GPU Accelerated Convex Approximations for Fast Multi-Agent Trajectory Optimization

Fatemeh Rastgar, Houman Masnavi, Jatan Shrestha, Karl Kruusamäe, Alvo Aabloo, Arun Kumar Singh

***Abstract*— In this paper, we present a computationally efficient trajectory optimizer that can exploit GPUs to jointly compute trajectories of tens of agents in under a second. At the heart of our optimizer is a novel reformulation of the non-convex collision avoidance constraints that reduces the core computation in each iteration to that of solving a large scale, convex, unconstrained Quadratic Program (QP). We also show that the matrix factorization/inverse computation associated with the QP needs to be done only once and can be done offline for a given number of agents. This further simplifies the solution process, effectively reducing it to a problem of evaluating a few matrix-vector products. Moreover, for a large number of agents, this computation can be trivially accelerated on GPUs using existing off-the-shelf libraries. We validate our optimizer's performance on challenging benchmarks and show substantial improvement over state of the art in computation time and trajectory quality.**

I. Introduction

Coordinating multiple agents between given start and goal positions without collision is crucial to any multi-agent application. For highly agile agents like quadrotors and autonomous cars, a popular approach has been to formulate this collision-free coordination as a trajectory optimization problem. There are two core computational challenges in this context. First, the number of variables in the optimization problem increases linearly with the number of agents n. Second and more importantly, the number of pair-wise non-convex collision avoidance constraints grow by a factor $\binom{n}{2}$. Existing works have predominantly explored two classes of simplifications to keep the optimization problem tractable. The sequential approaches, e.g., [1], [2], follow an iterative process wherein motion plans for only one agent is computed at a time. Collision avoidance is ensured by treating agents whose motions were computed in earlier iterations as dynamic non-responsive obstacles for the currently planned agent. On the other hand, the distributed model predictive control (MPC) approaches such as [3], [4] decouple the planning process by allowing each agent to view all the others at any given instant as dynamic obstacles with a known trajectory. The sequential approaches do not leverage the cooperation between the agents while it is incorporated only implicitly (through trajectory prediction) in the distributed MPC based approaches. As a result, both classes of simplifications have access to a smaller feasible space and are thus conservative.

All authors are with the Institute of Technology, University of Tartu. The work was supported in part by the European Social Fund through IT Academy program in Estonia, smart specialization project with BOLT and Estonian Centre of Excellence in IT (EXCITE) funded by the European Regional Development Fund.

In this paper, we explicitly account for inter-agent cooperation by adopting the classical set-up of [5], wherein a large scale optimization is formulated to compute the trajectories of all the agents jointly. The primary goal of this paper is to improve the computational tractability of such large scale optimization problems. Our optimizer falls into the class of existing algorithms such as [6] that reformulates the underlying numerical computation of the optimizer to parallelize them over CPUs/GPUs.

A. Main Idea

Consider the following unconstrained, quadratic program (QP) for a constant matrix $\mathbf{Q}$ and a vector $\mathbf{q}$. As shown, the solution process reduces to solving a set of linear equations.

$$\min_{\boldsymbol{\xi}} \frac{1}{2}\boldsymbol{\xi}^T\mathbf{Q}\boldsymbol{\xi} + \mathbf{q}^T\boldsymbol{\xi}, \Rightarrow \mathbf{Q}\boldsymbol{\xi} = -\mathbf{q} \tag{1}$$

Now, imagine that QP (1) needs to be solved for several instantiations of $\mathbf{q}$ for a given $\mathbf{Q}$. Such scenarios are common in linear MPC, wherein $\mathbf{Q}$ encodes the system dynamics and cost functions and is thus constant, while $\mathbf{q}$ that encodes the initial condition changes at each iteration. As shown in [7], an efficient way of handling such scenarios is to pre-compute the inverse (or just the factorization) of $\mathbf{Q}$, in which case, MPC computation (or solving (1)) in each iteration reduces to evaluating just matrix-vector products. This reduction becomes particularly important in cases where the QP (1) is formulated over tens of hundreds of variables.

B. Contributions

For the first time, we show how the idea of off-line caching of matrix inverses can be used to accelerate non-linear and non-convex, multi-agent trajectory optimization problems. The fundamental algorithmic challenge stems from the fact the inter-agent collision avoidance constraints have a non-convex quadratic form. In contrast, the techniques presented in existing works, such as [8] for leveraging off-line cached matrix inverses, assume a convex problem with affine constraints. A straightforward extension of works like [8] to a multi-agent trajectory optimization problem leads to critical computational bottlenecks with regards to matrix inverse caching and GPU accelerations. By-passing these limitations require several reformulation layers and forms the paper's main theoretical contribution, summarized below (see Section II-D).

Firstly, we avoid modeling collision avoidance as a non-convex quadratic and instead model it as a non-linear equality constraint by re-writing the inter-agent separation vector in polar form (see eqn. (12)). Secondly, instead of adopting the standard approach of sequential linearization of non-linear collision avoidance constraints (see Section II-D), we advocate an Alternating Minimization (AM) procedure [9] that groups the variables into specific blocks and optimizes them over them in a sequence. The most computationally intensive block has a structure similar to (1), while the rest of the optimization blocks can decompose into several parallel single-variable optimization problems.

Our proposed optimizer provides the following benefits over the current state of the art.

Ease of Implementation and GPU Accelerations: The entire numerical computation of our optimizer reduces to computing either element-wise operations over vectors or matrix-vector products. For a large number of agents, these can be trivially accelerated on GPUs using libraries like CUPY [10] and JAX [11]. We also provide an open-source implementation in `https://github.com/arunkumar-singh/GPU-Multi-Agent-Traj-Opt`. Our GPU accelerated optimizer can compute trajectories for 32 agents in 0.7 s on a RTX-2080 enabled desktop computer.

State of the Art Performance: Our optimizer outperforms the computation time of joint trajectory optimization of [5] by more than two orders of magnitude while achieving trajectories of similar quality. It also outperforms the current state of the art, sequential approach of [2], by obtaining shorter trajectories in all the considered benchmarks. Even more importantly, our optimizer also outperforms [2] in terms of the computation time on several benchmarks. The speed-up is particularly impressive given that our optimizer performs a much more rigorous joint search over the agents' trajectory space.

Suitability on Edge-Devices: Our optimizer can compute trajectories of 16 agents in around $2s$ on Nvidia Jetson-TX2. To put it in context, this is around two orders of magnitude faster than the computation time of [5] on an Intel i7 desktop computer with 32GB RAM. Thus, our work presents an important step towards achieving complex onboard decision-making abilities for quadrotors that can carry only small, light-weight computational resources.

II. Background and Preliminaries

This section introduces some necessary mathematical preliminaries and uses them to draw a contrast between existing works and our optimizer. We begin by summarizing next the basic symbols and notations used throughout the paper.

A. Symbols and Notations

We will use lower case normal font letters to represent scalars, while bold font variants represent vectors. Matrices are represented through upper case bold fonts. The time dependency of the variable is shown by t. The superscript T will denote the transpose of vectors and matrices. The left superscript k will be used to indicate the iteration index in a trajectory optimizer. We use subscript i, j as an agent index. We will use n, m to represent the number of agents and planning horizon throughout the paper.

B. Joint Multi-Agent Trajectory Optimization of [5]

For spheroid agents with dimension $\frac{l_{xy}}{2}, \frac{l_z}{2}$, the joint multi-agent trajectory optimization can be formulated in the following manner [5].

$$\min_{x_i(t), y_i(t), z_i(t)} \sum_i \sum_t \ddot{x}_i^2(t) + \ddot{y}_i^2(t) + \ddot{z}_i^2(t) \quad (2a)$$

$$(x_i(t), y_i(t), z_i(t)) \in \mathcal{C}_{boundary} \quad (2b)$$

$$f_c(x_i(t), y_i(t), z_i(t), x_j(t), y_j(t), z_j(t)) \leq 0, \forall t, i, j, i \neq j \quad (2c)$$

$$f_c(x_i(t), y_i(t), z_i(t), x_j(t), y_j(t), z_j(t)) = -\frac{(x_i(t) - x_j(t))^2}{l_{xy}^2} - \frac{(y_i(t) - y_j(t))^2}{l_{xy}^2} - \frac{(z_i(t) - z_j(t))^2}{l_z^2} + 1 \leq 0, \quad (3)$$

where, $(x_i(t), y_i(t), z_i(t))$ represents the position of the i^{th} at some time t. The cost function minimizes the acceleration magnitude along each motion axis at each time instant. The set $\mathcal{C}_{boundary}$ stems from the boundary conditions on the agent trajectories. The inequality constraints model the collision avoidance constraints.

There are two key bottlenecks in the above optimization problem. Firstly, as mentioned earlier, the number of variables and constraints show a linear and exponential increase respectively. Secondly, the non-convex quadratic collision-avoidance constraints makes the optimization problem computationally intractable. Interestingly, a simple linearization of (3) around any arbitrary guess trajectory leads to a convex but conservative approximation of the feasible space [12], [13]. This simplification has been exploited in many recent works on multi-agent trajectory optimization such as [5], [1] leading to a so-called sequential convex programming (SCP) optimizer.

C. GPU Acceleration Through Gradient Descent

An effective way of accelerating optimization problems on GPU is to reformulate them in an unconstrained form and then apply the method of Gradient Descent. For example, [14] achieves this for the multi-agent trajectory optimization by augmenting the constraints (2b)-(2c) as penalties in the cost function (2a).

Our optimizer provides substantial improvements over [14]. As mentioned by the authors themselves, [14] requires extensive hyper-parameter tuning that is likely to be redone if the problem parameters such as robot dimension change. In contrast, the proposed optimizer relies on accelerating a QP on GPU by offline caching of matrix inverses and worked with trivial default parameters on dozens of examples.

D. Connections with ADMM

Our optimizer is closely related to a class of optimization technique called Alternating Direction Method of Multipliers (ADMM) (see Algorithm 1) [8]. This section shows that a straightforward extension of existing ADMM to non-convex problems such as multi-agent trajectory optimization do not lead to the possibility of offline caching of matrix inverses. To understand this further, consider the following optimization problem over variables ($\boldsymbol{\xi}$, s), a constant positive definite matrix $\mathbf{Q}$ and a non-linear and non-convex function $\mathbf{f}$.

$$\frac{1}{2}\boldsymbol{\xi}^T\mathbf{Q}\boldsymbol{\xi} \tag{4}$$

$$f(\boldsymbol{\xi}) + s = 0, \qquad s \geq 0 \tag{5}$$

To solve the above problem using an ADMM based approach, we first formulate the following augmented Lagrangian with multiplier λ.

$$\frac{1}{2}\boldsymbol{\xi}^T\mathbf{Q}\boldsymbol{\xi} + \frac{\rho}{2}\|f(\boldsymbol{\xi}) + s + \frac{\lambda}{\rho}|_2^2 \tag{6}$$

Next, we minimize the Lagrangian through the following AM iterates, wherein k represents the iteration index [15].

$${}^{k+1}\boldsymbol{\xi} = \arg\min_{\boldsymbol{\xi}} \frac{1}{2}\boldsymbol{\xi}^T\mathbf{Q}\boldsymbol{\xi} + \frac{\rho}{2}\|f(\boldsymbol{\xi}) + {}^{k}s + \frac{{}^{k}\lambda}{\rho}|_2^2 \tag{7}$$

$${}^{k+1}s = \max(0, -f({}^{k+1}\boldsymbol{\xi}) - \frac{{}^{k}\lambda}{\rho}) \tag{8}$$

$${}^{k+1}\lambda = {}^{k}\lambda + \rho(f({}^{k+1}\boldsymbol{\xi}) + {}^{k+1}s) \tag{9}$$

The core computation takes place in step (7) and it is clear that this is not a QP and hence the question of caching matrix inverses do not arise. However, it is always possible to approximate $f(\boldsymbol{\xi})$ in an affine form as ${}^{k}\mathbf{F}\boldsymbol{\xi} + {}^{k}\mathbf{g}$, where the constant matrix ${}^{k}\mathbf{F}$ and vector ${}^{k}\mathbf{g}$ are obtained by the Taylor series expansion of $f(\boldsymbol{\xi})$ at the k^{th} iteration around the current solution ${}^{k}\boldsymbol{\xi}$. Using this affine approximation, r.h.s of step (7) can be rephrased as the following QP.

$$\arg\min_{\boldsymbol{\xi}} \frac{1}{2}\boldsymbol{\xi}^T(\mathbf{Q} + ({}^{k}\mathbf{F})^T({}^{k}\mathbf{F}))\boldsymbol{\xi} + ({}^{k}\mathbf{F}^T({}^{k}\mathbf{g} + {}^{k}s + \frac{{}^{k}\lambda}{\rho})^T\boldsymbol{\xi} \tag{10}$$

Remark 1. *The matrix $(\boldsymbol{Q} + ({}^{k}\boldsymbol{F})^T({}^{k}\boldsymbol{F}))$ and its factorization/inverse needs to be recomputed at each iteration based on the new Taylor series expansion around the current solution.*

Remark 2. *The joint trajectory optimzіation of [5] can be easily adapted to the ADMM set-up described above. However, due to the reasons summarized in Remark 1, the resulting QP structure will not be suitable for GPU acceleration .*

Our optimizer by-passes the computational bottleneck summarized in Remark 1 by avoiding any linearization of the underlying non-linear constraints. Instead, we break the problem into smaller QPs in a way that the associated matrices do not change within the iteration.

E. Connection with Author's Pior Work

Our optimizer is a multi-agent extension of singe-agent trajectory optimization proposed in our prior work [16]. In particular, we rewrite the underlying matrix algebra in a way to distribute computations over GPUs, a feature not explored in [16]. Furthermore, the connections established with existing ADMM techniques in the previous section, iteration complexity derived in Section III-B, and extensive bench-marking with state of the art are other key novelties over [16].

III. Main Results

In this section, we present our main theoretical result: a GPU accelerated optimizer based on convex optimization. We begin by reiterating the main assumptions.

- We consider differentially flat spheroid agents with decoupled affine motion models along the (x, y, z) axis. Such models are suitable for quadrotors [1] and even sometimes for autonomous cars [17]
- We do not explicitly consider the bounds on velocities and accelerations and rely on choosing an appropriate traversal time and regularization on accelerations to ensure the same. The traversal time heuristics is based on the distance between the start and goal positions and the average velocities of the agents. Alternately, like [2], we can also scale the traversal time during post-processing to satisfy the bounds.

A. Reformulation and Alternating Minimization

We reformulate (2a)-(2c) into the following form.

$$\min_{x_i, y_i, z_i, \alpha_{ij}, \beta_{ij}, d_{ij}} \sum_i \sum_t \ddot{x}_i^2(t) + \ddot{y}_i^2(t) + \ddot{z}_i^2(t) \tag{11a}$$

$$(x_i(t), y_i(t), z_i(t)) \in \mathcal{C}_{boundary} \tag{11b}$$

$$\mathbf{f}_c = \mathbf{0}, \forall i, j, t \tag{11c}$$

$$\beta_{ij}(t) \in [0, \pi], \alpha_{ij}(t) \in [-\pi, \pi], d_{ij}(t) \geq 1, \forall i, j, t \tag{11d}$$

$$\mathbf{f}_c = \left\{ \begin{array}{c} x_i(t) - x_j(t) - l_{xy} d_{ij}(t) \sin\beta_{ij}(t) \cos\alpha_{ij}(t) \\ y_i(t) - y_j(t) - l_{xy} d_{ij}(t) \sin\beta_{ij}(t) \sin\alpha_{ij}(t) \\ z_i(t) - z_j(t) - l_z d_{ij} \cos\beta_{ij}(t) \end{array} \right\} \tag{12}$$

The primary changes involve introducing additional time-dependent variables $\alpha_{ij}(t), \beta_{ij}(t), d_{ij}(t)$, and using them to rephrase quadratic collision avoidance constraints (3) into a set of non-linear equalities (12). Intuitively, $(\alpha_{ij}(t), \beta_{ij}(t)), d_{ij}(t)$ represent the 3D solid angles and length of the line of sight vector connecting agents (i, j). Consequently, (12) is just a polar representation of the quadratic constraints (3).

On the surface, our formulation (11a)-(11d) looks more complicated than the more conventional multi-agent trajectory optimization (2a)-(2c) as the former involves highly non-linear trigonometric functions. But in fact, (11a)-(11d) has some hidden geometrical and computational structures that we can expose using techniques from Alternating Minimization (AM). To this end, we first create an augmented cost function $\mathcal{L}$ by incorporating $\mathbf{f}_c$ as l_2 penalties

$$\mathcal{L} = \sum_{i,t} \ddot{x}_i^2(t) + \ddot{y}_i^2(t) + \ddot{z}_i^2(t) +$$
$$\sum_{i,j,t} \frac{\rho}{2}(x_i(t) - x_j(t) - l_{xy} d_{ij}(t) \sin\beta_{ij}(t) \cos\alpha_{ij}(t) + \frac{\lambda_{xij}(t)}{\rho})^2$$

$$+\frac{\rho}{2}(y_i(t) - y_j(t) - l_{xy}d_{ij}(t)\sin\beta_{ij}(t)\sin\alpha_{ij}(t) + \frac{\lambda_{yij}(t)}{\rho})^2$$
$$+\frac{\rho}{2}(z_i(t) - z_j(t) - l_z d_{ij}(t)\cos\beta_{ij}(t) + \frac{\lambda_{zij}(t)}{\rho})^2 \tag{13a}$$

In (13a), ρ is a scalar constant and $\lambda_{xij}(t), \lambda_{yij}(t), \lambda_{zij}(t)$ are time-dependent Lagrange multipliers that can be used to drive the residual of $\mathbf{f}_c$ to zero. Algorithm 1 summarizes the minimization of (13a) subject to (11b) and (11d) based on the AM technique. As before, the left superscript k represents the respective variable at iteration k. As shown, we start with an initialization for ${}^k\alpha_{ij}(t), {}^k\beta_{ij}(t), {}^kd_{ij}(t), {}^k\lambda_{xij}(t), {}^k\lambda_{yij}(t), {}^k\lambda_{zij}(t)$ at $k = 0$ and optimize the variables in a sequence. The optimizations (22a)-(22c) and (22f) are convex constrained QPs. In contrast, (22d) is non-convex. But interestingly, simple geometrical intuitions can be used to derive an analytical solution for it. We delve deeper into each of these optimizations next.

B. Analysis of Algorithm 1

1) Steps (22a)-(22c) : The most important feature of these optimizations is that each of them takes the form of a QP wherein the matrices do not change over iteration. To validate this assertion and to show how it is useful, we parametrize $x_i(t)$ and its derivatives in the following form.

$$\begin{bmatrix} x_i(t_1) \\ x_i(t_2) \\ \dots \\ x_i(t_n) \end{bmatrix} = \mathbf{P}\mathbf{c}_{x_i}, \begin{bmatrix} \dot{x}_i(t_1) \\ \dot{x}_i(t_2) \\ \dots \\ \dot{x}_i(t_n) \end{bmatrix} = \dot{\mathbf{P}}\mathbf{c}_{x_i}, \begin{bmatrix} \ddot{x}_i(t_1) \\ \ddot{x}_i(t_2) \\ \dots \\ \ddot{x}_i(t_n) \end{bmatrix} = \ddot{\mathbf{P}}\mathbf{c}_{x_i}, \tag{14}$$

where, $\mathbf{P}$ is a matrix formed with time-dependent basis functions (e.g polynomials) and $\mathbf{c}_{x_i}$ are the coefficients associated with the basis functions. Let $\mathbf{c}_x$ be the joint coefficient formed by stacking $\mathbf{c}_{x_i}$ for all the agents. Now, with the help of (14), we can derive the following matrix representation for optimization (22a)

$$\min \frac{1}{2}\mathbf{c}_x^T(\mathbf{Q}_x + \rho\mathbf{A}_{f_c}^T\mathbf{A}_{f_c})\mathbf{c}_x + (-\mathbf{A}_{f_c}^T\,{}^k\mathbf{b}_{f_c}^x)^T\mathbf{c}_x \tag{15a}$$
$$\mathbf{A}_{eq}\mathbf{c}_x = \mathbf{b}_{eq}^x \tag{15b}$$

Note, how only the vector ${}^k\mathbf{b}_{f_c}^x$ is shown to be dependent on the iteration index k. The various matrices and vectors involved in (15a)-(15b) are derived in the following manner.

$$\sum_{i,t}\ddot{x}_i(t)^2 \Rightarrow \frac{1}{2}\mathbf{c}_x^T\mathbf{Q}_x\mathbf{c}_x, \mathbf{Q}_x = \begin{bmatrix} \ddot{\mathbf{P}}^T\ddot{\mathbf{P}} & & \\ & \ddots & \\ & & \ddot{\mathbf{P}}^T\ddot{\mathbf{P}} \end{bmatrix} \tag{16}$$

$$x_i(t) \in \mathcal{C}_{boundary}, \forall i \Rightarrow \mathbf{A}_{eq}\mathbf{c}_x = \mathbf{b}_{eq}^x \tag{17}$$

$$\mathbf{A}_{eq} = \begin{bmatrix} \mathbf{A} & & \\ & \ddots & \\ & & \mathbf{A} \end{bmatrix}, \mathbf{A} = \begin{bmatrix} \mathbf{P}_1 \\ \dot{\mathbf{P}}_1 \\ \ddot{\mathbf{P}}_1 \\ \mathbf{P}_m \\ \dot{\mathbf{P}}_m \\ \ddot{\mathbf{P}}_m \end{bmatrix} \tag{18}$$

The matrix $\mathbf{A}$ is formed by stacking the first and last row of $\mathbf{P}$ and its derivatives, and $\mathbf{b}_{eq}^x$ is formed by stacking initial and final position, velocities and accelerations.
Similarly, we obtain the following matrix representation.

$$\sum_{i,j,t}\frac{\rho}{2}\Big(x_i(t) - x_j(t) - l_{xy}\,{}^kd_{ij}(t)\sin{}^k\beta_{ij}(t)\cos{}^k\alpha_{ij}(t) \tag{19}$$
$$+\frac{{}^k\lambda_{xij}(t)}{\rho}\Big)^2 \Rightarrow \frac{\rho}{2}\|\mathbf{A}_{f_c}\mathbf{c}_x - {}^k\mathbf{b}_{f_c}^x\|_2^2, \mathbf{A}_{f_c} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_r \end{bmatrix}$$

$$\mathbf{A}_r = \begin{bmatrix} \begin{pmatrix} \mathbf{P} \\ \mathbf{P} \\ \vdots \\ \mathbf{P} \end{pmatrix}_{\times n-r} & \begin{bmatrix} -\mathbf{P} & & \\ & \ddots & \\ & & -\mathbf{P} \end{bmatrix}_{\times n-r} \end{bmatrix} \tag{20}$$

$${}^k\mathbf{b}_{f_c}^x = l_{xy}\,{}^k\mathbf{d}\sin{}^k\boldsymbol{\beta}\cos{}^k\boldsymbol{\alpha} - \frac{{}^k\boldsymbol{\lambda_{xij}}}{\rho} \tag{21}$$

In (20), the operator $(.)_{\times n-r}$ vertically stacks the matrix $\mathbf{P}$ $n-r$ times. Similarly, the block-diagonal matrix on the r.h.s is formed by $(n-r)$ number of matrix $\mathbf{P}$. In (21), ${}^k\mathbf{d}$, ${}^k\boldsymbol{\beta}$, ${}^k\boldsymbol{\alpha}$ are formed by stacking the respective variables at all times and for all the agents. Similar construction is also followed for ${}^k\boldsymbol{\lambda}_{xij}$.

Reduction to Linear Equations: The equality constrained QP (15a)-(15b) can be reduced to a problem of solving the following set of linear equations, wherein $\boldsymbol{\mu}$ represents the dual variables associated with the equality constraints [7].

$$\overbrace{\begin{bmatrix} (\mathbf{Q}_x + \rho\mathbf{A}_{f_c}^T\mathbf{A}_{f_c}) & \mathbf{A}_{eq}^T \\ \mathbf{A}_{eq} & \mathbf{0} \end{bmatrix}}^{\widetilde{\mathbf{Q}}_x} \begin{bmatrix} \mathbf{c}_x \\ \boldsymbol{\mu} \end{bmatrix} = \overbrace{\begin{bmatrix} \mathbf{A}_{f_c}^T\,{}^k\mathbf{b}_{f_c}^x \\ \mathbf{b}_{eq}^x \end{bmatrix}}^{\widetilde{\mathbf{q}}_x} \tag{23}$$

Remark 3. *For a given ρ, the matrix on the l.h.s of (23) is independent of the iteration index k. Thus, its inverse can be pre-computed and used without any computational cost in each iteration of Algorithm 1. Furthermore, this matrix depends only on our choice of trajectory parametrization $\mathbf{P}$. Thus, the same pre-computed inverse can be used to solve trajectory optimization for any variations of start and goal positions as long as the number of agents and trajectory parametrization remains the same.*

Per-Iteration Complexity: Let, n_v be the number of decision variables of each agent (columns of $\mathbf{P}$). Let n_b be the number of boundary conditions (row of $\mathbf{A}$) for each agent along each motion axis. Then the per-iteration complexity of

```
Algorithm 1 Alternating Minimization based Multi-Agent Trajectory Optimization
1: Initialize ${}^k d_{ij}(t), {}^k\alpha_{ij}(t), {}^k\beta_{ij}$ at $k = 0$
2: while $k \leq maxiter$ or till norm of the residuals are below some threshold do
```

$$^{k+1}x_i(t) = \arg\min_{x_i(t)\in\mathcal{C}_{boundary}} \sum_{i,t} \ddot{x}_i(t)^2 + \sum_{i,j,t} \frac{\rho}{2}(x_i(t) - x_j(t) - l_{xy}{}^k d_{ij}(t)\sin{}^k\beta_{ij}(t)\cos{}^k\alpha_{ij}(t) + \frac{^k\lambda_{xij}(t)}{\rho})^2 \tag{22a}$$

$$^{k+1}y_i(t) = \arg\min_{y_i(t)\in\mathcal{C}_{boundary}} \sum_{i,t} \ddot{y}_i(t)^2 + \sum_{i,j,t} \frac{\rho}{2}(y_i(t) - y_j(t) - l_{xy}{}^k d_{ij}(t)\sin{}^k\beta_{ij}(t)\sin{}^k\alpha_{ij}(t) + \frac{^k\lambda_{yij}(t)}{\rho})^2 \tag{22b}$$

$$^{k+1}z_i(t) = \arg\min_{z_i(t)\in\mathcal{C}_{boundary}} \sum_{i,t} \ddot{z}_i(t)^2 + \sum_{i,j,t} \frac{\rho}{2}(z_i(t) - z_j(t) - l_z{}^k d_{ij}(t)\cos{}^k\beta_{ij}(t) + \frac{^k\lambda_{zij}(t)}{\rho})^2 \tag{22c}$$

$$\begin{aligned}
{}^{k+1}\alpha_{ij}(t), {}^{k+1}\beta_{ij}(t) = \arg\min_{\alpha_{ij},\beta_{ij}} \sum_{i,j,t} \frac{\rho}{2}({}^{k+1}x_i(t) - {}^{k+1}x_j(t) - l_{xy}{}^k d_{ij}(t)\sin\beta_{ij}(t)\cos\alpha_{ij}(t) + \frac{^k\lambda_{xij}(t)}{\rho})^2 \\
+\frac{\rho}{2}({}^{k+1}y_i(t) - {}^{k+1}y_j(t) + l_{xy}{}^k d_{ij}(t)\sin\beta_{ij}(t)\sin\alpha_{ij}(t) + \frac{^k\lambda_{yij}(t)}{\rho})^2 \\
+\frac{\rho}{2}({}^{k+1}z_i(t) - {}^{k+1}z_j(t) + l_z{}^k d_{ij}(t)\cos\beta_{ij}(t) + \frac{^k\lambda_{zij}(t)}{\rho})^2
\end{aligned} \tag{22d}$$

$$\begin{aligned}
\approx {}^{k+1}\alpha_{ij}(t) = \arctan 2({}^{k+1}y_i(t) - {}^{k+1}y_j(t), {}^{k+1}x_i(t) - {}^{k+1}x_j(t)) \\
{}^{k+1}\beta_{ij} = \arctan 2(\frac{{}^{k+1}x_i(t) - {}^{k+1}x_j(t)}{l_{xy}\cos{}^{k+1}\alpha_{ij}(t)}, \frac{{}^{k+1}z_i(t) - {}^{k+1}z_j(t)}{l_z})
\end{aligned} \tag{22e}$$

$$\begin{aligned}
{}^{k+1}d_{ij}(t) = \arg\min_{d_{ij}} \sum_{i,j,t} \frac{\rho}{2}({}^{k+1}x_i(t) - {}^{k+1}x_j(t) - l_{xy}d_{ij}(t)\sin{}^{k+1}\beta_{ij}(t)\cos{}^{k+1}\alpha_{ij}(t) + \frac{^k\lambda_{xij}(t)}{\rho})^2 \\
+\frac{\rho}{2}({}^{k+1}y_i(t) - {}^{k+1}y_j(t) - l_{xy}d_{ij}(t)\sin{}^{k+1}\beta_{ij}(t)\sin{}^{k+1}\alpha_{ij}(t) + \frac{^k\lambda_{yij}(t)}{\rho})^2 \\
+\frac{\rho}{2}({}^{k+1}z_i(t) - {}^{k+1}z_j(t) - l_z d_{ij}(t)\cos{}^{k+1}\beta_{ij}(t) + \frac{^k\lambda_{zij}(t)}{\rho})^2
\end{aligned} \tag{22f}$$

$$\begin{aligned}
{}^{k+1}\lambda_{xij}(t) = {}^k\lambda_{xij}(t) + \rho({}^{k+1}x_i(t) - {}^{k+1}x_j(t) - l_{xy}{}^{k+1}d_{ij}(t)\sin{}^{k+1}\beta_{ij}(t)\cos{}^{k+1}\alpha_{ij}(t)) \\
{}^{k+1}\lambda_{yij}(t) = {}^k\lambda_{yij}(t) + \rho({}^{k+1}y_i(t) - {}^{k+1}y_j(t) - l_{xy}{}^{k+1}d_{ij}(t)\sin{}^{k+1}\beta_{ij}(t)\sin{}^{k+1}\alpha_{ij}(t)) \\
{}^{k+1}\lambda_{zij}(t) = {}^k\lambda_{zij}(t) + \rho({}^{k+1}z_i(t) - {}^{k+1}z_j(t) - l_z{}^{k+1}d_{ij}(t)\cos{}^{k+1}\beta_{ij}(t))
\end{aligned} \tag{22g}$$

```
3: end while
```

step (22a) or (23) is dominated by two large matrix-vector products. The first product involves multiplying $\mathbf{A}_{f_c}^T$ with dimensions $(nn_v \times m\binom{n}{2})$ with vector ${}^k\mathbf{b}_{f_c}^x$ of $m\binom{n}{2} \times 1$, where we recall n, m to be the number of agents and length of the planning horizon respectively. The complexity of second matrix-vector product, $\widetilde{\mathbf{Q}}_x^{-1}\widetilde{\mathbf{q}}_x$ is $\mathcal{O}(n^2(n_b + n_v)^2)$ and follows similar reasoning. However, it should be noted that these are worst-case complexities without the GPU parallelization. For example, theoretically, $\mathbf{A}_{f_c}^T{}^k\mathbf{b}_{f_c}^x$ can be split into nn_v parallel computations. However, in practice, the speed-up through parallelization depends on the number of available GPU cores and other hardware limitations such as speed of CPU-GPU transfer.

The above analysis drawn for (22a) can be trivially extended to steps (22b)-(22c) as well.

2) Step (22d): Although optimization (22d) is non-convex, an approximate solution can be derived using simple geometrical intuition. Recall (12) to note that the set of feasible $(x_i(t)-x_j(t))$, $(y_i(t)-y_j(t))$ and $(z_i(t)-z_j(t))$ constitute a spheroid centered at origin with dimensions $l_{xy}d_{ij}$ and $l_z d_{ij}$. Thus, we compute ${}^{k+1}\alpha_{ij}(t)$ and ${}^{k+1}\beta_{ij}(t)$ by projecting ${}^{k+1}x_i(t)$, ${}^{k+1}y_i(t)$, ${}^{k+1}z_i(t)$ obtained from steps (22a)-(22c) onto the requisite spheroid through equations (22e). The projection satisfies the constraints on $\alpha_{ij}(t)$ by construction. For $\beta_{ij}(t)$, we simply clip the values to $[0, \pi]$.

3) Step 22f: For a given pair of agents (i, j), $d_{ij}(t)$ at different time instants are decoupled from each other. Similarly, they are also decoupled across agent pairs. Thus, (22f) splits into $m * \binom{n}{2}$ decoupled single-variable convex QPs, each of which can be solved symbolically. That is the solutions are available as analytical formulae that we can evaluate using just element-wise operations over vectors. The constraints on $d_{ij}(t)$ are ensured by simply clipping the values to $[0\ 1]$ at each iteration.

4) Step 22g : These steps update the Lagrange multipliers based on the residuals achieved at the current iteration [8].

IV. Simulation Results

Link to supplementary material `https://arxiv.org/abs/2011.04240`

A. Implementation Details

We implemented Algorithm 1 in Python using CUPY [10] and JAX [11] to accelerate linear algebra on GPUs. Specif-

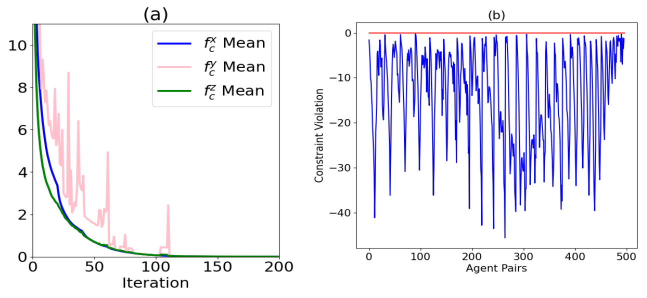


Fig. 1. Fig (a): The general trend in residual of $\mathbf{f}_c$ observed across 20 problem instances is shown in Fig. 1(a). A reliable convergence to zero validates the efficacy of Algorithm 1. In majority of the benchmarks, a maximum of 150 iterations proved sufficient to get a residual around 0.01. Fig.(b): Each pair of agent trajectories obtained by our optimizer in a 32 agent benchmark are substituted in (2c) and checked for constraint satisfaction. The constraint violations are averaged out across several 32 agent benchmarks. The figure represents the resulting mean-maximum constraint violation encountered for each agent pairs. A value less than zero for all agent pairs show that our optimizer can successfully produce collision avoiding trajectories.

ically, we use CUPY for trajectory optimization up to 32 agents, while JAX was used for a higher number of agents. Computing hardware consisted of a i7-8750 32 GB RAM desktop computer with RTX 2080 (8GB) and Nividia Jetson TX2.

We pre-computed the inverse of $\widetilde{\mathbf{Q}}_x$ in (23) for 10 different increasing values of ρ instead of just one. The higher values were used in the latter iterations of Algorithm 1. This is inspired by [18] that advocates using adaptive ρ to speed up the convergence of optimizer such as Algorithm 1. It is worth reiterating that for a given number of agents, these inverses are computed only once and can be subsequently used to optimize trajectories from arbitrary start and goal positions. For ease of implementation, we considered agents as spheres rather than spheroids. Along similar lines, we constructed the static obstacles' circumscribing sphere to incorporate them within our optimizer. For comparison with [5] and [2], we used the open-source implementation and data-set provided by the latter. For a fair comparison, we did not include any hard bounds on position, velocities, and accelerations on the implementation of [5]. This led to some reduction in the number of inequality constraints. Similarly, we also changed the so called "downwash" parameter in [2] to 1 to conform with the implementation of our optimizer. Since trajectories obtained with our optimizer and [5], [2] are at different time scales, we used the second-order finite-difference of the position as a proxy for comparing the accelerations across the three methods.

B. Benchmarks and Convergence

We test our optimizer on the following benchmarks.
Square Benchmark: The agents are placed on the edge of a square and are required to move to their anti-podal position.
Random Benchmark with static obstacles: Here the agents start and goal positions are sampled randomly. We also create a variant of this benchmark where static obstacles are placed randomly within the workspace. Due to space limitations, we present the sample trajectories obtained in the benchmarks in the supplementary results [19].

A key metric for validating Algorithm 1 is the trend in the residuals of $\mathbf{f}_c$ (recall (11c)) over iterations. It should converge to zero in a diverse set of benchmarks to ensure that the optimizer reliably computes at a collision-free trajectory. Fig.1 provides this empirical validation. The plots show the mean and one standard deviation of the residual trajectory obtained across 20 different problem instances. On average, 150 iterations were sufficient to obtain residuals around 0.01. Note that this is a norm of a vector that can possibly have tens of thousands of elements ($\binom{n}{2}m$ elements). For example, for 64 robots, each of $\mathbf{f}_c^x, \mathbf{f}_c^y, \mathbf{f}_c^z$ will have more than 10^6 elements. Thus, we observed that individual elements of the residual vector often have magnitude around 10^{-3} or lower. We can allow the optimizer to run for more iterations to obtain even lower residuals. Our implementation uses an additional practical trick. We inflate the radius of the agents by four times the typical residual we observe after 100 iterations of our optimizer. In practice, this results in a increase of around $4cm$ in agents' dimensions.

To validate the feasibility of the obtained trajectories, we checked them for the satisfaction of the classical collision avoidance inequality (2c). We computed the maximum constraint violation over all time instants for each agent pair and then averaged it out across the different benchmarks. The results are summarized in Fig.1(b). A value less than zero for mean maximum constraint violation indicates successful collision avoidance.

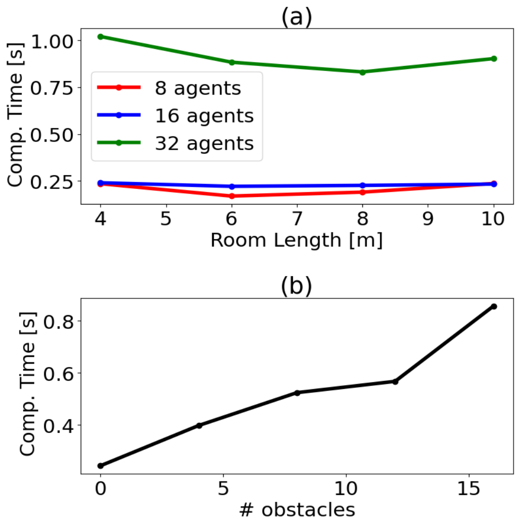


Fig. 2. Fig. (a) shows computation time for a varying number of agents for benchmarks where we sample start and goal positions from a square with varying lengths. Fig. (b) shows the linear scaling of computation time with obstacles for a given number of agents.

C. Computation Time

Fig. 2 (a) presents the mean computation time of our optimizer for a varying number of agents (with radii 0.4 cm) as a function of how closely packed the initial and final positions are. To be more precise, we sampled random initial and

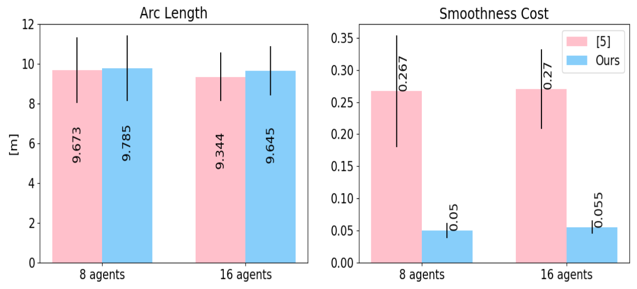


Fig. 3. Comparison of our optimizer with [5] in terms of arc-length and smoothness of the obtained trajectories. The arc-lengths are similar across both the approaches but our optimizer achieves smoother trajectories. Note that the smoothness cost is computed as the norm of the second-order finite-difference of the position of the respective agents at different time instants.

final positions in a square room of varying lengths to create problem instances of varying complexity levels. As can be seen, even in the most challenging instance, our optimizer could compute trajectories for 32 agents in around 1s.

Fig. 2(b) shows the computation time for 16 agents with different number of static obstacles. Our optimizer shows an almost linear scaling in computation time. This is because incorporation of static obstacles only affects the computation cost of obtaining $\mathbf{A}_{f_c}^{T}{}^{k}\mathbf{b}_{f_c}^{x}$ in (23). Furthermore, the dimensions of both the matrix and the vector increase linearly with the number of obstacles.

D. *Comparison with Reciprocal Velocity Obstacle (RVO) [?]*

RVO is a local, reactive one-step planning method used extensively in multi-agent navigation. It is slightly orthogonal to our optimizer that solves the more complex, global multi-agent trajectory optimization while considering collision avoidance over several steps (100 in our implementation). Despite this, we benchmark our optimizer with RVO with the sole motivation to create a baseline comparison of trajectory quality. The computation-time of RVO is unsurprisingly much faster than our optimizer.

The results are summarized

E. *Comparison with [5]*

Fig. 3 presents a comparison of the trajectory quality obtained with our optimizer and [5]. Although both the optimizer converge to different trajectories, the arc-length statistics observed across all the agents are very similar. Furthermore, our optimizer outperforms [5] in terms of trajectory smoothness cost.

The mean computation time required by the SCP of [5] in different 8 agent square benchmarks was around 6.79 s. In contrast, our optimizer was 28 times faster and took only 0.242 s. The computation time difference was even starker when the number of agents increased to 16, with the SCP of [5] taking 160.76s while our optimizer was 613 times faster at 0.262 s.

The trend in computation time is not surprising as even state-of-the-art primal-dual interior-point solvers for QP scale cubically with the total number of inequality and equality constraints. Furthermore, the number of inequality constraints stemming from collision avoidance will itself scale as $\binom{n}{2}$. As mentioned earlier, our optimizer by-passes this intractability by pre-computing the expensive matrix inverses and parallelizing matrix-vector product on GPUs. It is essential to point out that GPU accelerations of SCP of [5] is a challenging open problem on its own. We conjecture this to be the motivation behind the same authors adopting the gradient descent approach for leveraging GPUs [14].

F. *Comparison with [2]*

Fig. 4 presents the most important result of this paper, wherein we compare our optimizer with the current state of the art [2]. The cited work adopts a sequential approach but with a batch of agents. It also leverages the parallel QP solving ability of CPLEX [20] on multi-core CPUs. Our optimizer produces trajectories of similar smoothness as [2] but with substantially lower arc-lengths. This trend can be attributed to the reduced feasible space accessible to a sequential approach. More surprisingly, our optimizer also outperforms [2] in terms of computation time on 16 and 32 agent benchmarks. On the 64 agent benchmark, our optimizer is only marginally slower than [2]. We reiterate that it is essential to observe these timings with the context that our optimizer performs a much more rigorous search than [2]. The trends in computation time can be understood in the following manner. For a lower number of agents, the computation time of [2] is dominated by the niche trajectory initialization it leverages from sampling-based planners. Furthermore, for a lower number of agents, the overhead of CPU parallelization is also significant. But these overheads are easily offset by the computation speed-up achieved for a higher number of agents.

G. *Performance on Jetson TX2*

Table I shows the computation time for a different number of agents for the square benchmark on Nvidia Jetson TX2. The start and goal positions are sampled from a square of length $8m$. The timings indicate that our optimizer allows for fast on-board decision making for up to 16 agents. Moreover, even for 32 agents, the computation time is small enough to be useful for practical applications.

V. Conclusions and Future Work

In this paper, we fundamentally improved the scalability of joint multi-agent trajectory optimization. Simultaneously, our algorithm is simple to implement and requires the computation of only a few matrix-vector products and element-wise operation over vectors. We achieved this by leveraging hidden geometrical and convex structures in the problem. We outperformed state of the art in joint and sequential approaches in terms of trajectory quality and computation time. One limitation of our optimizer is that it has a constant computational overhead stemming from loading large scale matrices at run-time. However, this limitation has limited practical impact as it needs to be done only once for a given number of agents.

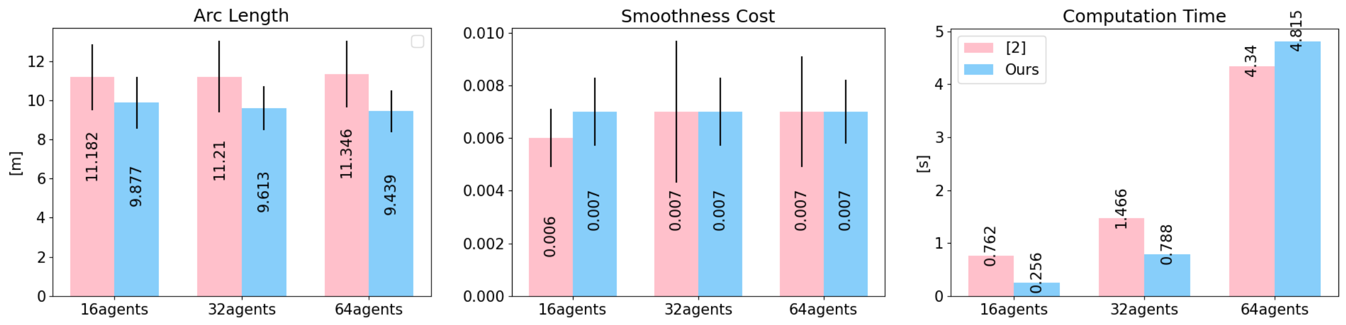


Fig. 4. Comparisons with current state of the art [2]. Our optimizer outperforms [2] in terms of trajectory arc-lengths. Even more importantly, it also outperforms in terms of computation time on several benchmarks.

We are extending our optimizer to work with complicated geometries like an elongated rectangle through multi-circle approximation. Along similar lines, we also aim to extend the results to non-linear agents like autonomous cars by building on bi-convex approximations proposed in our prior work [21].

TABLE I
COMPUTATION TIME ON NVIDIA-JETSON TX2

Square Benchmark	Comp. Time [s]
8 agents, radius = 0.1	1.01
8 agents, radius = 0.6	1.320
8 agents, radius = 1.2	1.270
16 agents, radius = 0.3	2.10
16 agents, radius = 0.6	2.34
32 agents, radius = 0.25	7.70

TABLE II

Number of agents	Benchmark	Arc-length	smoothness cost
16 agent	RVO	9.491	0.217
	Our	9.877	0.062
32 agents	RVO	9.348	0.26
	Our	9.613	0.06
64 agents	RVO	9.362	0.228
	Our	9.439	0.064

REFERENCES

[1] Y. Chen, M. Cutler, and J. P. How, “Decoupled multiagent path planning via incremental sequential convex programming,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 5954–5961.

[2] J. Park, J. Kim, I. Jang, and H. J. Kim, “Efficient multi-agent trajectory planning with feasibility guarantee using relative bernstein polynomial,” in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 434–440.

[3] C. E. Luis and A. P. Schoellig, “Trajectory generation for multiagent point-to-point transitions via distributed model predictive control,” *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 375–382, 2019.

[4] L. Ferranti, R. R. Negenborn, T. Keviczky, and J. Alonso-Mora, “Coordination of multiple vessels via distributed nonlinear model predictive control,” in *2018 European Control Conference (ECC)*. IEEE, 2018, pp. 2523–2528.

[5] F. Augugliaro, A. P. Schoellig, and R. D’Andrea, “Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach,” in *2012 IEEE/RSJ international conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 1917–1922.

[6] J. Bento, N. Derbinsky, J. Alonso-Mora, and J. S. Yedidia, “A message-passing algorithm for multi-agent trajectory planning,” in *Advances in neural information processing systems*, 2013, pp. 521–529.

[7] B. O’Donoghue, G. Stathopoulos, and S. Boyd, “A splitting method for optimal control,” *IEEE Transactions on Control Systems Technology*, vol. 21, no. 6, pp. 2432–2442, 2013.

[8] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Foundations and Trends® in Machine learning*, vol. 3, no. 1, pp. 1–122, 2011.

[9] P. Jain, P. Kar *et al.*, “Non-convex optimization for machine learning,” *Foundations and Trends® in Machine Learning*, vol. 10, no. 3-4, pp. 142–336, 2017.

[10] R. Nishino and S. H. C. Loomis, “Cupy: A numpy-compatible library for nvidia gpu calculations,” *31st confernce on neural information processing systems*, p. 151, 2017.

[11] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne, “JAX: composable transformations of Python+NumPy programs,” 2018. [Online]. Available: http://github.com/google/jax

[12] T. Lipp and S. Boyd, “Variations and extension of the convex–concave procedure,” *Optimization and Engineering*, vol. 17, no. 2, pp. 263–287, 2016.

[13] F. Gao and S. Shen, “Quadrotor trajectory generation in dynamic environments using semi-definite relaxation on nonconvex qcqp,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 6354–6361.

[14] M. Hamer, L. Widmer, and R. D’andrea, “Fast generation of collision-free trajectories for robot swarms using gpu acceleration,” *IEEE Access*, vol. 7, pp. 6679–6690, 2018.

[15] E. Ghadimi, A. Teixeira, I. Shames, and M. Johansson, “Optimal parameter selection for the alternating direction method of multipliers (admm): quadratic problems,” *IEEE Transactions on Automatic Control*, vol. 60, no. 3, pp. 644–658, 2014.

[16] A. K. Singh, R. Ghabcheloo, A. Muller, and H. Pandya, “Combining method of alternating projections and augmented lagrangian for task constrained trajectory optimization,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 7568–7575.

[17] X. Qian, F. Altché, P. Bender, C. Stiller, and A. de La Fortelle, “Optimal trajectory planning for autonomous driving integrating logical constraints: An miqp perspective,” in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2016, pp. 205–210.

[18] Y. Xu, M. Liu, Q. Lin, and T. Yang, “Admm without a fixed penalty parameter: Faster convergence with new adaptive penalization,” in *Advances in Neural Information Processing Systems*, 2017, pp. 1267–1277.

[19]

[20] C. Optimizer, “High-performance mathematical programming solver for linear programming, mixed integer programming, and quadratic programming,” *IBM ILOG CPLEX Optimization Studio, Version*, vol. 12, 2011.

[21] A. K. Singh, R. R. Theerthala, M. B. Nallana, U. K. R. Nair, and K. Krihna, “Bi-convex approximation of non-holonomic trajectory optimization,” in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

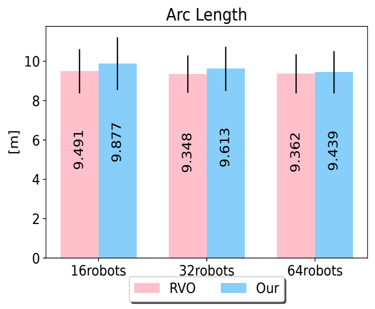
Suplimentary Results

A. Benchmarks

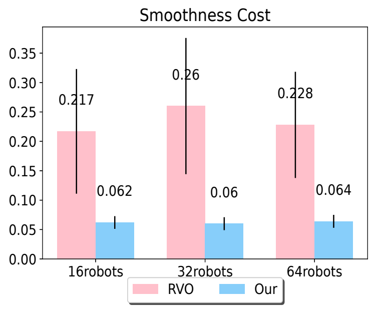
Fig. 6(a)-6(c) show the typical benchmarks employed in our experiments. In Fig.6(a), the agents are placed in a square and are required to navigate to their antipodal positions. In Fig. 6(b), the start and goal positions are sampled randomly. Fig. 6(c) repeats the previous benchmark by adding random static obstacles. Each of the benchmarks was evaluated with a different number of agents with a diverse range of radii. Fig. 7 shows the snapshots of 32 agents exchanging positions in a narrow hallway. Interestingly, our optimizer naturally leads to a line formation pattern among the agents in this benchmark.

B. Comparison with reciprocal velocity obstacles

It should be mentioned that there are main differences between reciprocal velocity obstacles (RVO) and the proposed trajectory optimization algorithm. At first RVO is a local navigation collision free algorithm which tracks one time-step at a time. However, our algorithm solves a global problem and computes trajectories from start point to goal point in multiple steps. In addition, it is not possible to execute RVO algorithm on drones or cars as velocity profiles are jerky. Despite the mentioned differences, we did a comparison between RVO and propose method in terms of Arc length and smoothness cost. A glance at the Fig. 5(b) reveals that both algorithms almost have same arc length for varying number of agents. However, Fig. 5(a) depicts that trajectories obtained for the proposed algorithm have much lower smoothness cost.

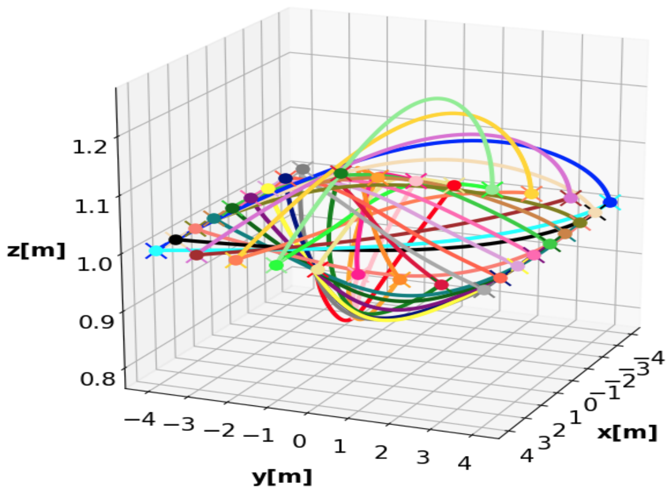


(a)

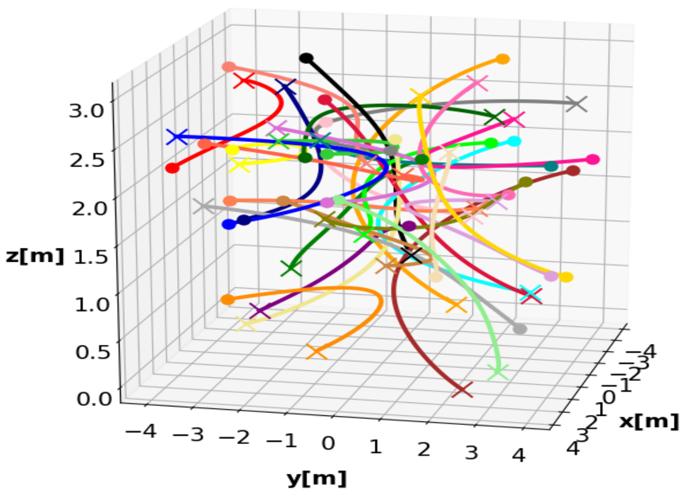


(b)

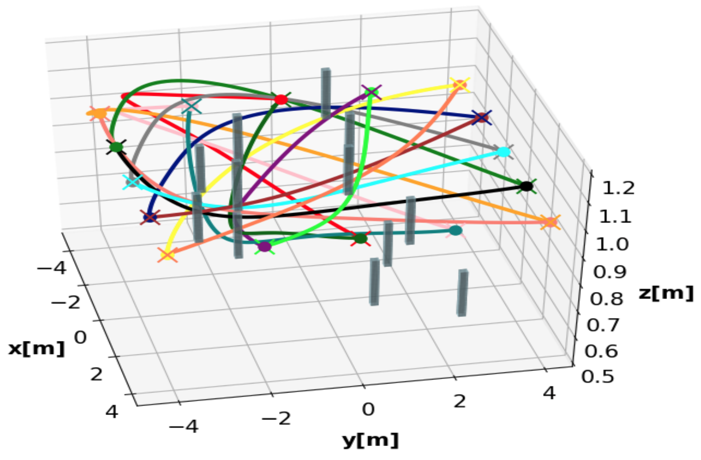
Fig. 5. Comparison of our optimizer with RVO method in terms of arc-length and smoothness of the obtained trajectories.



(a)



(b)



(c)

Fig. 6. Different benchmarks employed in our experiments along with some typical trajectories obtained with our optimizer is shown. The start and goal positions are marked with a "X" and a "o" respectively.

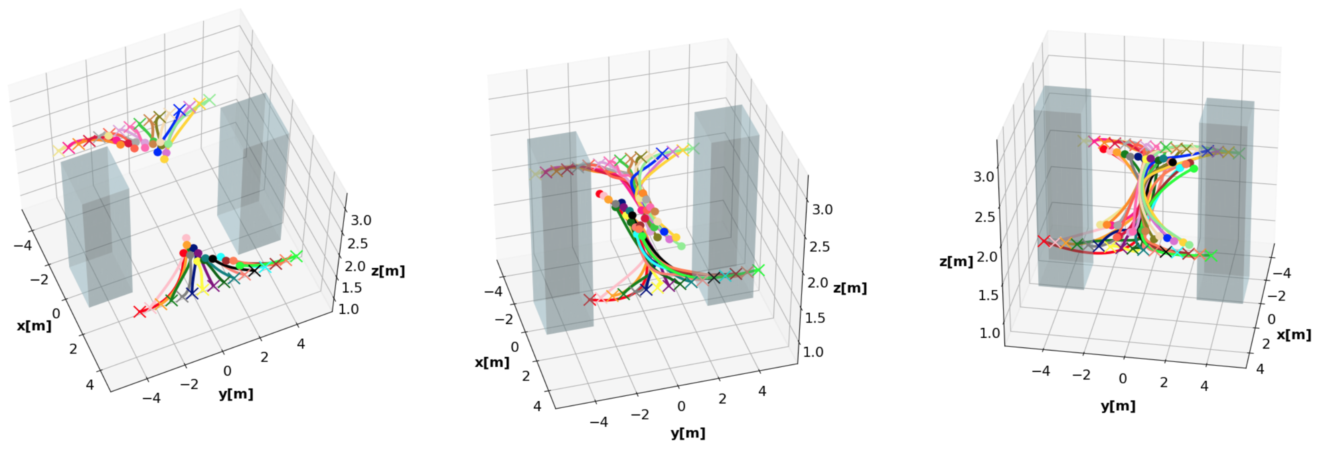


Fig. 7. Collision avoidance snapshots of 32 agents exchanging positions in a narrow hallway is shown. The start and goal positions are marked with a "X" and a "o" respectively.