p}_{y,i}[0]), \tag{40}$$

and at standstill, such that

$$\mathrm{v}_{b,i}[0] = 0. \tag{41}$$

We randomly assign robot i a goal location

$$G_{\mathrm{p},i} := (G_{x,i},\ G_{y,i}), \tag{42}$$

and assume that it begins facing this goal

$$\varphi_i[0] = \mathrm{arctan2}\left(\mathrm{p}_{y,i} - G_{y,i},\ \mathrm{p}_{x,i} - G_{x,i}\right). \tag{43}$$

We require robot i to finish at its goal and at rest, that is

$$\begin{aligned}
\mathrm{p}_{x,i}[K] &\approx G_{x,i}\\
\mathrm{p}_{y,i}[K] &\approx G_{y,i}\\
\mathrm{v}_{b,i}[K] &\approx 0,
\end{aligned} \tag{44}$$

where tolerances around the goal state are defined below. We leave the robot's final orientation $\varphi_i[K]$ unconstrained.

3) INITIALIZATION OF INDIVIDUAL TRAJECTORIES

Since each robot begins facing its goal, the trajectory generation problem is reduced to a one-dimensional problem of generating a straight-line trajectory, which is feasible under the dynamic constraints (discussed below). We parameterize the position trajectory using a fifth-order polynomial optimized to minimize trajectory jerk [5]. Trajectory duration is calculated such that the robot with the furthest distance to travel will reach its goal as quickly as is allowed by its acceleration limits and by the trajectory parameterization. We sample the polynomial corresponding to each robot's acceleration trajectory with the desired sampling period (in our case, 50 ms), and use this to initialize its acceleration trajectory. We initialize each robot's steering trajectory with zero. This stage can be computed entirely on the GPU.

4) CONSTRAINTS

The constraint set consists of constraints on the robots' acceleration and steering angle, constraints prohibiting robot collisions, and constraints on the final state of each robot.

a: INPUT CONSTRAINTS

In this case-study, we constrain robot i's acceleration to

$$\mathrm{a}_{b,i} \in (-\mathrm{a}_{\max},\ \mathrm{a}_{\max}), \tag{45}$$

where $\mathrm{a}_{\max} := 2\,\mathrm{m\,s}^{-2}$, and its steering angle to

$$\delta_i \in (-\delta_{i,\max},\ \delta_{i,\max}), \tag{46}$$

where $\delta_{i,\max} := 20°$ if $i \leq 50$ or otherwise $70°$.

As demonstrated in the previous case-study, constraints can be implemented using loss functions and added to the objective function. Although input constraints can also be modeled in this way, it is often easier to enforce simple input constraints directly by expressing a robot's constrained input as a function of an unconstrained optimization variable. In this case study, we use the $\tanh(\cdot)$ function to map an input on the domain of $(-\infty,\ \infty)$ to an output on the range $(-1,\ 1)$. The input constraints can then be directly expressed as

$$\mathrm{a}_{b,i} = \mathrm{a}_{\max} \tanh(\bar{\mathrm{a}}_{b,i}) \tag{47}$$

$$\delta_i = \delta_{i,\max} \tanh(\bar{\delta}_i), \tag{48}$$

where $\bar{\mathrm{a}}_{b,i}$ and $\bar{\delta}_i$ are the unconstrained targets of the optimization routine. We amend the robot state update equations (39) with the above transformations, and define the $M = 2$ dimensional input space of each robot as

$$U_i := (\bar{\mathrm{a}}_{b,i},\ \bar{\delta}_i), \tag{49}$$

from which the actual, constrained robot inputs $\mathrm{a}_{b,i}$ and δ_i can later be recovered.

b: ROBOT COLLISION CONSTRAINTS

In this case study we model robots as circles and require a minimum center-to-center distance of $D := 1.0\,\mathrm{m}$. We again use Algorithm 2 to compute the corresponding collision loss (24) and associated gradients.

c: GOAL CONSTRAINTS

We penalize the deviation from the desired goal position and velocity as in (25), and again set $D_{\text{goal-pos}} = 0.05\,\mathrm{m}$ and $D_{\text{goal-vel}} = 0.05\,\mathrm{m\,s^{-1}}$.

5) OPTIMIZATION & RESULTS

We use the Adam gradient descent method [38] for minimization of the global loss. Using an automatic hyperparameter search [39] yielded the hyperparameter values shown in Table 2.

TABLE 2. Hyperparameter values for the ground robot transition case study, as determined by a hyperparameter search [39] aiming to minimize the algorithm's 90th percentile convergence time.

learning-rate	6.94×10^{-3}
$w_{\text{collision}}$	2.14×10^{3}
$w_{\text{final-pos}}$	2.07×10^{4}
$w_{\text{final-vel}}$	5.95×10^{3}

Fig. 5 shows a histogram of the time required to generate feasible, collision-free trajectories for 100 ground robots based on 1000 random initializations of this case study. These results show that 50% of initializations were completed in under 1.6 seconds, and 90% of initializations in under 3.34 seconds.

As described in Section III-E, the hyperparameter values (loss weights, as well as the learning rate of the optimizer) play an important role in determining the speed of convergence. We investigate the effects of hyperparameter tuning in Fig. 6 by plotting the the number of collisions (orange) and the root-mean-squared distance violation of the final position constraint (black), against the optimization iteration. Each plot begins from an identical initial state, and we are therefore able to reason about how variations in hyperparameter values affect convergence.

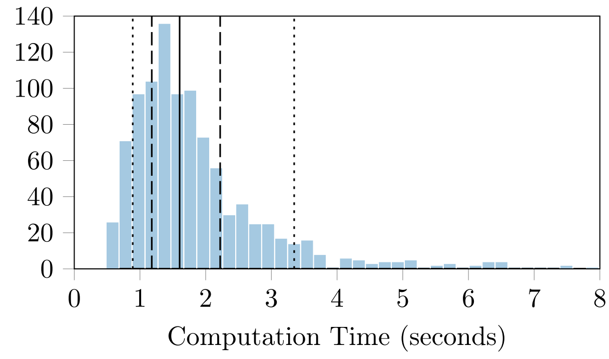


FIGURE 5. A histogram showing the time required for the generation of feasible, collision-free trajectories for a fleet of 100 ground robots. This histogram summarizes the results of 1000 trials. The median calculation time of 1.6 seconds is shown as a solid line. The 10th and 90th percentiles (0.88 and 3.34 seconds respectively) are shown as dotted lines, and the 25th and 75th percentiles (1.18 and 2.22 seconds respectively) are shown as dashed lines.

The center plots of Fig. 6 show the convergence toward feasibility when using the optimal hyperparameters (Table 2), which strike a balance between quickly decreasing the number of collisions, while keeping the final positions close to their goals. In Fig. 6(ai) we observe that by reducing $w_{\text{collision}}$, the number of collisions is not reduced as quickly, however the final positions remain closer to their goals. By increasing $w_{\text{collision}}$ in Fig. 6(aiii) we observe the opposite: the number of collisions is quickly reduced (by drastically perturbing robot trajectories), thus causing an increase in the violation of the final position constraint. In Fig. 6(bi) the effect of a low learning rate, which causes a slow convergence to feasibility, is contrasted against the optimal learning rate in Fig. 6(bii), and against a high learning rate in Fig. 6(biii), which fails to converge.

V. FUTURE WORK

In this paper we introduce an approach for quickly generating feasible, collision-free trajectories for robot swarms. Our method leverages the computational power of modern GPUs to directly solve the non-convex optimization problem using gradient descent. As experimentally shown in the case-studies of Section IV, our method is capable of generating feasible, collision-free trajectories for swarms of hundreds of robots in just seconds, and can easily be extended to heterogenous swarms. Given the parallel nature of the problem, we reason that our method's performance will increase with progressive advances in GPU processing power.

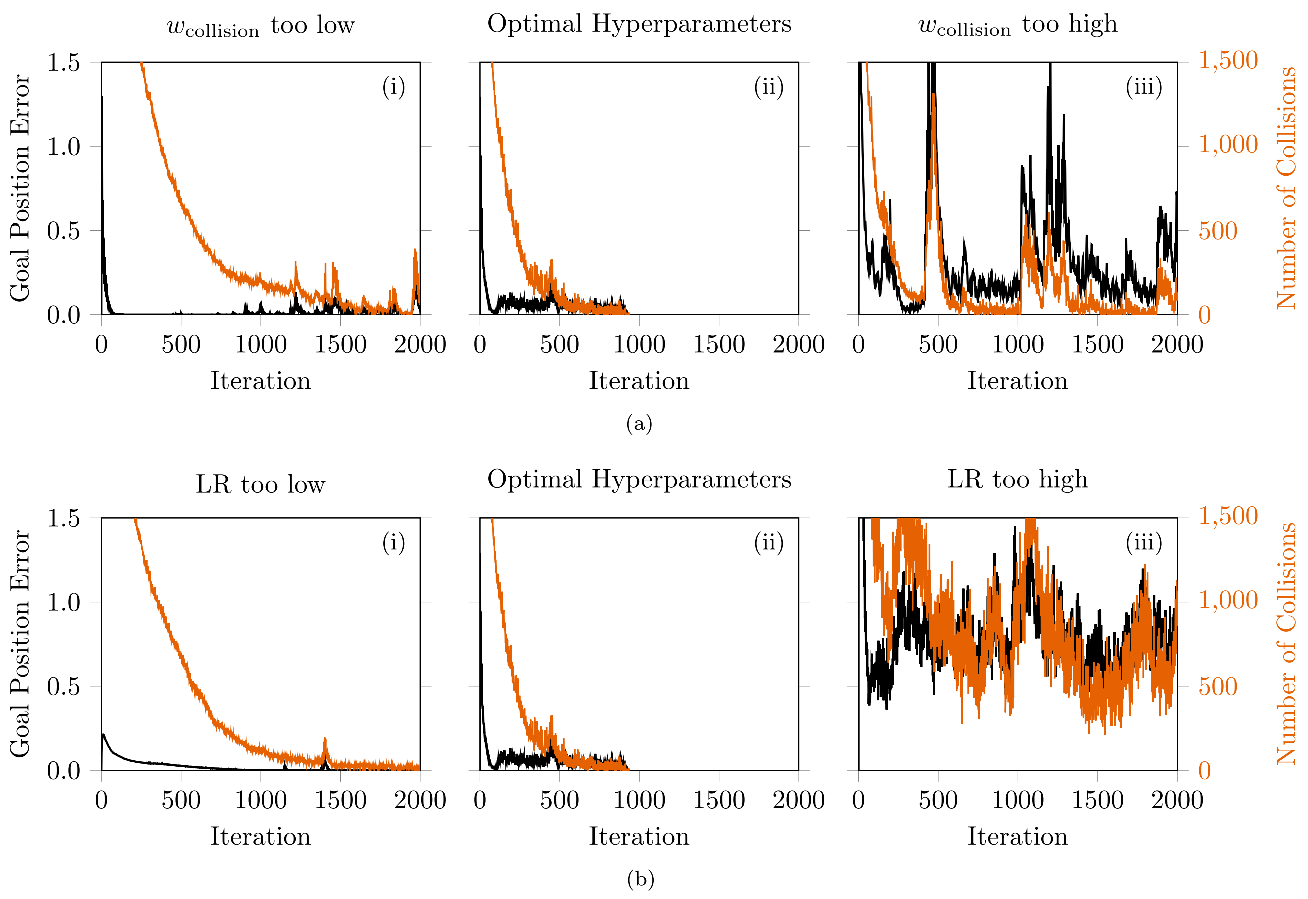


FIGURE 6. In this figure we investigate the effects of hyperparameter tuning by plotting the root-mean-squared distance violation of the final position constraint (black) and the number of collisions (orange) against the optimization iteration. Each plot begins from an identical initial state, thus allowing us to reason about how variations in hyperparameter values affect convergence. These plots exemplify the importance of hyperparameter selection, demonstrating how the optimal selection of hyperparameters strikes a careful balance between quickly decreasing the number of collisions and ensuring that other constraints are not too drastically violated. (a) The effect of varying $w_{collision}$, the weight given to reducing quadcopter collisions. (b) The effect of varying the "learning rate" of the gradient descent optimizer.

A. REAL-TIME, MODEL PREDICTIVE CONTROL

Although not touched upon in this paper, the speed with which our method can generate trajectories for large swarms of robots suggests a possible application to real-time reference generation. Warm starting the algorithm should only improve its speed and allow for trajectories to be quickly replanned to account for situational changes and compensate for unmodeled effects.

B. FACTORIZATION INTO LOCAL POLICIES

The centralized method we present is efficient at computing trajectories, but communication bottlenecks may limit its application to large swarms if trajectories need to be communicated or updated in real time. Despite the complexity of the swarm trajectory generation problem, once a feasible solution is found, the collision avoidance behaviors demonstrated by individual robots are largely predictable. This suggests that such behavior could be encoded in a local policy and run on each robot independently.

REFERENCES

[1] J. A. Preiss, W. Hönig, N. Ayanian, and G. S. Sukhatme, "Downwash-aware trajectory planning for large quadrotor teams," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2017, pp. 250–257.

[2] M. Hehn and R. D'Andrea, "Quadrocopter trajectory generation and control," *IFAC Proc. Volumes*, vol. 44, no. 1, pp. 1485–1491, 2011.

[3] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2011, pp. 2520–2525.

[4] D. Mellinger, N. Michael, and V. Kumar, "Trajectory generation and control for precise aggressive maneuvers with quadrotors," *Int. J. Robot. Res.*, vol. 31, no. 5, pp. 664–674, 2012.

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

[6] C. Richter, A. Bry, and N. Roy, "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments," in *Robotics Research*. Cham, Switzerland: Springer, 2016, pp. 649–666.

[7] S. Liu, N. Atanasov, K. Mohta, and V. Kumar, "Search-based motion planning for quadrotors using linear quadratic minimum time control," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2017, pp. 2872–2879.

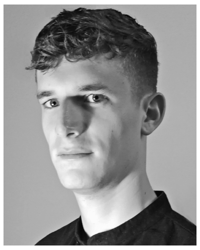
[8] S. Liu, K. Mohta, N. Atanasov, and V. Kumar, "Search-based motion planning for aggressive flight in SE(3)," *IEEE Robot. Autom. Lett.*, vol. 3, no. 3, pp. 2439–2446, Jul. 2018.

[9] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Dept. Comput. Sci., Iowa State Univ., Ames, IA, USA, Tech. Rep. 98-11, 1998.

[10] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," in *Proc. Robot. Sci. Syst. Conf. VI*, Zaragoza, Spain, vol. 104, 2010, p. 2.
[11] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Autom.*, vol. 12, no. 4, pp. 566–580, Aug. 1996.
[12] M. E. Flores, "Real-time trajectory generation for constrained nonlinear dynamical systems using non-uniform rational B-spline basis functions," Ph.D. dissertation, California Inst. Technol., Pasadena, CA, USA, 2008.
[13] S. Tang and V. Kumar, "Safe and complete trajectory generation for robot teams with higher-order dynamics," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2016, pp. 1894–1901.
[14] M. G. Earl and R. D'Andrea, "Iterative MILP methods for vehicle-control problems," *IEEE Trans. Robot.*, vol. 21, no. 6, pp. 1158–1167, Dec. 2005.
[15] F. Augugliaro, A. P. Schoellig, and R. D'Andrea, "Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2012, pp. 1917–1922.
[16] Y. Chen, M. Cutler, and J. P. How, "Decoupled multiagent path planning via incremental sequential convex programming," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2015, pp. 5954–5961.
[17] J. Alonso-Mora, S. Baker, and D. Rus, "Multi-robot navigation in formation via sequential convex programming," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2015, pp. 4634–4641.
[18] D. Morgan, G. P. Subramanian, S.-J. Chung, and F. Y. Hadaegh, "Swarm assignment and trajectory optimization using variable-swarm, distributed auction assignment and sequential convex programming," *Int. J. Robot. Res.*, vol. 35, no. 10, pp. 1261–1285, 2016.
[19] D. Mellinger, A. Kushleyev, and V. Kumar, "Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2012, pp. 477–483.
[20] J. Bento, N. Derbinsky, J. Alonso-Mora, and J. S. Yedidia, "A message-passing algorithm for multi-agent trajectory planning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 521–529.
[21] C. E. Luis and A. P. Schoellig. (2018). "Trajectory generation for multiagent point-to-point transitions via distributed model predictive control." [Online]. Available: https://arxiv.org/abs/1809.04230
[22] S. Agarwal and S. Akella, "Simultaneous optimization of assignments and goal formations for multiple robots," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2018, pp. 6708–6715.
[23] M. Debord, W. Hönig, and N. Ayanian, "Trajectory planning for heterogeneous robot teams," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, 2018.
[24] W. Hönig, J. A. Preiss, T. S. Kumar, G. S. Sukhatme, and N. Ayanian, "Trajectory planning for quadrotor swarms," *IEEE Trans. Robot.*, vol. 34, no. 4, pp. 856–869, Aug. 2018.
[25] D. Lucas and C. Crane, "Development of a multi-resolution parallel genetic algorithm for autonomous robotic path planning," in *Proc. 12th Int. Conf. Control, Autom. Syst. (ICCAS)*, Oct. 2012, pp. 1002–1006.
[26] U. Cekmez, M. Ozsiginan, and O. K. Sahingoz, "A UAV path planning with parallel ACO algorithm on CUDA platform," in *Proc. Int. Conf. Unmanned Aircr. Syst. (ICUAS)*, May 2014, pp. 347–354.
[27] P. Cai, Y. Cai, I. Chandrasekaran, and J. Zheng, "A GPU-enabled parallel genetic algorithm for path planning of robotic operators," in *GPU Computing and Applications*. Singapore: Springer, 2015, pp. 1–13.
[28] P. Cai, Y. Cai, I. Chandrasekaran, and J. Zheng, "Parallel genetic algorithm based automatic path planning for crane lifting in complex environments," *Automat. Construct.*, vol. 62, pp. 133–147, 2016.
[29] V. Roberge, M. Tarbouchi, and G. Labonté, "Fast genetic algorithm path planner for fixed-wing military UAV using GPU," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 54, no. 4, pp. 2105–2117, Oct. 2018.
[30] J. Pan, C. Lauterbach, and D. Manocha, "g-planner: Real-time motion planning and global navigation using GPUs," in *Proc. AAAI*, 2010, pp. 1–7.
[31] J. Pan, C. Lauterbach, and D. Manocha, "Efficient nearest-neighbor computation for GPU-based motion planning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2010, pp. 2243–2248.
[32] J. T. Kider, M. Henderson, M. Likhachev, and A. Safonova, "High-dimensional planning on the GPU," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2010, pp. 2515–2522.
[33] M. Likhachev and A. Stentz, "R* search," in *Proc. Nat. Conf. Artif. Intell. (AAAI)*, 2008, pp. 1–9.
[34] B. Chretien, A. Escande, and A. Kheddar, "GPU robot motion planning using semi-infinite nonlinear programming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 2926–2939, 2016.
[35] J. Pan and D. Manocha, "GPU-based parallel collision detection for fast motion planning," *Int. J. Robot. Res.*, vol. 31, no. 2, pp. 187–200, 2012.
[36] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI*, vol. 16, 2016, pp. 265–283.
[37] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *Proc. 31st Conf. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 1–4.
[38] D. P. Kingma and J. Ba. (2014). "Adam: A method for stochastic optimization." [Online]. Available: https://arxiv.org/abs/1412.6980
[39] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 2951–2959.
[40] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.
[41] R. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles," *IEEE Robot. Autom. Mag.*, vol. 19, no. 3, pp. 20–32, Sep. 2012.
[42] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
[43] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Res. Logistics Quart.*, vol. 2, nos. 1–2, pp. 83–97, Mar. 1955.
[44] Bitcraze AB. (2018). *Crazyflie 2.0*. [Online]. Available: https://www.bitcraze.io/crazyflie-2/
[45] J. Förster, "System identification of the crazyflie 2.0 nano quadrocopter," B.S. thesis, Dept. Mech. Eng., Inst. Dyn. Syst. Control, ETH Zürich, Zürich, Switzerland, 2015.
[46] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli, "Kinematic and dynamic vehicle models for autonomous driving control design," in *Proc. Intell. Vehicles Symp.*, 2015, pp. 1094–1099.

MICHAEL HAMER received the B.Eng. degree in computer engineering and the B.Sc. degree in computer science from Curtin University, WA, Australia, in 2009, and the M.S. degree in robotics, systems, and control from ETH Zurich, Switzerland, in 2013, where he is currently pursuing the Ph.D. degree with the Institute for Dynamic Systems and Control. His research focuses on robot localization, and the application of machine learning to physical systems.

He is with the Institute for Dynamic Systems and Control, ETH Zurich, where he is involved in various public quadcopter exhibitions, including quadcopter task scheduling for the Flight Assembled Architecture installation, in 2011, and an artistic quadcopter exhibit for TEDGlobal, in 2013. He is also an active Contributor of the Crazyflie Open-Source Project, with major contributions made to the control, estimation, and localization systems.



LINO WIDMER received the B.Sc. degree in mechanical engineering from ETH Zurich, Switzerland, in 2015, where he is currently pursuing the M.S. degree in mechanical engineering, with a focus on optimization and machine learning. He has focused on mitigating numerical issues that arise when solving second-order cone programs, optimization-based trajectory generation, and using machine learning to improve the accuracy of radio-based localization.

RAFFAELLO D'ANDREA received the B.Sc. degree in engineering science from the University of Toronto, in 1991, and the M.S. and Ph.D. degrees in electrical engineering from the California Institute of Technology, in 1992 and 1997, respectively.

He was an Assistant and then an Associate Professor with Cornell University, from 1997 to 2007. While on leave from Cornell University, from 2003 to 2007, he co-founded Kiva Systems (now Amazon Robotics), where he led the systems architecture, robot design, robot navigation and coordination, and control algorithms development. He is currently a Professor of dynamic systems and control with ETH Zurich and also the Chairman of the Board with Verity Studios AG.