Discrete and Continuous Trajectory Optimization Methods for Complex Robot Systems

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Electronics and Communication Engineering
by Research

by

Dipanwita Guhathakurta
2018112004
dipanwita.g@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
March, 2023

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Discrete and Continuous Trajectory Optimization Methods for Complex Robot Systems" by Dipanwita Guhathakurta, has been carried out under my supervision and is not submitted elsewhere for a degree.

March 14, 2023

Adviser: Prof. K. Madhava Krishna

March 14, 2023

Co-Adviser: Prof. Arun Kumar Singh

One Small Step for Robot, One Giant Leap for Mankind

Acknowledgements

I wish to express my gratitude to my adviser, Prof. K. Madhava Krishna from the Robotics Research Centre at IIIT Hyderabad, for being a huge pillar of support, from helping me recognize my aptitude for research to allowing me to hone my skills in multiple sub-domains under Robotics. I am also immensely thankful to my co-adviser, Prof. Arun Kumar Singh from the University of Tartu, Estonia, for guiding me in every step of my research journey, starting from formulating research problems to hands-on pair programming. I would also like to thank Prof. Ponnurangam Kumaraguru for boosting my confidence in research paper writing and allowing me to explore diverse domains during my Independent Study under his supervision.

Next, I would like to thank the faculty at IIIT Hyderabad for providing rigorous training in mathematics and Electronics at the undergraduate level that helped me build my foundations in Robotics. I owe my understanding of Robotics to the project-based electives taught by Prof. Harikumar Kandath, Prof. K. Madhava Krishna, and Prof. Spandan Roy and to the Robotics Summer School organized by my seniors.

I would also like to convey my gratitude to Fatemeh Rastgar, a Junior Ph.D. Researcher under Prof. Singh, who has supported me as a mentor, helping me gain a strong understanding of multi-robot planning algorithms and GPU-acceleration libraries. My teammates in the manipulator project from RRC - Vishal Reddy Mandadi, Md. Nomaan Qureshi, Bipasha Sen, Aditya Aggarwal, and Kallol Saha have also played a significant role in my understanding of physics simulators and Reinforcement Learning-based robot control. My student mentors - Mithun Babu Nallana, Jyotish P, and Raghu Ram Theerthala, helped me take my first steps in robot path planning at the end of my second year and patiently reviewed my implementations of state-of-the-art algorithms. In addition, my batchmates and fellow research students at RRC have often readily contributed suggestions to improve my research work and validated my ideas. Aditya Sharma Meduri has helped me get acquainted with ROS and Gazebo to run simulations for my paper. My batchmates and friends, Vedant Mundheda, Karan Mirakhor, Rahul Swayampakula, Nipun Wahi and Manav Bhatia have constantly offered their feedback on my research projects as teammates in course projects and external competitions. They have also supported me throughout my degree, helping me balance my academics with research, along with my social life on campus.

Finally, I would like to thank my mother and grandmother for constantly supporting me and believing in me.

Abstract

Path planning for autonomous systems is a fundamental problem in robotics, especially when the task assigned to a robot is heavily dependent on the motion of one or more of its constituent parts. Efficient path planning requires fast adaptation to changes in the environment and generalizability to a variety of tasks while ensuring the kinematic and dynamic constraints of the robot and its workspace are respected. Present-day planning algorithms for robots fail to achieve real-time performance for complex systems such as multi-robot systems and multi-jointed robot manipulators. In fact, the path planning objectives for these complex systems often present mathematical infeasibilities when posed as an optimization problem or become computationally cumbersome to compute by pure sampling. Further, these algorithms do not generalize to the task of collision avoidance against a wide variety of obstacles for multi-robot systems with a large number of robots or robots with high-dimensional articulation. The primary focus of this dissertation is to present computationally-tractable solutions to the trajectory optimization problem for both multi-robot systems and manipulators. We explore gradient-based and stochastic optimization techniques and perform mathematical reformulations to adapt them to a wide variety of applications.

First, we provide the requisite background in robot path planning, including robot kinematics, trajectory representation methods, and collision-avoidance techniques. We also discuss state-of-the-art methods in gradient-based trajectory optimization and stochastic trajectory optimization. Next, we present a distributed GPU-based multi-agent trajectory optimizer that first converts the gradient-based multi-agent trajectory optimization problem into a set of simple matrix-matrix products and then leverages parallel computations over GPUs to accelerate these computations. We demonstrate through a large number of qualitative and quantitative experiments in simulation that our distributed algorithm outperforms existing sequential trajectory optimizers or sampling-based methods for multi-robot planning. Next, we design a collision-aware path planner for robot manipulators operating in the high-dimensional joint space and demonstrate its application across scenes with different types and numbers of obstacles. We finally couple this stochastic optimization-based path planner with a low-level controller for the task of pushing objects on a table using a robot manipulator. We also make our software for multi-agent path planning public for open-source development so that our algorithm can be used for further research in this domain.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scope of the Thesis

The task of path planning for a robot involves finding an optimal start-to-goal trajectory that enables autonomous intelligent motion. A path planning algorithm for an autonomous robot performs the task of a pilot operating an aircraft, first understanding the environment, then estimating a high-level path from its start to destination, and, finally, deciding on the controls necessary to execute this path. It must be aware of the dynamic and hardware constraints associated with the robot mechanical system while interacting heavily with its environment and adapting its motion to potential changes in its surroundings. The ability of drones, autonomous cars, and personal robots to maneuver through complex environments relies on the following components as depicted in Fig 1.1:

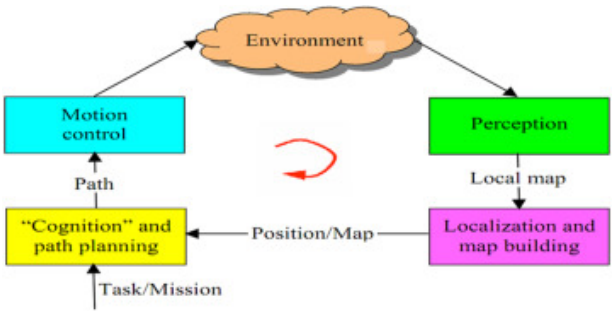


Figure 1.1: An overview of the components of an autonomous robot system. The perception component gathers data from the environment and sends a local map to the robot for detecting its own pose, which is eventually sent into the task/path planning module. The planned path or task sequence is finally sent for execution to the low-level motion controller. *Source: [5]*

1. A data ingestion system consisting of a fusion of sensors (camera, LiDAR, Infrared Sensors) that form the sensory system of the robot, allowing it to interact with its surroundings.

2. A localization system that allows the robot to map its environment and calibrate its own pose with respect to it.

3. A global path planner that returns a high-level path from the start to destination, similar to Google Maps planning the shortest route between two cities.

4. A low-level motion planner that takes the high-level path plan, returns a sequence of controls, such as linear and angular velocities, and sends it to the controller that interfaces with the robot hardware.

While points 3 and 4 have been widely explored for individual mobile robots, there are multiple challenges associated with generalizing existing approaches for complex problems such as multiple interacting robots or robots with high-dimensional articulation. The goal of this thesis is to discuss some of these challenges and propose novel solutions.

1.1.1 Research problems Tackled

Solving the high-level path-planning problem introduces the challenge of representing a start-to-goal robot trajectory efficiently. To put it simply, a trajectory should define a sequence of robot states, stacked with respect to time. For fixed-base robots, such as tabletop manipulators, the trajectory needs to be planned in terms of its moving components, i.e. the joints and the end-effector. So, a manipulator with n moving joints would have to plan a trajectory consisting of n-dimensional position vectors. The motion planner is equipped with the knowledge of the robot's kinematics(the laws governing its motion) and needs to find a sequence of controls(such as velocity, acceleration) that executes this high-level path. The optimality of a trajectory is determined by the performance of both the path planner and the motion planner, judged by several heuristics (such as those discussed in Section 2.3.4), as well as the ability to avoid collisions with other interacting objects. Mathematically, finding the optimal trajectory can be represented as an optimization problem with a predefined task-based objective function, collision avoidance constraints, kinematic and dynamic bounds, and boundary constraints, stretching across both the path planner and the motion planner.

The difficulty of solving this trajectory optimization problem depends on the following:

- the complexity of the robot's own kinematics,

- the extent of its interactions among its moving components and with its environment

In this thesis, we consider solving the trajectory optimization for two challenging applications - multi-robot systems with heavy interactions among the constituent robots, and multi-jointed manipulators with complex kinematics. Both of these systems interact constantly with the environment. Broadly, we tackle the following research objectives:

T1 *To accelerate robot trajectory optimization for real-time performance*

For robots continuously interacting with their environment, generating a high-level path and motion controls to be followed in the future may not guarantee optimal motion. The robot should be able to adapt its trajectory with respect to changes in the environment, which necessitates repeated re-computation of the trajectory optimization problem. It is, therefore, important to design an optimizer fast enough to allow the robot to re-compute its path. For instance, an individual quadrotor in a drone swarm would need to update its trajectory plan by predicting the trajectories of the other drones, or a self-driving car would re-compute its route based on traffic on the road. We target to achieve real-time generation of start-to-goal robot trajectories and demonstrate mathematical techniques that allow this computational acceleration.

T2 *To design highly-parallelized trajectory optimizers for multi-robot systems*

The path planning problem for multi-robot systems requires us to repeat the core per-robot trajectory optimization over multiple agents while considering interactions among the robots as well as with the environment. For each robot, the other robots in the system can be thought of as dynamic obstacles. Collision avoidance between a pair of robots involves predicting each other's trajectories or communicating their current states continuously. The pairwise collision constraints scale quadratically with an increase in the number of robots and the system-wide optimization problem eventually becomes intractable. We attempt to parallelize trajectory computation for multi-robot systems in batches and accelerate the per-batch optimization through fast matrix operations over GPUs and approximations of collision constraints.

T3 *To extend trajectory optimization methods for motion planning in robots with complex kinematics such as manipulators*

For multi-jointed robot manipulators, the high-level trajectory is planned for each of the movable joints, either as a sequence of joint position vectors or joint angle vectors. For the motion planner to execute this trajectory, the kinematics of the robot brings several constraints in terms of joint angle limits, velocity limits and so on. The high dimensionality of motion spaces and the complexity of dynamic and kinematic constraints makes continuous-time trajectory optimization computationally expensive and slow. We adopt a unique approach to distribute the path planning and motion planning tasks into two levels of optimization. Our path planner samples points from a distribution to guess candidate trajectories and simulates them through the motion planner to intelligently update the sample distribution along the optimization iterations.

1.2 Motivation

This section discusses the inspiration behind some of the methods discussed in the subsequent chapters of this dissertation. We present the challenges that arise in solving the path-planning problem

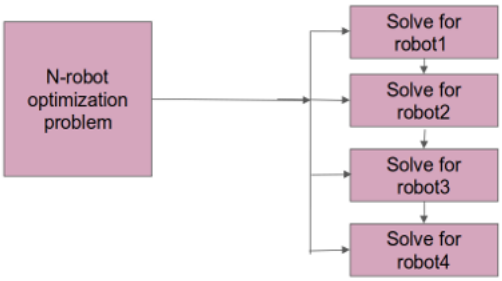


Figure 1.2: For sequential optimization, an implicit ordering is maintained among the robots, and each robot uses the trajectories of other robots computed before it to compute the collision avoidance constraints while planning its own path.

for multi-robot systems and multi-jointed robot manipulators and discuss the limitations of existing approaches.

1.2.1 Fast Trajectory Optimization for Multi-Robot Systems

The problem of coordination planning between multiple robots in robot fleets or drone swarms has been tackled through two optimization paradigms in literature:

- *Centralized Optimization*, where the trajectories of all robots are computed together. An implicit ordering is maintained among the robots for sequentially optimizing their trajectories, where each robot avoids collision by using the optimal trajectories of other robots computed before it. This sequential solution to the multi-robot planning problem has been depicted in Figure 1.2.

  For joint optimization, the system-wide trajectory planning is performed simultaneously as a single large-optimization problem returning trajectories for all the robots in one go. The optimization is performed over the larger feasible solution space, involving trajectory variables for all the robots together.

- *Distributed Optimization*, where the trajectory planning problem of each robot is completely decoupled from the other robots. Each robot makes predictions about the others' trajectories based on a prediction model to compute collision-free paths. That is, for an n-robot system, each robot considers $n-1$ collision constraints with other robots while planning its trajectory. The per-robot optimization problem becomes simpler to solve, and this is repeated over all n robots.

Centralized optimization approaches, whether sequential or joint, discussed in [6], [7], fail to scale efficiently with an increase in the number of robots in the system. The pairwise collision avoidance constraints among the robots scale quadratically with the number of robots. Distributed approaches, on

the other hand, are fast to compute paths but susceptible to digressions between the trajectory predictions and actual robot trajectories and have been shown in [8] to be less effective in collision avoidance.

To achieve our goal discussed in 1.1.1 of real-time trajectory planning for multiple robots, we choose the distributed optimization paradigm owing to its speed. We aim to achieve computational acceleration on the following two fronts:

1. Speeding up the per-robot trajectory optimization
2. Avoiding having to loop over all robots while computing individual paths by parallelizing decoupled trajectory computations.

To achieve 1, we approximate the trajectory optimization problem as a sequence of matrix operations so that effectively we only need to solve a set of linear equations to obtain the trajectory of each robot. We then vectorize our computations and solve the planning problem for all robots by stacking up independent per-robot problems into a large block matrix. This removes the need to loop over an increasing number of robots and returns trajectories for all robots in a single shot. We cache computationally expensive operations such as block matrix inversing through clever mathematical reformulations of the optimization problem and leverage GPUs to accelerate matrix multiplications. We also compare our approach against a diverse range of state-of-the-art multi-robot path planners and achieve a 2x computational acceleration while not compromising on other metrics such as collisions and trajectory length, as will be shown in Section 3.4.

1.2.2 Trajectory Optimization for Robot Manipulators

Motion planning for robot manipulators involves planning a sequence of controls to achieve a desired objective, for example, pushing an object from one point to another or picking and placing an object at a target location. The motion planner executes the trajectory plan coming from a high-level path planner. Finding the optimal high-level plan is coupled with calculating a permissible set of controls that allow the robot to execute it, and thus, a trajectory optimizer should be aware of the motion controls to obtain an optimal feasible trajectory.

We first use the Stochastic Trajectory Optimization(STO) method as a high-level path planner to generate collision-free trajectories for manipulators. STO optimizes the control inputs of a robot to minimize a cost function that captures the desired task objectives using a probabilistic approach. It generates a large number of random trajectories and evaluates their cost function values. It then selects the best-performing trajectories and generates new random trajectories around them to further explore the search space. This process is repeated until a satisfactory solution is found. The STO algorithm is particularly useful for manipulator path planning since it can work with high-dimensional motion spaces and non-differentiable and discontinuous cost functions, It is also robust to uncertainties and disturbances in the environment and sensor noise, which gives it advantages in complex and uncertain planning tasks over gradient-based optimizers.

We design a novel cost function, incorporating collision penalty, joint limit cost, and trajectory length for STO to give us a high-level trajectory. We then combine this high-level path planner with a low-level motion planner designed specifically for non-prehensile tasks, such as pushing an object on a tabletop from a start position to a desired goal position. This low-level planner decides an optimal set of controls to enable the manipulator to execute the planned STO trajectory, and sends feedback to the STO algorithm to adapt its cost according to the actual low-level motion.

1.3 Thesis Layout

C1 This is the introductory chapter, which discusses the scope of the work carried out in this thesis in terms of path planning and motion planning, addresses the research problems we are tackling and outlines the motivation behind some of the methods we have adopted to solve them.

C2 In this chapter, we present a background of standard algorithms used for continuous-time and discrete-time trajectory optimization for individual robots that form a necessary prerequisite to understand the ideas discussed in the following chapters. We also discuss two primary paradigms in trajectory optimization - gradient-based optimization and sampling-based optimization, and compare their applications in complex robot systems.

C3 We present the first contribution of this thesis - Fast Joint Multi-Robot Trajectory Optimization by GPU Accelerated Batch Solution of Distributed Sub-Problems. This algorithm employs mathematical reformulations to allow efficient caching of gradients and convex approximations for collision constraints in multi-robot systems and computes trajectories in near real-time for as many as 36 robots.

C4 This chapter presents the second contribution of this thesis - a sampling-based trajectory optimization framework for global path planning and its application in designing a bi-level trajectory optimizer for push actions in robot manipulators, generalizable to avoid collisions against a range of objects. We test and show the efficacy of our optimizer in complex tabletop rearrangement scenarios for a few common robot manipulators.

C5 We conclude with a summary of methods and results discussed in this thesis and the scope of extension of this work in the future.

Chapter 2

Gradient-based and Sampling-based Trajectory Optimization

This chapter introduces some fundamental background concepts that are essential for understanding the methods adopted in the subsequent chapters.

We first discuss the differences in kinematics among different types of robots and the necessity of designing a suitable trajectory representation respecting these differences prior to formulating a path planning problem. We talk about continuous-time trajectory representation using Cubic Splines and Bernstein polynomials, which are leveraged extensively in both our GPU-based multi-robot optimizer and the bi-level manipulator optimizer. Next, we introduce how to pose a trajectory optimization problem and explore two broad types of optimization - Gradient-based and Sampling-based. We contrast the performance of these two paradigms for various applications and then explore collision avoidance constraints to the trajectory optimization problem. Under standard collision avoidance techniques, we discuss two approaches - distance-based Collision Avoidance, and Time-scaling. We also coin a few standard evaluation metrics for judging the quality of a trajectory planned by an optimizer.

Next, we discuss Multi-Robot Path Planning, starting with real-world multi-agent systems and the challenges involved in extending standard path planning approaches directly to them. We analyze a few existing multi-robot path planners and discuss their advantages and weaknesses. Finally, we provide a brief introduction to batch trajectory optimization and commercial off-the-shelf GPU-based mathematical libraries to prime the reader for the next chapter on our GPU-accelerated optimizer.

Next, we move on to the path planning and motion planning for robot manipulators and discuss state-of-the-art optimization methods used for manipulator planning. We offer a detailed explanation of methods such as CHOMP[9], TrajOpt[10], STORM[11] and Cross-Entropy Methods(CEM)[12]. These optimization techniques form the backdrop for the Via-Point Stochastic Trajectory Optimization(VP-STO)[13] method, which forms a crucial component in our bi-level optimization framework for push actions in manipulators.

2.1 Mobile Robot Kinematics

Kinematics equations dictate how given a set of controls, the state of a robot changes. Thus if a robot starts at a given state, its kinematics will tell us what will be the robot state if a set of controls are executed. These kinematic laws are specific to the design and motion of the robot. Broadly, mobile robots (those in which a full-body displacement occurs during motion) can be classified into the following two types based on their kinematics:

2.1.1 Holonomic Robots

For a robot with n controllable degrees of freedom, it is considered holonomic if it is capable of moving in any of those n dimensions, i.e., the number of degrees of freedom is equal to the number of controllable degrees of freedom. Let us demonstrate this with the help of an example. Considering an omnidirectional robot fitted with special wheels so that it is capable of moving in any direction on a 2-D plane. The 2-D kinematics equation for the robot that started moving from the point (x_0, y_0) can, therefore, be written as follows:

$$x = x_0 + \dot{x} * \delta t \tag{2.1a}$$

$$y = y_0 + \dot{y} * \delta t \tag{2.1b}$$

where $\dot{x}$ and $\dot{y}$ denote the velocities in the two axes and δt indicates a time step. A few examples of common holonomic robots include drones (holonomic in 3-D), omnidirectional robots, etc.

2.1.2 Non-holonomic Robots

Non-holonomic robot systems are characterized by constraint equations that limit the degrees of freedom. Thus the permissible degrees of freedom for nonholonomic robot systems are less than the controllable degrees of freedom. A typical example of a non-holonomic system in 2-D is a differential drive mobile robot that is incapable of slipping in the lateral direction. The non-holonomic constraint is written as below:

$$\dot{y} = \dot{x} \tan \theta \tag{2.2}$$

The controls for a non-holonomic system are given by the linear velocity v and the angular velocity ω, giving rise to the following kinematics equations:

$$x = x_0 + v\delta t \cos\left(\theta_0 + \int \omega \delta t\right) \tag{2.3a}$$

$$y = y_0 + v\delta t \sin\left(\theta_0 + \int \omega \delta t\right) \tag{2.3b}$$

2.2 Trajectory Representation

A trajectory consists of state vectors for each moving component of the robot, stacked with respect to time. For example, for a mobile holonomic robot in 2-D, each point in the trajectory denotes a 2-D position of the centre of the robot, consisting of X and Y position coordinates. As the robot moves with time, this centre position changes, giving rise to a sequence of 2-D vectors corresponding to different timestamps. We could represent the X-coordinate position as a time-varying sequence $g_x(t)$ and the Y-coordinate position as $g_y(t)$. Thus the trajectory is of the following form:

$$g(t) = \begin{bmatrix} g_x(t_1) & g_y(t_1) \\ g_x(t_2) & g_y(t_2) \\ ... & ... \\ g_x(t_f) & g_y(t_f) \end{bmatrix} \tag{2.4}$$

For a manipulator with n movable joints, the trajectory would be a time-parameterized sequence of n-dimensional vectors, each denoting either a joint position or joint angle at a given timestamp.

We could also use the kinematics equations to transform the trajectory from state space to control space. The control space contains controls as independent time-varying sequences. We can calculate the robot states from them to denote the trajectory. This technique is particularly useful for non-holonomic robots, where the robot state is governed by the linear velocity $v(t)$ and the angular velocity $\omega(t)$ controls. Thus while the robot states in different dimensions may be kinematically coupled to each other (such as in eqn.2.2), these controls are independent and can be used to represent the trajectory.

The time-varying controls or robot states can be represented as a time-parameterized function or a discrete sequence of values over time as discussed below:

2.2.1 Continuous-time representation

Here, each time-varying robot state is represented through a function $f : t \mapsto S$ where S denotes the allowed set of states of the robot and t extends over the time of observation. If we denote time-varying controls through the function f, then S would denote the allowed set of controls of the robot.

The function f and its derivatives should satisfy the boundary limits on the robot states and controls as dictated by the robot kinematics. For example, for a holonomic robot in 2-D, we could have different functions f_x and f_y denoting the X and Y positions of the robot with respect to time. The choice of this function forms a very important design choice in most path-planning applications. Some of the commonly used functions used to represent holonomic and non-holonomic mobile robot trajectories include:

2.2.1.1 Cubic Spline

The function f is considered to be a 3rd-order polynomial in t with four weights, as depicted below:

$$f(t) = w_0 + w_1 t + w_2 t^2 + w_3 t^3 \tag{2.5}$$

To solve for the weights, we would require a minimum of four equations, i.e. four deterministic states of the robot at predefined timestamps. To give an idea, suppose we are dealing with the trajectory of a holonomic robot. We could know start and goal positions $f(t_0)$, $f(t_f)$, as well as the start and goal velocity values $f(\dot{t}_0)$ and $f(\dot{t}_f)$ for a trajectory between time t_0 and t_f forming the boundary state vector A and solve the following system of linear equations using the basis polynomial matrix B to compute the weights $W = [w_0, w_1, w_2, w_3]$.

$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ t_0 & t_f & 1 & 1 \\ t_0^2 & t_f^2 & 2t_0 & 2t_f \\ t_0^3 & t_f^3 & 3t_0^2 & 3t_f^2 \end{bmatrix} = \begin{bmatrix} f(t_0) & f(t_f) & f(\dot{t}_0) & f(\dot{t}_f) \end{bmatrix} \tag{2.6a}$$

$$or, \qquad WB = A \tag{2.6b}$$

The above system of linear equations becomes over-constrained as the number of constraints on the trajectory increases, necessitating an increase in the degree of the polynomial. Another approach to the over-constrained problem would be to sample goal points along the trajectory and plan piecewise trajectories leading up to the final destination state.

2.2.1.2 Bernstein Polynomials

The trajectory can be represented as a linear combination of Bernstein basis polynomials [14] coming from the Bezier curve. The i-th Bernstein basis polynomial of order n is a real-valued function valued on $t \in [0, 1]$ as follows:

$$\boldsymbol{B_{i,n}(t)} = \binom{n}{i} t^i (1-t)^{n-i}, \qquad i = 0, 1, ...n \tag{2.7}$$

Fig. 2.1 depicts Bernstein basis polynomials of order-4.

The trajectory represented as a Bernstein polynomial can be depicted in terms of the basis polynomials and Bernstein coefficients W_i as follows:

$$f(t) = \sum_{i=0}^{n} W_i \boldsymbol{B_{i,n}(t)} \tag{2.8}$$

The derivatives of the Bernstein basis function are, in themselves, Bernstein polynomials[14]. Similar to eqn. 2.5, we can equate the function f at discrete timestamps to obtain the Bernstein coefficients W_i by solving a system of linear equations.

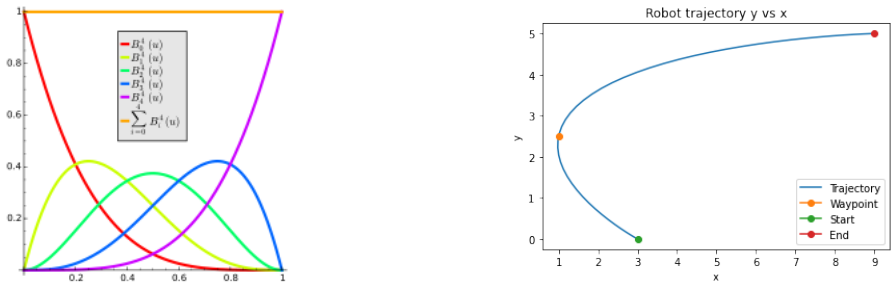


Figure 2.1: (a) Bernstein basis curves of order-4, (b) An example of a holonomic robot trajectory fitted using order-4 Bernstein polynomials.

2.2.2 Discrete-time representation

A discrete-time trajectory represented as a sequence of states $f(t) = \{f(t_0), f(t_1), ...f(t_f)\}$ corresponding to successive timestamps $t = \{t_0, t_1, ...t_f\}$. It can be thought of as sampling a continuous-time trajectory function at predefined timestamps. For the purpose of computational representation of trajectories, discrete-time sequences are generally used. To represent a trajectory in continuous time, generally, the basis functions are predetermined and only the coefficients are saved to reconstruct the trajectory as a weighted linear combination of known basis functions, similar to eqns. 2.5 and 2.8.

2.3 The Trajectory Optimization Problem

Linear Programming: Solving polynomial coefficients for the boundary state constraints as discussed in section 2.2 falls into a class of mathematical optimization known as Linear Programming(LP), which, simply put, involves solving a system of linear equations. Eqn. 2.6b with the polynomial weight vector as W, the basis polynomial matrix as B and the right-hand side boundary vector A can be solved using LP as follows:

$$W = inv(B)A \tag{2.9}$$

In addition to solving for boundary constraints, we can define a general objective function(ζ) in terms of the trajectory function f that should be subject to the robot's kinematic, dynamic, and environmental equality and inequality constraints g. The evaluation of this objective function comes from the motion planning module of the robot system. Mathematically, this optimization problem may be written as:

$$\min_{f(t)} \zeta(f(t)) \qquad \text{st: } g(f(t)) \leq 0 \tag{2.10}$$

Quadratic Programming: For most path planning objectives such as goal-reaching, jerk minimization, and smooth trajectory generation, the cost function ζ is quadratic with respect to f, giving rise to constrained Quadratic Program(QP) optimization, and can be written as follows:

$$\min_{f} \frac{1}{2} f^T \boldsymbol{P} f + q^T f \qquad \text{st: } Gf \leq h \tag{2.11}$$

For example, the objective function could be the Euclidean distance of the current position of a robot from the intended goal position, and the constraint matrices G and h could denote the boundary states and velocity limits for the robot. The solution to the above QP optimization problem presents challenges, including the non-convexity of the objective function and its constraints and discontinuities in the objective function. We will now discuss two approaches to solving QP optimization.

2.3.1 Gradient-based Optimization

The QP optimization proposed in eqn. 2.11 for $\zeta(f) = \frac{1}{2} f^T P f + q^T f$ can be solved by finding the inflection point in the convex objective curve. To do this, we compute the gradient of the objective function with respect to the optimization variables as follows:

$$\nabla_f \zeta = f^T P + q^T = 0 \tag{2.12}$$

Now, we attempt to attain the minima by updating the optimization variable with an iterative Gradient Descent approach. The update rule for f in the k^{th} optimization iteration for a learning rate of $\alpha > 0$ is as shown below:

$$f_{k+1}^T = f_k^T - \alpha \nabla_f \zeta \tag{2.13}$$

The feasibility of solving the optimization problem with this approach is dependent on factors such as the convexity of the objective function, the convexity of the constraints, and the computational complexity of computing the gradient. A few existing approaches for mobile robot path planning that use gradient-based optimization include [2], and [15]. Even for manipulator planning, [9], [16] use gradient-based optimization where the gradient of the objective is computed over the joint angle space.

2.3.2 Sampling-based Optimization

Sampling-based approaches bypass the gradient computation in eqn.2.12 by sampling points in the solution space and picking the sample that minimizes the optimization objective ζ. This approach is adopted in Rapidly-Exploring Random Trees(RRT) [17], where around each current point, random uniform samples of points are considered and connected to an expanding tree toward the goal. Similarly, in A* [18], Djikstra, and EBS-A* [19] algorithms, a heuristic function is optimized over an expanding graph search to compute the most optimal path from a start pose to an end pose.

Recent approaches [11], [12], and [**?**] sample candidate solutions to the optimization problem from a distribution and compute the cost associated with each candidate. Then the sample distribution is adapted to produce a "better" sample in the next iterations. This translates a deterministic optimization problem into a stochastic optimization problem. The benefit of this approach is its speed and generalizability to

non-convex, discontinuous objective functions that would be infeasible to solve using gradient-based optimization techniques.

2.3.3 Collision Avoidance methods

Collision avoidance for a robot involves the interaction of the robot with its environment and ensuring the following:

1. Collision avoidance with static obstacles (e.g, tree on a road, brick wall, etc.)
2. Collision avoidance with dynamic obstacles (e.g, other moving robots, pedestrians (for cars), birds (for drones) etc.)
3. Self-collision avoidance in case of an extended robot, such as a manipulator, between its own links and joints.

Collision detection is usually performed by sensors onboard the robot in simulation or hardware, but for the purpose of path planning, we mathematically estimate collisions using geometric approximations of obstacles and the robot. We assume the map of the environment is known to the robot from the Localization and Mapping modules. Most existing research, such as [3], approximate obstacles of varied shapes by their circumscribing spheres in 3-D or circles in 2-D. The robot itself is reduced to a point object, while its circumscribing radius is added as padding to the obstacles in the environment to account for grazing scenarios to simplify the path-planning process and collision detection. [2] estimates obstacles with axis-aligned ellipsoids with padding to generalize to a wider range of obstacles with non-uniform dimensions.

2.3.3.1 Distance-based Collision Avoidance

For Mobile Robots: For a 2-D mobile robot centered at (x_r, y_r), let us consider a circumscribing circle of radius R_{bot}. We aim to avoid collision with an obstacle centered at (x_o, y_o) of a circumscribing radius of R_{obs}. To avoid collision of the robot with the obstacle, we need to ensure the Euclidean distance between their centers is greater than or equal to the sum of their radii (the equality holding in the situation where the robot just grazes along the obstacle). Mathematically this can be represented as follows:

$$\sqrt{(x_r - x_o)^2 + (y_r - y_o)^2} \geq R_{obs} + R_{bot} \tag{2.14}$$

For unequal robot dimensions along the axes, we can estimate the robots with their circumscribing ellipsoids instead of spheres in 3-D. Fig 2.2 depicts the axis-aligned ellipsoidal robot representation and the distance between them that we use in Chapter 3 for our multi-agent optimizer.

For Manipulators: The same approach can be extended to robots moving in higher dimensional spaces. For complex robot systems such as manipulators, distance-based collision avoidance involves

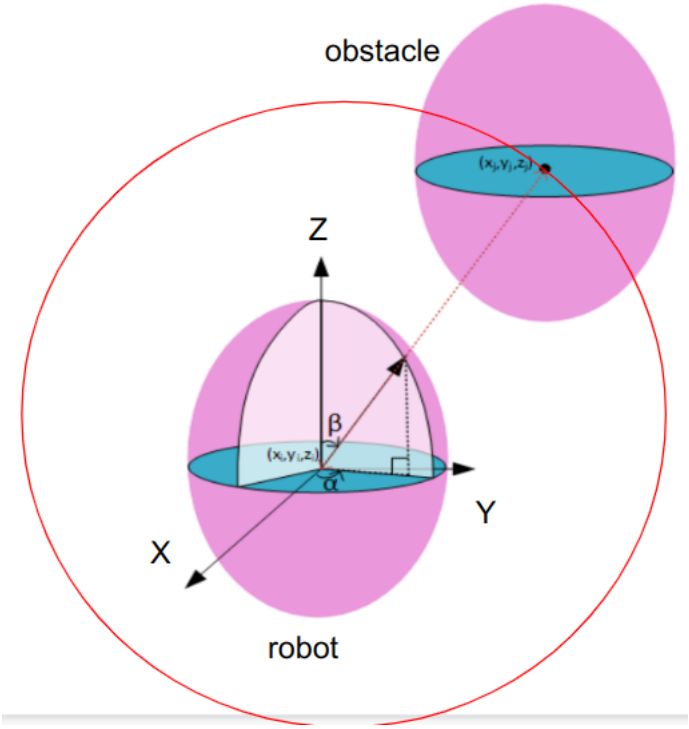


Figure 2.2: Axis-aligned ellipses representing robots for collision avoidance in chapter 3.

self-collision among the robot links, as discussed in point 3, along with collision avoidance with the environment. Based on our literature survey, there are broadly three approaches to solving this problem:

1. *Distance of sampled points:* [16] approximates the robot body B as a set of overlapping spheres $b \in B$ of radius r_l. We require all points in each sphere to be a distance, at least ϵ away from the closest obstacle. This constraint can be simplified as the center of the sphere being at least $\epsilon + r_l$ away from obstacles. Thus, our obstacle cost function is as follows:

$$q_o(\theta_t) = \sum_{b \in B} max(\epsilon + r_l - d(x_b), 0)||\dot{x_b}|| \tag{2.15}$$

2. *Implicit Representation Learning-based Collision Avoidance:* [11] uses *jointNERF*, a query-able neural network that learns implicit representations of the objects in the environment, including robot links and obstacles, and computes distances between them to detect collisions.

This dissertation addresses collisions for manipulators by involving mesh overlaps for collision avoidance in manipulators using the PyBullet simulator.

2.3.3.2 Time-scaling

A change in the independent variable from t to τ in the trajectory definition $f(t)$ does not change the path taken by the robot, but brings the following changes in the velocity profile of the trajectory:

$$\dot{f(\tau)} = \dot{f(t)}\frac{dt}{d\tau} \tag{2.16}$$

The trajectory $f(\tau)$ can be thought of as the robot passing through the same points as $f(t)$, with a scaled version of the original velocity $\dot{f(t)}$, i.e. changing the velocity and acceleration of the robot while keeping its position undisturbed. This transformation in the time scale allows us to regulate the velocity of the robot, playing a key role in collision avoidance. The robot sticks to its original planned global path with a scaled version of its original velocity, and regains its original velocity once the collision with a dynamic obstacle has been avoided. This approach proves to be agnostic to the kinematics of the robot itself since it does not hold any assumptions about the initial trajectory of the robot.

As for the transformation between t and τ, [15] experiments with constant as well as exponential time-scaling, estimating the parameters of the transformation function and using them for collision avoidance against workspaces cluttered with dynamic obstacles.

2.3.4 Performance Metrics

This subsection discusses a few performance metrics that are used to evaluate trajectories obtained from different planning methods. We use the following metrics for bench-marking one algorithm against another:

- *Smoothness cost:* It is computed as the norm of the second-order finite-difference of the robot position at different time instances. We aim to minimize this smoothness cost to get smooth trajectories feasible for execution through a controller.

- *Arc-length:* It is computed as the norm of the first-order finite-difference of the robot positions at different time instances. We aim to minimize arc-length for computed trajectories to ensure short trajectory curves are returned.

- *Computation Time:* The time taken for each approach to return a smooth and collision-free solution. Depending on the application, we may want computation time in the order of milliseconds for real-time online adaptive planning or in the order of a few seconds for offline planning.

- *Success Rate:* It is computed as the number of successful completion of the intended task (such as, reaching a goal, picking up a payload) as a fraction of the total number of tries. An algorithm with a higher success rate will be preferred for the task at hand.

- *Number of Collisions:* It is calculated as the number of times the robot collides with static and dynamic obstacles, as well as with itself while following a planned trajectory. Lower the number of collisions, the more effective our planning method.

2.4 Multi-Robot Path Planning

This section provides a brief overview of the multi-robot path planning problem, including a primer for some of the ideas that shape our algorithm discussed in Chapter 3.

2.4.1 Applications

Multi-robot systems consist of several interaction-aware robots, each capable of independent decision-making to execute a collective task. The inspiration behind such systems is derived from swarms of bees and other insects. Typical examples of such systems used widely include swarm drones and robot fleets. Multi-robot systems involve coordination and communication among constituent robots, allowing a division of labor to achieve a given task. For example, robot fleets are being extensively used today to perform search-and-rescue missions in areas unsuitable for human intervention, particularly during natural disasters. Swarm drones are also used for imaging and mapping vast territories, particularly for military applications to detect enemy advances. They are also used for demonstrating geometric formations, as well as for military combat, last-mile parcel delivery, and precision agriculture.

Figure 2.3: Swarm drones are used for a wide variety of robotics applications, ranging from search-and-rescue missions, military combat, parcel delivery, and precision agriculture. Source: `https://www.economist.com/the-economist-explains/2015/10/02/how-swarm-drones-are-mimicking-nature`

2.4.2 Challenges Involved

A multi-robot system must ensure that individual robots do not crash into their surroundings or each other. The individual robots must also follow short and safe trajectories from start to goal, and replan their trajectories in near real-time if required. A few challenges associated with multi-robot path planning are as follows:

1. *Coordination and Communication Issues:* Individual robots may communicate directly with each other or relay through a central ground station to share information and coordinate their actions. However, communication and bandwidth limitations can cause delays and disruptions in data transmission.

2. *Scalability Issues:* Multi-robot systems need to perceive their environment accurately to avoid collisions and perform their tasks effectively. However, the limited sensing capabilities of robots and the complexity of the environment can make it difficult to obtain reliable information about the surroundings. Moreover, as the number of drones in a swarm increases, the overhead for collision avoidance also increases since collision constraints need to be computed pairwise.

Overall, multi-robot planning and coordination are complex and challenging tasks that require careful consideration of communication, sensing, control, and optimization. Despite these challenges, the potential benefits of swarm drones make them an attractive area of research and development for many applications. To solve problem 1, multi-robot systems typically use decentralized control, meaning that each robot makes decisions for itself without central coordination. We have already discussed Centralized versus Distributed trajectory optimization paradigms in Section 1.2.1, which can solve the scalability issues in problem 2, rising from an increasing number of robots in a multi-robot system. However, the accuracy of distributed multi-robot planning depends on the accuracy of the predictive model each robot uses for the other robots. The algorithms we discuss in the following subsections can be implemented either in a centralized or a distributed setting, depending on the speed, scale and collision avoidance requirements.

2.4.3 Graph-Search-based Multi-Agent Path Finding

Multi-Agent Path Finding(MAPF) consists of a class of algorithms under the umbrella of multi-robot planning that involves the computation of collision-free paths from the start to the goal in a shared environment. Typical solutions for the multi-agent path-finding problem derive inspiration from shortest path computation in graph theory by discretizing the action space of the robots into graph nodes and constructing edges between them based on permissible controls. Let us now discuss some of the different graph-search-based algorithms that have been developed for MAPF:

2.4.3.1 A* Algorithm:

A*[18] is a popular heuristic search algorithm that can be used for MAPF. It uses an admissible heuristic function to estimate the cost of reaching the goal state, typically based on the distance between agents and their targets. A* algorithm can find optimal solutions but can be slow for large-scale MAPF problems.

2.4.3.2 Conflict-based Search(CBS) Algorithm:

CBS[20] is a popular algorithm for solving MAPF problems. CBS is a two-level search algorithm that first searches for individual agent paths and then resolves conflicts between them. CBS can handle a large number of agents and can find optimal solutions for MAPF problems.

2.4.3.3 Push and Swap (PaS):

PaS[21] is a local search algorithm that operates by pushing agents and swapping their positions to find a collision-free solution. PaS algorithm is efficient for small-scale MAPF problems but can struggle to find optimal solutions for large-scale problems.

2.4.4 Batch Trajectory Optimization

Batch trajectory optimization can be applied to multi-robot systems, where a group of robots works together to perform a task. In this context, the goal is to optimize the trajectories of all robots in the system to achieve the desired task while minimizing the overall cost of the system. Instead of considering all the robots in the system at once, they could be split into groups or batches that solve the trajectory optimization problem independent of one another as well. A set of candidate trajectories are chosen for each robot based on the robot dynamics and task definition and are optimized for all the robots in a single shot by minimizing the overall cost function for all the robots in the system. The cost function could the the norm of the acceleration of all the robots to ensure precise, smooth movements, or the total time taken by all the robots to execute the given task. Note that this optimization problem also includes collision avoidance constraints between pairs of agents and obstacles in the environment. Then the optimal set of trajectories are executed by all the robots.

2.4.5 Accelerating Batch Optimization over GPUs

Matrix computations can be sped up over a GPU (Graphics Processing Unit) by taking advantage of the massively parallel architecture of the GPU. A GPU typically contains thousands of small processing cores that work together to perform computations in parallel. Matrix computations, such as matrix multiplication, are particularly well-suited for parallel processing because each element of the output matrix can be computed independently.

To speed up matrix computations on a GPU, the matrix data is first transferred from the CPU memory to the GPU memory. This transfer can be done using specialized libraries such as CUDA, which is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs. Once the matrix data is in the GPU memory, the computation is split into many smaller computations that can be performed in parallel across the thousands of cores on the GPU. For example, in matrix multiplication, each output element can be computed by multiplying the corresponding row of the first matrix by the corresponding column of the second matrix. By breaking the computation down into smaller tasks that can be performed in parallel, the GPU can perform matrix computations much faster than a CPU, which typically has fewer processing cores.

We use the JAX library [22], developed by Google in Chapter 3 to accelerate matrix operations. Additional details and usage of the JAX NumPy library has been discussed in Appendix A.1

2.5 Manipulator Path Planning

This section provides a brief overview of the kinematics of robot manipulators and introduces the task of path planning for manipulators, as a background for our approach discussed in Chapter 4.

The problem of robot manipulator path planning is the task of finding a collision-free path for a robotic arm to move from its initial position to a desired final position while avoiding obstacles in its workspace. This is a challenging problem because the workspace of the robot can be complex and high-dimensional, and the motion of the robot can be constrained by its physical limits, such as joint velocity and angle limits. Path planning algorithms must take into account these constraints and find an optimal path that satisfies the constraints while minimizing some performance criteria, such as the path length or the time required to complete the motion. This problem is relevant in many applications of robotics, including industrial automation, manufacturing, and healthcare, where robots are used to perform repetitive or complex tasks in a safe and efficient manner. An example of path planning required for a robot manipulator is illustrated in Figure 2.4.

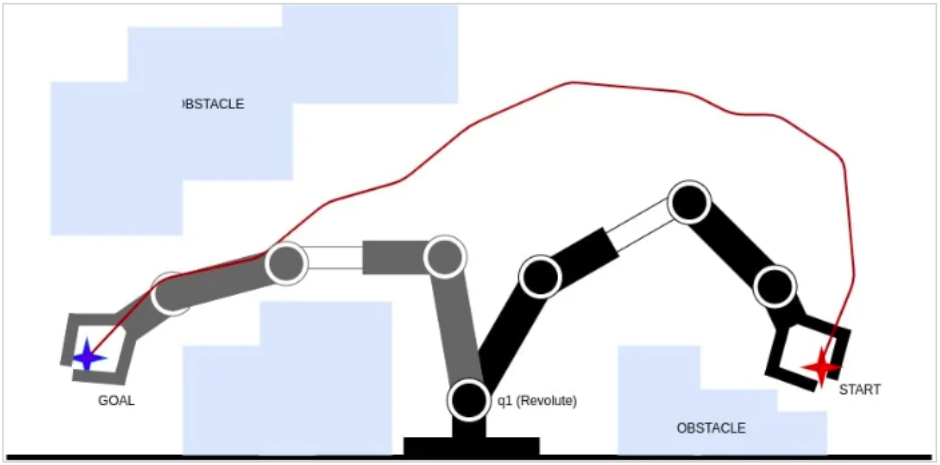


Figure 2.4: An illustration showing the task of path planning for a robot manipulator from a start pose to an end pose, while avoiding obstacles in the surroundings and preventing self-collisions and coiling in on itself. The red line denotes the trajectory to be followed by the end-effector from start to goal. Source: `https://control.com/technical-articles/how-does-motion-planning-for-autonomous-robot-manipulation-work/`

2.5.1 Manipulator Kinematics

In the context of this thesis, we discuss the manipulator planning problem primarily for two robot manipulators - Franka Emika Panda robot arm and Universal Robots(UR5e) robot arm. The number of degrees of freedom(DoF) for the former is 7, while that for the latter is 6. Figure 2.5 shows the Franka

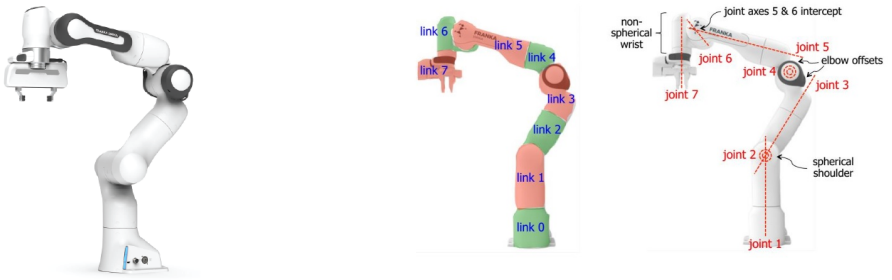


Figure 2.5: (a) The Franka Emika Panda Robot Arm with an end-effector(two-finger gripper)(b) The links and types of joints and their locations for the 7-DoF Franka Panda manipulator.

Panda robot manipulator against its links and joints, while Figure 2.7 denotes the same for the UR5e robot arm.

The kinematics of a robot manipulator describes the relationship between the joint angles and the position and orientation of the end-effector of the robot. Forward kinematics denotes the transformation of joint angles to an end-effector position, while inverse kinematics attempts to calculate the required joint angles to achieve a desired end-effector position. We will now discuss the forward kinematics of both manipulators mathematically.

2.5.1.1 Franka Emika Panda

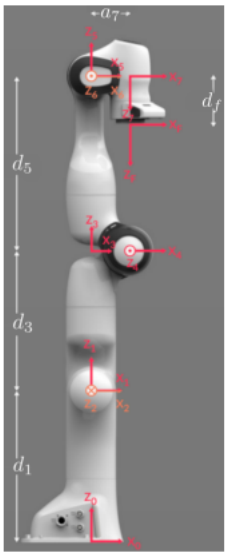
The Franka Emika Panda is a 7 degree-of-freedom robot manipulator that is widely used in research and industrial applications. The Panda robot has a serial kinematic structure, where each joint is connected to the previous joint and to the base of the robot. The joint angles, corresponding to the 7 joints, shown in Figure 2.5 are denoted by q_1, q_2, ..., q_7, corresponding to the rotations of each joint around its respective axis. The position and orientation of the end-effector are represented by a 4x4 transformation matrix T. The forward kinematics of the Panda robot can be described by the following equations:

$$T = T_1 T_2 T_3 T_4 T_5 T_6 T_7 \tag{2.17}$$

where T_i is the transformation matrix corresponding to the i^{th} joint of the robot. These transformation matrices can be computed from the Denavit-Hartenberg parameters, which describe the geometry and kinematics of the robot as per the following equation:

$$T_i = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.18}$$

The Denavit-Hartenberg frames and parameters for the Franka Panda robot arm are depicted in Figure 2.6.



i	a_i	α_i	d_i	θ_i
1	0	0	d_1	q_1
2	0	$-\pi/2$	0	q_2
3	0	$\pi/2$	d_3	q_3
4	a_4	$\pi/2$	0	q_4
5	a_5	$-\pi/2$	d_5	q_5
6	0	$\pi/2$	0	q_6
7	a_7	$\pi/2$	0	q_7
8	0	0	d_f	0

Figure 2.6: Denavit-Hartenberg(D-H) frames and parameters for the 7-DoF Franka Panda robot arm. Source: [23].

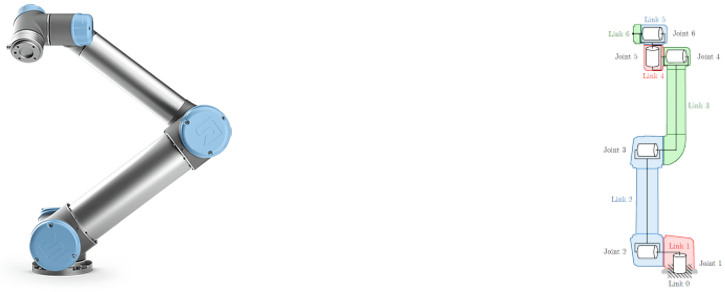
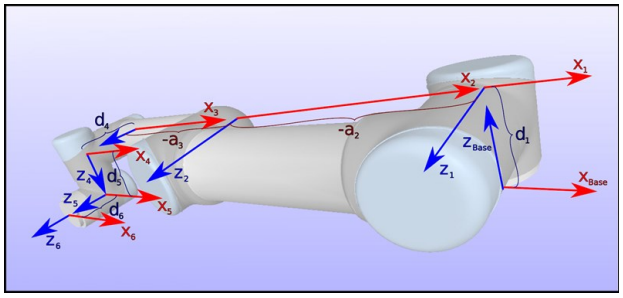


Figure 2.7: (a) The Universal Robots(UR5e) Robot Arm (b) The links and types of joints and their locations for the 6-DoF UR5e manipulator.

2.5.1.2 UR5e

The Universal Robots UR5e is a 6-degree-of-freedom robot arm. The joint angles, corresponding to the 6 joints, shown in Figure 2.7 are denoted by q_1, q_2, ..., q_6, and the overall forward kinematics is described by the serial multiplication of six transformation matrices corresponding to the six joints, similar to eqn.2.17. The Denavit-Hartenberg(D-H) frames and parameters for the 6-DoF UR5e robot arm are depicted in Figure 2.8.



UR5e				
Kinematics	theta [rad]	a [m]	d [m]	alpha [rad]
Joint 1	0	0	0.1625	π/2
Joint 2	0	-0.425	0	0
Joint 3	0	-0.3922	0	0
Joint 4	0	0	0.1333	π/2
Joint 5	0	0	0.0997	-π/2
Joint 6	0	0	0.0996	0

Figure 2.8: Denavit-Hartenberg(D-H) frames and parameters for the 6-DoF UR5e robot arm. Source: `https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics`

2.5.2 Challenges Involved

Path planning for robot manipulators is a complex problem that involves several challenges. Here are some of the key challenges involved:

- *High Dimensionality:* Robot manipulators typically operate in high-dimensional spaces, with each joint providing a degree of freedom, such as the 7-dimensional joint angle space in the case of the Franka Panda arm and the 6-dimensional joint space for the UR5e arm. This makes the search space for path-planning algorithms very large, and finding an optimal path becomes computationally expensive.
- *Nonlinear Constraints:* The motion of robot manipulators is subject to nonlinear constraints, such as the dynamics of the robot and the workspace bounds, as well as physical constraints, such as joint limits and collision avoidance. Incorporating these constraints into path-planning algorithms can be challenging, and some algorithms may require sophisticated mathematical techniques to solve the constraints.
- *Real-time Operation:* In many rearrangement applications, path planning must be performed in real-time, with the manipulator making decisions and adapting to new obstacles or changes in the environment. Real-time path planning requires efficient algorithms that can quickly search the space of possible paths and adapt the plan as needed.

Overcoming the above challenges requires the development of efficient path-planning algorithms that can handle these constraints and produce high-quality paths. We attempt to address all these challenges through our bi-level trajectory optimizer for manipulators discussed in Chapter 4.

2.5.3 Joint space-vs end-effector space

For a robot manipulator, the path-planning can be performed in two motion spaces - either in the high-dimensional space formed by the joint angles corresponding to each movable joint or the end-effector

space, which is a 3-dimensional space limited by the workspace bounds or a 2-dimensional table plane (for planar end-effector motion). The choice of motion space is determined by the requirements of the task at hand. For example, for a robot arm pushing objects on the table, the major task at hand involves the pose of the end-effector on the 2-D tabletop plane, and hence the path planning can be performed in the Cartesian end-effector space or even in the Cartesian space of the center of the object being pushed on the table. For tasks involving grasping objects, it is more effective to plan trajectories in the joint space in order to avoid collisions and minimize joint movements.

Based on the two paradigms of optimization discussed in Sections 2.3.1 and 2.3.2, let us discuss both Gradient-based and Sampling-based trajectory optimization for a robot manipulator.

2.5.4 Gradient-based Optimization

2.5.4.1 CHOMP

CHOMP (Covariant Hamiltonian Optimization for Motion Planning)[9] is a gradient-based optimization method for efficient motion planning of robot manipulators in high-dimensional spaces. This method was developed by Brian Paden and Emo Todorov in 2011.

The key idea behind CHOMP is to formulate the motion planning problem as an optimization problem, where the objective is to minimize an objective function that typically includes terms that penalize collisions with obstacles, deviation from a desired trajectory, and energy consumption. In CHOMP, the optimization is performed using a gradient descent method, where the gradient of the cost function with respect to the joint angles of the robot is computed and used to update the joint angles iteratively. The gradient is computed using backpropagation through time, which involves propagating the gradient backward through a sequence of time steps.

One of the key advantages of CHOMP is its ability to handle non-convex constraints and to generate smooth, continuous paths that avoid collisions with obstacles. This is achieved by using a smoothing term in the cost function, which encourages the joint angles to vary smoothly over time. Another advantage of CHOMP is its scalability to high-dimensional spaces, which makes it well-suited for motion planning of robot manipulators with many degrees of freedom.

However, CHOMP does have some limitations. It requires a good initial guess of the path, which can be challenging in complex environments with many obstacles. The optimization can also converge to local minima, which can result in suboptimal paths. Further, CHOMP involves gradient computations which may get intractable to compute in very high-dimensional spaces and spend a significant amount of time to return a feasible trajectory, eliminating the scope of real-time path planning. To address these limitations, various extensions and modifications to CHOMP have been proposed, such as incorporating stochastic optimization or combining CHOMP with other planning algorithms.

2.5.4.2 TrajOpt

TrajOpt[10], developed by John Schulman and his colleagues at UC Berkeley in 2013, is an open-source library for motion planning and trajectory optimization of robot manipulators.

TrajOpt also involves gradient-based trajectory optimization using techniques such as sequential quadratic programming or gradient descent. The TrajOpt library provides a flexible and modular framework for specifying the motion planning problem, allowing users to define a wide range of cost functions and constraints, including collision avoidance, joint limits, end-effector constraints, and task-specific objectives such as minimizing energy consumption. TrajOpt also uses a collision-checking algorithm based on bounding volume hierarchies, which can quickly identify potential collisions between the robot and the environment. It also incorporates a warm-starting strategy, which initializes the optimization algorithm with a feasible solution from a previous iteration, reducing the time required for convergence.

TrajOpt, written in C++ and Python, is designed to be highly customizable and extensible. It can be integrated with various robot simulation and control frameworks, such as ROS, Gazebo, or PyBullet, and supports a wide range of environments, robot models, and kinematic solvers. [24] uses TrajOpt as a collision-detection module.

2.5.5 Sampling-based Optimization

Sampling-based optimizers rely on stochastic optimization, i.e. sampling a set of potential candidate trajectories to be followed by the manipulator and evaluating the associated costs to finally choose the trajectory with the minimum cost. The best set of candidates is used to iteratively improve on the samples for the next iteration, ultimately helping the optimizer converge to an optimum in the cost function.

2.5.5.1 STORM

STORM(Stochastic Tensor Optimization for Robot Motion)[11] is an optimization algorithm designed to efficiently and effectively generate smooth and collision-free trajectories for robots in complex environments between a start state and a desired state by iteratively optimizing a tensor-based cost function. STORM works by first defining a tensor-based representation of the state space, designed to capture the high-dimensional motion space of the robot. The tensor is defined as a multi-dimensional array, where each dimension corresponds to a specific aspect of the robot's state, such as position, velocity, or acceleration. The size of each dimension is determined by the resolution of the state space.

The optimization process is stochastic, which means that it involves randomly sampling different trajectories and evaluating their costs. Self-collisions are detected by using *jointNERF*, a network that computes the closest distance between robot links given joint poses, while environment collisions are based on signed distances. This randomness helps to ensure that STORM is able to find a diverse set of trajectories that can be used to navigate complex environments. Additionally, STORM is able to leverage parallel computing to speed up the optimization process, which can be critical for real-time applications.

One of the key advantages of STORM is its ability to handle high-dimensional state spaces, which is particularly useful for manipulator path planning. It achieves this by using a tensor-based representation of the state space, which allows it to efficiently explore large and complex environments. STORM is also able to incorporate a wide range of constraints, including collision avoidance, joint limits, and kinematic constraints.

2.5.5.2 Cross-Entropy Methods(CEM)

The cross-entropy method (CEM)[25] is a derivative-free optimization technique that employs an adaptive importance sampling procedure using the cross-entropy measure. In each iteration of the CEM-based stochastic trajectory optimization, the following two steps are performed:

- Derive candidate trajectories by sampling from a probability distribution
- Minimize the cross-entropy between the sample distribution and a target distribution to update the parameters of the former.

Mathematically, these steps can be represented as follows (Source: `https://en.wikipedia.org/wiki/Cross-entropy_method`):

1. Choose initial parameter vector $\boldsymbol{v}^0$; set t = 1
2. Generate a random sample $\boldsymbol{X_1}$, ...,$\boldsymbol{X_N}$ from $f(\boldsymbol{v^{(t-1)}})$.
3. Solve for $\boldsymbol{v^{(t)}}$, for a given cost function ζ where,

$$\boldsymbol{v^{(t)}} = \underset{v}{\operatorname{argmax}} \frac{1}{N} \sum_{i=1}^{N} \zeta(\boldsymbol{X_i}) \frac{f(\boldsymbol{X_i}; u)}{f(\boldsymbol{X_i}, \boldsymbol{v^{(t-1)}})} \log f(\boldsymbol{X_i}; \boldsymbol{v}) \tag{2.19}$$

4. If convergence is reached, then stop, otherwise proceed to the next iteration by incrementing t by 1 and repeat from step 2.

The cost function ζ takes into account factors such as the length of the path, the amount of time it takes to complete the path, and the distance from the robot arm to obstacles in the environment. Over time, the CEM algorithm converges to an optimal solution, which represents the best path for the robot arm to reach the target point while avoiding obstacles.

We use an improved version of CEM, called Via-Point Stochastic Trajectory Optimization(VP-STO)[13] for manipulator path planning in Chapter 4.

Chapter 3

Fast Joint Multi-Robot Trajectory Optimization by GPU Accelerated Batch Solution of Distributed Sub-Problems

In this chapter, we design a novel approach to solve our research problem **T2** from Section 1.1.1 - a joint multi-robot trajectory optimizer that can compute trajectories for tens of robots within a small fraction of a second. The computational efficiency of our approach is built on breaking the per-iteration computation of the joint optimization into smaller, decoupled sub-problems and solving them in parallel through a custom batch optimizer. By achieving near real-time planning through a distributed optimization paradigm, we are able to address challenges involving both coordination and scalability in a multi-robot system, discussed in Section 2.4.2.

(Published in the journal Frontiers in Robotics and AI, Vol. 9, 2022. [26], also presented at the DMMAS workshop, International Conference on Intelligent Robots and Systems (IROS) 2022.)

3.1 Introduction

Deployment of multiple aerial vehicles such as quadrotors is critical for applications like search and rescue and exploration and mapping of large areas [27]. Over the last decade, robot fleets have also become ubiquitous in applications like ware-house automation that have a substantial economic impact on society [28, 29]. Furthermore, with the advent of connected autonomous cars, it becomes imperative also to view urban mobility as a multi-robot system [30]. A fundamental component of any multi-robot system is the coordination planning that guides individual robots between their start and goal locations while avoiding collisions with the environment and other robots. In this chapter, we adopt the optimization perspective for multi-robot motion planning [2]. In this context, the existing approaches broadly fall into two spectra. On one end, we have the centralized approaches wherein the trajectory of all the robots are computed together. The centralized approach can be further subdivided into sequential [7], [1] and joint optimization [31], [2] respectively depending on whether the trajectories of the robots are computed one at a time or simultaneously. On the other end of the spectrum, we have online distributed model predictive control (DMPC) [8], [32] based approaches wherein each individual

robot computes its trajectories in a decoupled manner based on the trajectory prediction of the other robots in the environment. In some works, the prediction module is replaced by robots communicating their current trajectory with each other [33].

Centralized approaches, especially the joint optimization variants, provide a rigorous treatment of the collision avoidance constraints and access a larger feasible space formed by all trajectory variables of all the robots. However, joint optimization quickly becomes intractable as the number of robots increases [6]. In contrast, the distributed MPC approaches can run in real-time but can lead to oscillatory behaviors, and consequently, low success rates of collision avoidance [8], [33]. This is because the trajectories computed at each control cycle by any robot are only collision-free with respect to the predicted (or prior communicated) trajectories of other robots and not the actual trajectories followed by them.

Our main motivation in this chapter is to improve the computational tractability of multi-robot trajectory planning using a distributed optimization approach to the extent that it becomes possible to compute trajectories for tens of robots in densely cluttered environments in a few tens of milliseconds. To put in context, the said timing is several orders of magnitude faster than some of the existing approaches for joint multi-robot trajectory optimization [31], [34]. Such improvements in computation time would ensure the applicability of our approach for even online re-planning besides the standard use case of computing offline global trajectories for the robots. For example, consider a scenario wherein each robot uses local real-time planners such as Dynamic Window Approach [35] or DMPC [8] to avoid collisions with other robots in a distributive manner. Our approach could provide global re-planning for the local planners at more 5Hz. or more.

On the application side, our main focus is on coordination of multiple quadrotors, typically for applications like search and rescue and coordinated exploration. These applications require point-to-point, collision-free navigation and forms the main benchmark in our experiments. However, our algorithm can be useful for coordination of multiple wheeled mobile robots and even autonomous cars.

Contributions: The computational efficiency of our approach is built on several layers of reformulation of the underlying numerical aspects of the joint multi-robot trajectory optimization. We summarize the key points and the benefits that it leads to below.

Algorithmic: Our main idea is to break the per-iteration computation of the joint multi-robot trajectory optimization into smaller, distributed sub-problems by leveraging the solution computed in the previous iterations. Although similar ideas have appeared in many existing works [34], [33], a core challenge remains: how to efficiently solve the decoupled problem arising at each iteration in parallel. The basic assumption is that the decoupled optimizations can be parallelized across separate CPU threads [34]. However, our recent works have shown that such a parallelization approach does not scale well with an increase in the number of problems [36]. The inherent limitation stems from the available CPU cores and thread synchronization issues.

Thus our main algorithmic contribution in this chapter lies in deriving a novel optimizer that can be efficiently run in a batch setting. In other words, our optimizer can take a set of decoupled optimization problems and vectorize the underlying numerical computations across multiple problem instances.

Consider an optimizer that solves a given problem by adding two vectors as a hypothetical example. We can trivially vectorize the computation over different problem instances by stacking each problem's vectors together in the form of a matrix and adding them together. Moreover, this matrix addition can be easily parallelized over GPUs for many problem instances. Our proposed optimizer achieves similar vectorization but for a set of difficult non-convex sub-problems, resulting in each iteration of joint multi-robot trajectory optimization. Specifically, we show that solving the decoupled sub-problems predominantly reduces to solving novel equality constrained quadratic programming (QP) problems under certain collision constraint reformulations. The novelty of the QPs stems from the fact that they all share the same matrices (e.g., Hessian), and only the vectors associated with the QPs vary across the sub-problems. We show that solving all the QP sub-problems in one shot reduces to computing one large matrix-vector product that can be trivially parallelized over GPUs using off-the-shelf linear algebra solvers.

Applied: We release our entire implementation for review and to promote further research on this problem. We also release the benchmark data sets used in our simulations.

State-of-the-art Performance We compare our GPU accelerated optimizer with two strong baselines [1, 2] and show massive improvement in computation time while being competitive in trajectory quality as measured by metrics like arc-length and smoothness. Our first comparison is with [1] that uses a sequential approach for multi-robot trajectory optimization. Our computation time is at least 76.48% lower than that of [1] for a smaller problem size involving 16 robots. Moreover, the performance gap increases substantially in our favor as we increase the number of robots and make the environment more cluttered by introducing more static obstacles. We observe similar trends in trajectory arc-length and smoothness comparison between the two approaches. Our second comparison is with [2] that searches directly in the feasible joint space formed by all the pair-wise collision avoidance constraints. Our proposed optimizer shows improved scalability over [2] for a larger number of robots while being also superior in trajectory arc length and smoothness.

3.2 Problem Formulation and Related Work

This section introduces the general problem formulation for multi-robot trajectory optimization. We subsequently use the problem set-up to review existing works and contrast our optimizer with them. We begin by summarizing the main symbols and notations used throughout the chapter.

3.2.1 Symbols and Notations

In this chapter, the lower normal and bold letters denote the scalars and vectors, respectively, while the upper bold case variants represent matrices. The left and right super-scripts denoted by k and T will be used to denote the iteration index of the optimizer and transpose of the vectors and matrices. The time-stamp of any variable will be denoted by t. The symbol $\|.\|_2$ stands for l_2 norm. We summarize

some of the main symbols in Table (3.1) while some are also introduced in their first place of use. At some places, we perform a special construction where time-stamped variables are stacked to form a vector. For example, $\mathbf{x}_i$ will be formed by stacking $x_i(t)$ at different time instants.

3.2.2 Robot Kinematics

Our optimizer is designed for robots with holonomic motion models. That is, the motion along each axis is decoupled from each other. This is a common assumption made in quadrotor motion planning. Many commercially available wheeled mobile robots also have similar kinematic model. Under certain conditions, even motion planning for car-like vehicles also adopt similar kinematic model and thus our optimizer is suitable for those as well [37].

3.2.3 Trajectory Optimization

For holonomic robots modeled as series of integrators, the joint trajectory optimization can be formulated in the following manner.

$$\min_{x_i(t),y_i(t),z_i(t)} \sum_{t,i} \Big(\ddot{x}_i^2(t) + \ddot{y}_i^2(t) + \ddot{z}_i^2(t) \Big), \tag{3.1a}$$

$$\Big(x_i(t_0), \dot{x}_i(t_0), \ddot{x}_i(t_0), y_i(t_0), \dot{y}_i(t_0), \ddot{y}_i(t_0), z_i(t_0), \dot{z}_i(t_0), \ddot{z}_i(t_0) \Big) = \mathbf{b}_{o,i}, \forall i \tag{3.1b}$$

$$\Big(x_i(t_f), \dot{x}_i(t_f), \ddot{x}_i(t_f), y_i(t_f), \dot{y}_i(t_f), \ddot{y}_i(t_f), z_i(t_f), \dot{z}_i(t_f), \ddot{z}_i(t_f) \Big) = \mathbf{b}_{f,i}, \forall i \tag{3.1c}$$

$$- \begin{pmatrix} x_i(t) - x_j(t) \\ y_i(t) - y_j(t) \\ z_i(t) - z_j(t) \end{pmatrix}^T \mathbf{S} \begin{pmatrix} x_i(t) - x_j(t) \\ y_i(t) - y_j(t) \\ z_i(t) - z_j(t) \end{pmatrix} + 1 \leq 0, \forall t, \{i, j \in \{1, 2, ..., n_r\}, j \neq i\}, \mathbf{S} = \begin{bmatrix} a^2 & 0 & 0 \\ 0 & a^2 & 0 \\ 0 & 0 & b^2 \end{bmatrix} \tag{3.1d}$$

The cost function (3.1a) minimizes the squared norm of the acceleration at each time instant for all the robots. The equality constraints (3.1b)-(3.1c) enforces the initial and final boundary conditions on positions, velocity, and accelerations on each robot trajectory. The pair-wise collision avoidance constraints are modeled by inequalities (3.1d), wherein we have assumed that the robots are shaped as axis-aligned spheroids with axis dimensions (a, a, b). For the ease of exposition, we consider all robots to have the same shape. Extension to a more general setting is trivial. The constraints (3.1d) are typically enforced at pre-selected discrete time-stamps, and thus a fine resolution of discretization is necessary for accurately satisfying the constraints. For now, we do not consider any static obstacles in the environment

in the formulation above. The extension is trivial as static obstacles can be considered robots with zero velocity and whose trajectories are not updated within the optimizer's iteration.

Let the trajectory of each robot along each motion axis x, y, z be parameterized through n_v number of variables. For example, these variables could be time-stamped way-points representing the trajectory or the coefficients of their polynomial representation (see (3.8)). Then, for a set-up with n_r number of robots and a planning horizon of n_p, optimization (3.1a)-(3.1d) involves $n_r * n_v$ variables, and $18 * n_r$ equality constraints. The number of pair-wise collision constraints would be $\binom{n_r}{2} * n_p$.

The number of decision variables in optimization (3.1a)-(3.1d) scales linearly with the number of robots. Although this increase poses a computational challenge, the main difficulty in solving the optimization stems from the non-convex pair-wise collision avoidance constraints (3.1d) as the rest of the cost and constraint functions are convex. Moreover, the number of collision avoidance constraints increases exponentially with the number of robots. Existing works [31, 2, 7, 29, 1] have adopted different simplifications on the collision avoidance constraints to make multi-robot trajectory optimization more tractable. We thus next present a categorization of these works based on the exact methodology used.

3.2.4 Literature Review

3.2.4.1 Joint Optimization with Conservative Convex Approximation

The most conceptually simple approach is to solve (3.1a)-(3.1d) as one large optimization problem, wherein the trajectory of every robot is computed in one shot. Authors in [31] simplified the joint optimization by deriving a conservative affine approximation of the collision avoidance constraints (3.1d) and consequently reducing (3.1a)-(3.1d) to a sequence of QPs. As a result, their solution process becomes somewhat tractable for a moderate number of robots (≈ 10). However, the computation time of [31] scales poorly because the number of affine constraints still increases exponentially with the number of robots. Our prior work [2] substantially improved the scalability of joint multi-robot trajectory optimization by reformulating the Euclidean collision constraints (3.1d) into polar form and augmenting them into the cost function by using concepts from the Alternating Direction Method of Multipliers (ADMM). Moreover, we showed that such reduction allowed one-time offline caching of the most expensive parts of the computation. As a result, [2] achieved over two orders of magnitude speed-up over [31] for 16 robots. The current proposed work provides a further significant improvement over [2] in computation time and trajectory quality.

3.2.4.2 Sequential Optimization

Sequential planners plan for only one robot at a time. At any given planning cycle, the previously computed robot trajectories are considered dynamic obstacles for the currently planned robot. As a result, these approaches ensure that the number of decision variables does not increase with robots. Moreover, the number of collision avoidance constraints increases linearly as the planning cycle progresses. How-

ever, note that the linear increase in the number of constraints does not translate to similar scaling in computation time. Even state-of-the-art interior-point solvers have cubic complexity with respect to the number of constraints.

A critical disadvantage of sequential planners is that each subsequent robot has access to less feasible space to maneuver. As a result, optimization problems become progressively constrained as the planning cycle progresses, leading to potential infeasibility. Authors in [7] tackle this problem by developing an incremental constraint tightening approach. The authors integrate a subset of collision avoidance constraints into the optimization problem, and the size of this set is gradually increased based on the actual collision residuals.

Sequential planners naturally have the notion of priority, and these can be chosen carefully for improved performance. For example, [29] adopts a priority-based optimization method in which the robots are divided into groups/batches with pre-determined priorities, and the trajectory optimization problem is solved from the highest to the lowest priority group. Similar approach was adopted in [1]. Performing sequential planning over a small batch of robots reduces its conservativeness. On the other hand, it introduces an additional challenge of ensuring collision amongst the robots in a given batch. Authors in [1] tackle this bottleneck by leveraging graph-based Multi-robot Path Finding (MAPF) methods.

3.2.4.3 Distributed Optimization

Distributed optimizers at each iteration, break (3.1a)-(3.1d) into decoupled smaller problems. For example, see [34], [38]. The key insight upon which all existing works build is that the only coupling between different robots stem from the pair-wise collision constraints (3.1d) [38]. Thus, we if we discard this coupling, (3.1a)-(3.1d) can be easily reduced to n_r number of decoupled optimizations. One way to achieve the said decoupling is to let each robot make prediction of how the trajectories of other robots are going to look in the immediate next iterations and use that to simplify the collision avoidance constraints. More formally, let $(\overline{x}_j(t), \overline{y}_j(t), \overline{z}_j(t))$ be the predicted position of j^{th} robot at time t. Then, the collision avoidance constraints can be simplified as (3.2).

$$\begin{pmatrix} x_i(t) - \overline{x}_j(t) \\ y_i(t) - \overline{y}_j(t) \\ z_i(t) - \overline{z}_j(t) \end{pmatrix}^T \mathbf{S} \begin{pmatrix} x_i(t) - \overline{x}_j(t) \\ y_i(t) - \overline{y}_j(t) \\ z_i(t) - \overline{z}_j(t) \end{pmatrix} + 1 \leq 0 \tag{3.2}$$

Note that $(\overline{x}_j(t), \overline{y}_j(t), \overline{z}_j(t))$ is a known constant in (3.2). Fig.3.1 shows how the process of using (3.2) to formulate decoupled optimization problems for each robot. Existing works differ in their method of computing the prediction $(\overline{x}_j(t), \overline{y}_j(t), \overline{z}_j(t))$. The simplest possibility is to set it as the solution obtained in the previous iteration [34], which is what we use in our formulation as well.

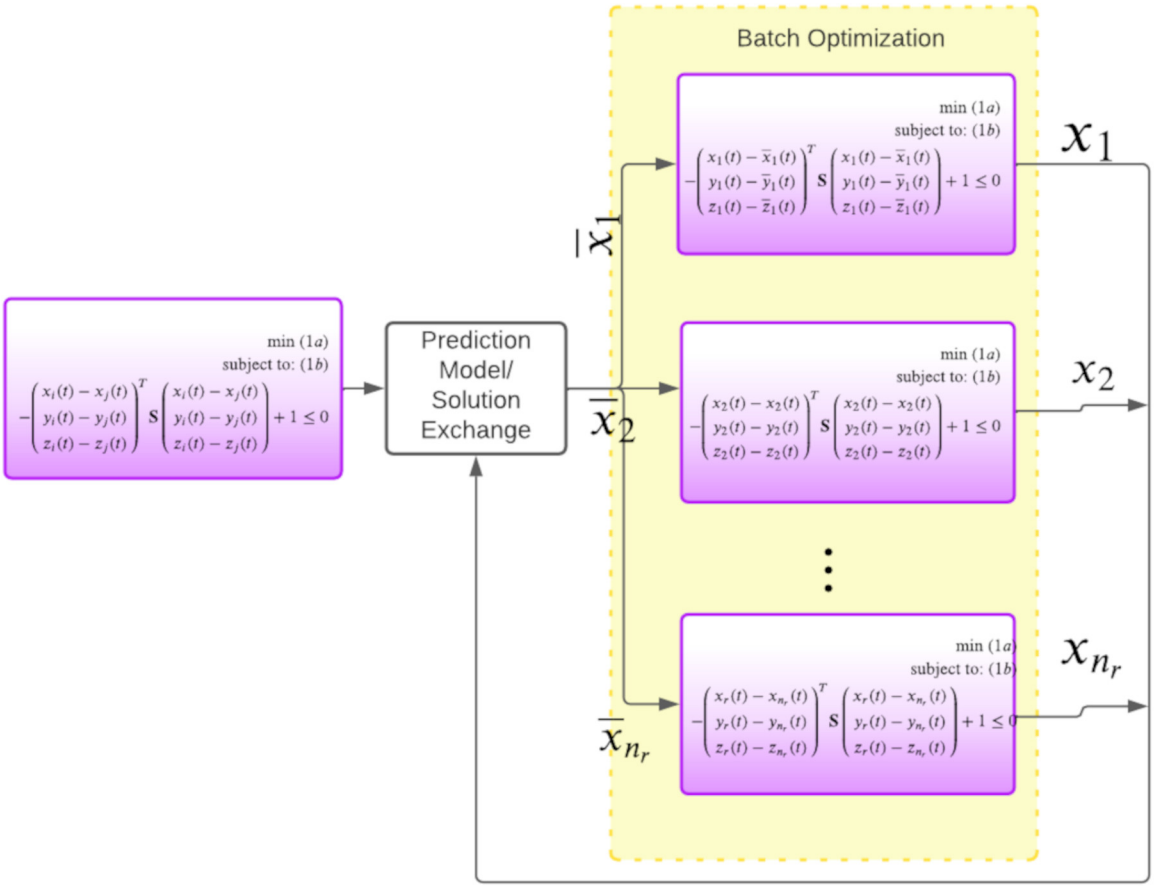


Figure 3.1: Figure shows our approach for breaking the joint optimization (first block) at each iteration into decoupled sub-problems (third group of blocks). Each robot exchanges their current computed trajectories with each other. For the next iteration, robots will use the communicated trajectories to frame their collision avoidance constraints. Thus, in this manner, we can avoid the inter-robot coupling in collision constraints. Our core novelty in this chapter is a batch optimizer that can solve all the decoupled problems in parallel over GPUs. It is also possible to replace the trajectory exchange set-up with a model that predicts the nature of the robot trajectories in the immediate next iteration. Note that we only show the x component of the trajectory purely to maintain clarity in the figure.

3.2.4.4 Online DMPC

DMPC approaches are the online variants of the distributed optimization approach. In other words, if we run one iteration of distributed optimization and let each robot move with the computed trajectory, we recover the DMPC works such as [8], [33]. This insight also points to the main issue of DMPC. At each control cycle, imagine a robot i receiving information (directly through communications or indirectly through prediction) about the trajectory that the other robot j computed. Then it uses this information to construct collision avoidance constraints in its trajectory optimization set-up. However, robot j will follow the same process and update its trajectory as well. Thus, essentially both robot i and j compute their motion based on outdated information about each other's behavior.

3.2.4.5 Batch optimization over CPU Vs GPU

Parallelization of a batch of optimization problems across CPUs and GPUs operates fundamentally differently, and both classes of approaches have been tried in existing works to speed up multi-robot trajectory optimization. Each CPU core is efficient at handling arbitrary numerical computations, and thus solving a batch of optimizations problems in parallel is conceptually simple. We can solve each problem in a separate thread without needing to make any change in the underlying numerical algebra of the optimizer [36], [34]. As mentioned earlier, the scalability of CPU parallelization is limited by the number of cores (typical 6 in a standard laptop). On the other hand, GPUs have many cores, but these are primarily efficient at parallelizing primitive operations such as matrix-vector and matrix-matrix multiplication. Moreover, GPUs excel in performing the same primitive operations over many data points. Thus, to fully leverage the compute power of GPUs, it is necessary to modify the underlying numerical aspect of an optimizer to fit the strengths of GPUs. For example, GPU acceleration of Newton's method requires adopting indirect matrix factorization over the more common direct approaches [39]. One optimization technique that trivially accelerates over GPUs is Gradient Descent (GD) since it boils down to just matrix-vector multiplication. Authors in [40] leverage this insight for developing a fast multi-robot trajectory optimization algorithm. One critical issue of [40] is that the proposed GD is very sensitive to hyper-parameters like weights of the different cost function, learning rate, etc.

GPUs are designed using threads grouped into blocks, which are themselves organized as grids to parallelize computations for computational efficiency [41]. The GPU first tiles an $n \times n$ matrix using $p \times q$ tiles indexed with a 2-dimensional index to multiply large matrices. The output of each tile in the result matrix is independent of other tiles, which allows for parallelization. The parallelized CUDA code uses a block of threads to compute each tile of the result matrix, and to compute the entire result matrix; it uses a $\frac{n}{p} \times \frac{n}{q}$ grid of thread blocks. Many threads and blocks in modern GPUs allow for simultaneous computation of tile outputs, allowing for a many-fold boost in the computation time required for large matrix multiplications. Most off-the-shelf GPU-based libraries have this inbuilt CUDA programming for parallel GPU computations and can be utilized for achieving computational speed-ups in matrix multiplications.

3.3 Methods

3.3.1 Overview

Similar to [34], we break the joint multi-robot trajectory optimization (3.1a)-(3.1d) into decoupled smaller sub-problems at each iteration. This is illustrated in Fig.3.1. At a conceptual level, this decoupling process can be interpreted in the following manner: the robots communicate among themselves the trajectories they obtained in the previous iteration of the optimizer. Each robot then uses them to independently formulate their collision avoidance constraints. Our work differs from existing works in the way the decoupled problems illustrated in Fig.3.1 is solved. As mentioned before, a trivial approach to solving the sub-problems in parallel CPU threads is not scalable for tens of robots. In contrast, our main idea in this chapter is to develop a GPU accelerated optimizer that can solve a batch of optimization problems in one shot.

In this sub-section, we aim to provide a succinct mathematical abstraction of our main idea. We discuss a special class of problems that are simple to solve in batch fashion. To this end, consider the following batch of equality constrained QPs, $i \in \{1, 2, ..., n_r\}$.

$$\min_{\boldsymbol{\xi}_i} \Big(\frac{1}{2} \boldsymbol{\xi}_i^T \overline{\mathbf{Q}} \boldsymbol{\xi}_i + \overline{\mathbf{q}}_i^T \boldsymbol{\xi}_i \Big), \qquad \text{st: } \overline{\mathbf{A}} \boldsymbol{\xi}_i = \overline{\mathbf{b}}_i \tag{3.3}$$

In total, there are n_r QPs to be solved, each defined over variable $\boldsymbol{\xi}_i$. The QPs defined in (3.3) have a unique structure. The Hessian $\overline{\mathbf{Q}}$ and the constrained matrix $\overline{\mathbf{A}}$ are shared across the problems and only the vectors $\overline{\mathbf{q}}_i$ and $\overline{\mathbf{b}}_i$ varies across the batch. This special structure leads to efficient batch solution formulae. To see how, note that each QP in the batch can be reduced to solving the following set of linear equations.

$$\begin{bmatrix} \overline{\mathbf{Q}} & \overline{\mathbf{A}}^T \\ \overline{\mathbf{A}} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\xi}_i \\ \boldsymbol{\mu}_i \end{bmatrix} = \begin{bmatrix} \overline{\mathbf{q}}_i \\ \overline{\mathbf{b}}_i \end{bmatrix}, \qquad \forall i \in \{1, 2, ..., n_r\} \tag{3.4}$$

where $\boldsymbol{\mu}_i$ are the dual optimization variables. Now, it can be observed that the matrix on the left-hand side of (3.4) is independent of the batch index i, and thus, the solutions for the entire batch can be computed in one shot through (3.5).

$$\left[\begin{array}{c|c|c} \boldsymbol{\xi}_1 & \dots & \boldsymbol{\xi}_{n_r} \\ \boldsymbol{\mu}_1 & \dots & \boldsymbol{\mu}_{n_r} \end{array}\right] = \overbrace{\left(\begin{bmatrix} \overline{\mathbf{Q}} & \overline{\mathbf{A}}^T \\ \overline{\mathbf{A}} & \mathbf{0} \end{bmatrix}^{-1} \right)}^{matrix} \overbrace{\left[\begin{array}{c|c|c|c} \overline{\mathbf{q}}_1 & \overline{\mathbf{q}}_2 & \dots & \overline{\mathbf{q}}_{n_r} \\ \overline{\mathbf{b}}_1 & \overline{\mathbf{b}}_2 & \dots & \overline{\mathbf{b}}_{n_r} \end{array}\right]}^{\text{stacked vectors}}, \tag{3.5}$$

where $|$ represents that the columns are stacked horizontally. The batch solution (3.5) amounts to multiplying one single matrix with a batch of vectors. Furthermore, the matrix is constant, and its

dimension is independent of the number of problems in the batch. Thus, operation (3.5) can be trivially parallelized over GPUs using off-the-shelf libraries like JAX [22].

How it all fits: In the next few sub-sections, we will show how the distributed sub-problems of (3.1a)-(3.1d), shown in Fig.**??** can be solved efficiently in a batch setting. Specifically, we reformulate these problems in such a way that the most intensive part of their solution process reduces to solving a batch of QPs with the special structure presented in (3.3).

3.3.2 Collision Avoidance in Polar Form

An important building block of our approach is rephrasing the collision avoidance constraints into the following polar representation from [2], [42].

$$\mathbf{f}_c(x_i(t), y_i(t)) = \left\{ \begin{array}{l} x_i(t) - \overline{x}_j(t) - ad_{ij}(t)\sin\beta_{ij}(t)\cos\alpha_{ij}(t) \\ y_i(t) - \overline{y}_j(t) - ad_{ij}(t)\sin\beta_{ij}(t)\sin\alpha_{ij}(t) \\ z_i(t) - \overline{z}_j(t) - bd_{ij}(t)\cos\beta_{ij}(t) \end{array} \right\}, \qquad d_{ij}(t) \geq 1, \tag{3.6}$$

where $\alpha_{ij}(t), \beta_{ij}(t), d_{ij}(t)$ are unknown variables that will be computed by the optimizer along with each robot's trajectory. Physically, $\alpha_{ij}(t)$ and $\beta ij(t)$ represent the 3D solid angle of the line-of-sight connecting robot i and j based on the predicted motion of the latter. The variable $d_{ij}(t)$ is the ratio of the length of the line-of-sight vector with minimum safe distance $\sqrt{a^2 + a^2 + b^2}$ (see [2]).

3.3.3 Proposed Reformulated Distributed Problem

Using (3.6), we can reformulate the distributed sub-problems presented in Fig.**??** for the i^{th} robot in the following manner. We reiterate that $(\overline{x}_j(t), \overline{y}_j(t), \overline{z}_j(t)), \forall j \neq i$ is known based on the prediction of the trajectories of other robots.

$$\min_{x_i(t),\, y_i(t),\, z_i(t),\, d_{ij}(t),\, \alpha_{ij}(t),\, \beta_{ij}(t)} \sum_t \Big(\ddot{x}_i^2(t) + \ddot{y}_i^2(t) + \ddot{z}_i^2(t) \Big), \tag{3.7a}$$

$$\Big(x_i(t_0), \dot{x}_i(t_0), \ddot{x}_i(t_0), y_i(t_0), \dot{y}_i(t_0), \ddot{y}_i(t_0), z_i(t_0), \dot{z}_i(t_0), \ddot{z}_i(t_0) \Big) = \mathbf{b}_{o,i}, \forall i \tag{3.7b}$$

$$\Big(x_i(t_f), \dot{x}_i(t_f), \ddot{x}_i(t_f), y_i(t_f), \dot{y}_i(t_f), \ddot{y}_i(t_f), z_i(t_f), \dot{z}_i(t_f), \ddot{z}_i(t_f) \Big) = \mathbf{b}_{f,i}, \forall i \tag{3.7c}$$

$$\mathbf{f}_c: \left\{ \begin{array}{l} x_i(t) - \overline{x}_j(t) - ad_{ij}(t)\sin\beta_{ij}(t)\cos\alpha_{ij}(t) \\ y_i(t) - \overline{y}_j(t) - ad_{ij}(t)\sin\beta_{ij}(t)\sin\alpha_{ij}(t) \\ z_i(t) - \overline{z}_j(t) - bd_{ij}(t)\cos\beta_{ij}(t) \end{array} \right\} \tag{3.7d}$$

$$d_{ij}(t) \geq 1, \qquad \forall t, j, \{j | j \in \{1, 2, ..., n_r\}, j \neq i\} \tag{3.7e}$$

3.3.3.1 Finite Dimensional Representation

Optimization (3.7a)-(3.7e) is expressed in terms of functions and thus has the so called infinite dimensional representaton. To obtain a finite-dimensional form, we assume some parametric form for this functions. For different $d_{ij}(t), \alpha_{a_{ij}}(t), \beta_{ij}(t)$, we assume a way-point paramterization. That is, these functions are represented through values at discrete time instants. The trajectories along each motion axis are represented as following polynomials.

$$\begin{bmatrix} x_i(t_1) \\ x_i(t_2) \\ \vdots \\ x_i(t_{n_p}) \end{bmatrix} = \mathbf{P}\mathbf{c}_{x,i}, \qquad \begin{bmatrix} \dot{x}_i(t_1) \\ \dot{x}_i(t_2) \\ \vdots \\ \dot{x}_i(t_{n_p}) \end{bmatrix} = \dot{\mathbf{P}}\mathbf{c}_{x,i}, \qquad \begin{bmatrix} \ddot{x}_i(t_1) \\ \ddot{x}_i(t_2) \\ \vdots \\ \ddot{x}_i(t_{n_p}) \end{bmatrix} = \ddot{\mathbf{P}}\mathbf{c}_{x,i}. \tag{3.8}$$

Similar expressions as (3.8) can be written for the y, z component of the trajectory as well. The matrix $\mathbf{P}$ is formed with time dependent polynomial basis functions. Using (3.8), we can re-write (3.7a)-(3.7e) in the following matrix form.

$$\min_{\boldsymbol{\xi}_{1,i},\boldsymbol{\xi}_{2,i},\boldsymbol{\xi}_{3,i},\boldsymbol{\xi}_{4,i}} \Big(\frac{1}{2}\boldsymbol{\xi}_{1,i}^T\mathbf{Q}\boldsymbol{\xi}_{1,i}\Big), \tag{3.9a}$$

$$\mathbf{A}_{eq}\boldsymbol{\xi}_{1,i} = \mathbf{b}_{eq}, \tag{3.9b}$$

$$\mathbf{F}\boldsymbol{\xi}_{1,i} = \mathbf{g}_i(\boldsymbol{\xi}_{2,i}, \boldsymbol{\xi}_{3,i}, \boldsymbol{\xi}_{4,i}), \tag{3.9c}$$

$$\boldsymbol{\xi}_{4,i} \geq \mathbf{1}, \tag{3.9d}$$

where, $\boldsymbol{\xi}_{1,i} = (\mathbf{c}_{x,i}, \mathbf{c}_{y,i}, \mathbf{c}_{z,i})$, $\boldsymbol{\xi}_{2,i} = \boldsymbol{\alpha}_{ij}$, $\boldsymbol{\xi}_{3,i} = \boldsymbol{\beta}_{ij}$ and $\boldsymbol{\xi}_{4,i} = \mathbf{d}_{ij}$. Note that $\boldsymbol{\alpha}_{ij}$ is formed by stacking $\alpha_{ij}(t)$ at different time instants. Similar construction is followed for other elements in $\boldsymbol{\xi}_2, \boldsymbol{\xi}_3$. The matrix $\mathbf{Q}$ is block diagonal matrix with $\ddot{\mathbf{P}}^T\ddot{\mathbf{P}}$ as main diagonal block. The affine constraint (3.9b) is a matrix representation of the initial and final boundary conditions (3.7b)-3.7c. The matrix $\mathbf{A}_{eq}$ and vector $\mathbf{b}_{eq}$ is constructed in the following manner.

$$\mathbf{A}_{eq} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{A} \end{bmatrix}, \mathbf{A}_{eq} = \Big[\mathbf{P}_1|\dot{\mathbf{P}}_1|\ddot{\mathbf{P}}_1|\mathbf{P}_{-1}|\dot{\mathbf{P}}_{-1}\ddot{\mathbf{P}}_{-1}\Big]^T, \mathbf{b}_{eq} = \begin{bmatrix} \mathbf{b}_{0,i} \\ \mathbf{b}_{f,i}, \end{bmatrix} \tag{3.10}$$

where, $\mathbf{P}_1, \dot{\mathbf{P}}_1, \ddot{\mathbf{P}}_1, \mathbf{P}_{-1}, \dot{\mathbf{P}}_{-1}, \ddot{\mathbf{P}}_{-1}$ represents the first and last elements of the corresponding matrices. Matrix $\mathbf{F}$ and vector $\mathbf{g}_i$ defining constraints (3.7e) are constructed as

$$\mathbf{F} = \begin{bmatrix} \mathbf{F}_o & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_o & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{F}_o \end{bmatrix} \qquad \mathbf{g}_i = \begin{bmatrix} \mathbf{g}_{x,i}(\boldsymbol{\xi}_{2,i}, \boldsymbol{\xi}_{3,i}, \boldsymbol{\xi}_{4,i}) \\ \mathbf{g}_{y,i}(\boldsymbol{\xi}_{2,i}, \boldsymbol{\xi}_{3,i}, \boldsymbol{\xi}_{4,i}) \\ \mathbf{g}_{z,i}(\boldsymbol{\xi}_{2,i}, \boldsymbol{\xi}_{3,i}, \boldsymbol{\xi}_{4,i}) \end{bmatrix} \tag{3.11}$$

where,

$$\mathbf{g}_{x,i} = \overline{\mathbf{x}}_j + a\mathbf{d}_{ij}\sin\boldsymbol{\beta}_{ij}\cos\boldsymbol{\alpha}_{ij}, \forall j, \qquad \mathbf{g}_{y,i} = \overline{\mathbf{y}}_j + a\mathbf{d}_{ij}\sin\boldsymbol{\beta}_{ij}\sin\boldsymbol{\alpha}_{ij}, \forall j, \qquad \mathbf{g}_{z,i} = \overline{\mathbf{z}}_j + b\mathbf{d}_{ij}\cos\boldsymbol{\beta}_{ij}, \forall j \tag{3.12}$$

and $\mathbf{F}_o$ is formed by vertically stacking $\mathbf{P}$, $n_r - 1$ times. The vectors $\overline{\mathbf{x}}_j, \overline{\mathbf{y}}_j, \overline{\mathbf{z}}_j$ are formed by stacking $\overline{x}_j(t), \overline{y}_j(t), \overline{z}_j(t)$ at different time instants.

Remark 1 *The subscript i signifies that* (3.9a)-(3.9c) *is constructed for the i^{th} agent.*

Remark 2 *All the non-convexity in optimization (3.9a)-(3.9d) is rolled into the equality constraint (3.9c)*

Remark 3 *The matrices $\boldsymbol{A}_{eq}, \boldsymbol{F}$ in optimization (3.9a)-(3.9d) is independent of the robot index. In other words, these matrices remain the same irrespective of the which sub-problems shown in Fig.**??**, we are solving.*

Remark 3 sheds light behind our motivation of presenting the elaborate reformulations of the collision avoidance constraints. In fact, on the surface, our chosen representation (3.6) seems substantially more complicated than the conventional form (3.1d) based on the Euclidean norm. In the next sub-section, we present an optimizer that can leverage the insights presented in Remark 3. More precisely, we will show that the due to the matrices $\mathbf{A}_{eq}, \mathbf{F}$ being independent of the robot index i, the most intensive part of solving (3.9a)-(3.9d) reduces to the batch QP structure presented in sub-section 3.3.1

3.3.4 Augmented Lagrangian and Alternating Minimization

Our proposed optimizer for (3.9a)-(3.9d) relies on relaxing the non-convex equality constraints (3.9c) as l_2 penalties and incorporating them into the cost function in the following manner.

$$\min_{\boldsymbol{\xi}_{1,i},\boldsymbol{\xi}_{2,i},\boldsymbol{\xi}_{3,i},\boldsymbol{\xi}_{4,i}} \left(\frac{1}{2}\boldsymbol{\xi}_{1,i}^T\mathbf{Q}\boldsymbol{\xi}_{1,i} - \langle\boldsymbol{\lambda}_i, \boldsymbol{\xi}_{1,i}\rangle + \frac{\rho}{2}\left\|\mathbf{F}\boldsymbol{\xi}_{1,i} - \mathbf{g}_i(\boldsymbol{\xi}_{2,i},\boldsymbol{\xi}_{3,i},\boldsymbol{\xi}_{4,i})\right\|_2^2\right) \tag{3.13}$$

As the residual of the constraint term is driven to zero, we recover the solution to the original problem. To this end, the parameter $\boldsymbol{\lambda}_i$, known as the Lagrange multiplier, plays an important part. Its role is to appropriately weaken the effect of the primary cost function so that the optimizer can focus on minimizing the constraint residual [43]. The parameter ρ is a scalar and is typically constant. However, it is possible to increase or decrease it depending on the magnitude of the constraint residual at each iteration of the optimizer.

The relaxation of non-convex equality constraints, as augmented Lagrangian (AL) cost, is extensively used in non-convex optimization [44], [45]. However, what differentiates our use of AL from existing works is how we minimize (3.13). Typical approaches towards non-convex optimization are based on

first (and sometimes second) order Taylor Series expansion of the non-convex costs or constraints. In contrast, we adopt an Alternating Minimization (AM) based approach, wherein at each iteration, we minimize only one of the variable blocks amongst $\boldsymbol{\xi}_{1,i}, \boldsymbol{\xi}_{2,i}, \boldsymbol{\xi}_{3,i}, \boldsymbol{\xi}_{4,i}$ while others are held constant at specific values. In the next section, we present the various steps of our AM optimizer and highlight how it never requires any linearization of cost or constraints. Moreover, we show how the AM steps naturally lead to a simple yet efficient batch update rule using which we can solve (3.9a)-(3.9d) for all the robots in one shot.

Algorithm 1 Alternating Minimization (AM) based solution for the i^{th} Sub-Problem

Initialize ${}^{k}\boldsymbol{\xi}_{2,i}$, ${}^{k}\boldsymbol{\xi}_{3,i}$ and ${}^{k}\boldsymbol{\xi}_{4,i}$ values at iteration $k = 0$. **while** $k \leq$ max iteration or till norm of the residuals are below some threshold

$$ {}^{k+1}\boldsymbol{\xi}_{1,i} = \min_{\boldsymbol{\xi}_{1,i}} \Big(\frac{1}{2} \boldsymbol{\xi}_{1,i}^T \mathbf{Q} \boldsymbol{\xi}_{1,l} - \langle {}^{k}\boldsymbol{\lambda}_i, \boldsymbol{\xi}_{1,i} \rangle + \frac{\rho}{2} \Big\| \mathbf{F}\boldsymbol{\xi}_{1,i} - \mathbf{g}_i({}^{k}\boldsymbol{\xi}_{2,i}, {}^{k}\boldsymbol{\xi}_{3,i}, {}^{k}\boldsymbol{\xi}_{4,i}) \Big\|_2^2 \Big), \qquad st. \mathbf{A}_{eq}\boldsymbol{\xi}_1 = \mathbf{b}_{eq} \tag{3.14} $$

$$ {}^{k+1}\boldsymbol{\xi}_{2,i} = \min_{\boldsymbol{\xi}_{2,i}} \Big(\frac{\rho}{2} \Big\| \mathbf{F}^{k+1}\boldsymbol{\xi}_{1,i} - \mathbf{g}_i(\boldsymbol{\xi}_{2,i}, {}^{k}\,\boldsymbol{\xi}_{3,i}, {}^{k}\,\boldsymbol{\xi}_{4,i}) \Big\|_2^2 \Big) \tag{3.15} $$

$$ {}^{k+1}\boldsymbol{\xi}_{3,i} = \min_{\boldsymbol{\xi}_{3,i}} \Big(\frac{\rho}{2} \Big\| \mathbf{F}^{k+1}\boldsymbol{\xi}_{1,i} - \mathbf{g}_i({}^{k+1}\boldsymbol{\xi}_{2,i}, \boldsymbol{\xi}_{3,i}, {}^{k}\,\boldsymbol{\xi}_{4,i}) \Big\|_2^2 \Big) \tag{3.16} $$

$$ {}^{k+1}\boldsymbol{\xi}_{4,i} = \min_{\boldsymbol{\xi}_{4,i}} \Big(\frac{\rho}{2} \Big\| \mathbf{F}^{k+1}\boldsymbol{\xi}_{1,i} - \mathbf{g}_i({}^{k+1}\boldsymbol{\xi}_{2,i}, {}^{k+1}\,\boldsymbol{\xi}_{3,i}, \boldsymbol{\xi}_{4,i}) \Big\|_2^2 \Big) \tag{3.17} $$

$$ {}^{k+1}\boldsymbol{\lambda}_i = {}^{k}\boldsymbol{\lambda}_i - \rho({}^{k+1}\mathbf{F}\,{}^{k+1}\boldsymbol{\xi}_{1,i} - \mathbf{g}_i({}^{k+1}\boldsymbol{\xi}_{2,i}, {}^{k+1}\,\boldsymbol{\xi}_{3,i}, {}^{k+1}\,\boldsymbol{\xi}_{4,i}))\mathbf{F} \tag{3.18} $$

Return ${}^{k+1}\overline{\boldsymbol{\xi}}_{1,i}$

3.3.5 AM Steps and Batch Update Rule

Our AM based optimizer for minimizing (3.13) subject to (3.9b)-(3.9d) is presented in Algorithm 1. Here, the left superscript k is used to track the values of the variable across iteration. For example, ${}^{k}\boldsymbol{\xi}_{2,i}$ denotes the value of this respective variable at iteration k.

The Algorithm begins (line 1) by providing the initial guesses for $\boldsymbol{\xi}_{2,i}, \boldsymbol{\xi}_{3,i}, \boldsymbol{\xi}_{4,i}$. The main optimizer iterations run within the while loop for the specified max iteration limit or till the constraint residuals are a below specified threshold. Each step within the while loop involves solving a convex optimization over just one variable block. We present a more detailed analysis of each of the steps next.

3.3.5.1 Analysis

Step (3.14)**:** This optimization is a convex QP with a similar structure as (3.3) with

$$\overline{\mathbf{Q}} = \mathbf{Q} + \rho \mathbf{F}^T \mathbf{F}, \qquad \overline{\mathbf{q}}_i = -{}^k\boldsymbol{\lambda}_i - (\rho \mathbf{F}^T \mathbf{g}_i({}^k\boldsymbol{\xi}_{2,i}, {}^k\boldsymbol{\xi}_{3,i}, {}^k\boldsymbol{\xi}_{4,i}))^T. \tag{3.19}$$

Thus, we can easily solve (3.14) for all the robots in parallel to obtain $(\boldsymbol{\xi}_{1,1}, \boldsymbol{\xi}_{1,2}, \boldsymbol{\xi}_{1,3}, \ldots, \boldsymbol{\xi}_{1,n_r})$ in one shot. The exact solution update is given by (3.5).
For a constant ρ, the inverse of $\overline{\mathbf{Q}}$ needs to be obtained only once irrespective of the number of robots. Thus, the complexity of the batch solution of all the sub-problems stems purely from obtaining the matrix-matrix products in (3.5) and $\mathbf{F}^T\mathbf{g}_i, \forall i$. We can formulate the latter also as one large matrix-matrix product in the following manner.

$$\mathbf{F}^T(\overbrace{\Big[\mathbf{g}_1|\mathbf{g}_2|\ldots|\mathbf{g}_{n_r}\Big]}^{\mathbf{G}})^T \tag{3.20}$$

The dimension of $\mathbf{F}$, $\mathbf{g}_i$ and $\mathbf{G}$ is $((n_r-1)*n_p) \times 3n_v$, $((n_r-1)*n_p) \times 1$, and $((n_r-1)*n_p) \times n_r$ respectively. For convenience, we recall that n_r, n_p, n_v represents the number of robots, planning steps and coefficients of the trajectory polynomial (along each axis) respectively. Thus, the row-dimension of $\mathbf{F}$ and $\mathbf{G}$ increases linearly with n_r.
Step (3.15)**:** The variable ${}^{k+1}\boldsymbol{\xi}_{1,i}$ computed in the previous step and (3.8) can be used to fix the position trajectory ${}^{k+1}\boldsymbol{x}_i$, ${}^{k+1}\boldsymbol{y}_i$, ${}^{k+1}\boldsymbol{z}_i$ at the $(k+1)^{th}$ iteration. Thus, optimization (3.15) reduces the following form

$$\forall i,j, \qquad {}^{k+1}\boldsymbol{\alpha}_{ij} = \min_{\boldsymbol{\alpha}_{ij}} \frac{\rho}{2} \left\| \begin{matrix} \overbrace{{}^{k+1}\mathbf{x}_i - \overline{\mathbf{x}}_j}^{{}^{k+1}\tilde{\boldsymbol{x}}_i} - a^k \mathbf{d}_{ij} \sin\boldsymbol{\beta}_{ij} \cos\boldsymbol{\alpha}_{ij} \\ \underbrace{{}^{k+1}\mathbf{y}_i - \overline{\mathbf{y}}_j}_{{}^{k+1}\tilde{\boldsymbol{y}}_i} - a^k \mathbf{d}_{ij} \sin\boldsymbol{\beta}_{ij} \sin\boldsymbol{\alpha}_{ij} \end{matrix} \right\|_2^2 \tag{3.21}$$

where $\overline{\mathbf{x}}_j, \overline{\mathbf{y}}_j$ is formed by stacking $\overline{x}_j(t), \overline{y}_j(t)$ at different time instants.

Although (3.21) is a seemingly non-convex problem but it has a few favorable computational structures. First, for a fixed position trajectory ${}^{k+1}\boldsymbol{x}_i$, ${}^{k+1}\boldsymbol{y}_i$, ${}^{k+1}\boldsymbol{z}_i$, we can treat each element of $\boldsymbol{\alpha}_{ij}$ as independent from each other. Thus , (3.21) reduces to $(n_r-1)*n_p$ decoupled problems. Second, the solution can be obtained by purely geometrical intuition; $\boldsymbol{\alpha}_{ij}$ is simply one part of the 3D solid-angle of the line-of-sight connecting the i^{th} robot and the predicted trajectory of j^{th} agent. The exact solution update is given by the following.

$${}^{k+1}\boldsymbol{\xi}_{2,i} = {}^{k+1}\boldsymbol{\alpha}_{ij} = \arctan 2({}^{k+1}\tilde{\boldsymbol{y}}_i, {}^{k+1}\tilde{\boldsymbol{x}}_i), \tag{3.22}$$

Step (3.16)**:** Following the exact same reasoning as the previous step (3.15), we have the following solution update rule for $\boldsymbol{\xi}_{3,i}$:

$$\boldsymbol{\xi}_{3,i}{}^{k+1} = {}^{k+1}\boldsymbol{\beta}_{ij} = \arctan 2(\frac{{}^{k+1}\tilde{\boldsymbol{x}}_i}{a\cos{}^{k+1}\boldsymbol{\alpha}_{ij}}, \frac{{}^{k+1}\tilde{\boldsymbol{z}}_i}{b}) \tag{3.23}$$

Step ${}^{k+1}\boldsymbol{\xi}_{4,i}$**:** Similar to the last two steps, each element of $\boldsymbol{\xi}_{4,i} = \mathbf{d}_{ij}$ once the position trajectory ${}^{k+1}\boldsymbol{x}_i$, ${}^{k+1}\boldsymbol{y}_i$, ${}^{k+1}\boldsymbol{z}_i$ is fixed. Thus, (3.17) can be broken down into $(n_r - 1) * n_p$ parallel problems of the following form.

$${}^{k+1}\boldsymbol{\xi}_{4,i} = {}^{k+1}\mathbf{d}_{ij} = \min_{\mathbf{d}_{ij}\geq 1} \frac{\rho}{2} \left\| \begin{matrix} \overbrace{{}^{k+1}\mathbf{x}_i - \mathbf{x}_j}^{{}^{k+1}\tilde{\boldsymbol{x}}_i} - a\mathbf{d}_{ij}\sin{}^{k+1}\boldsymbol{\beta}_{ij}\cos{}^{k+1}\boldsymbol{\alpha}_{ij} \\ \overbrace{{}^{k+1}\mathbf{y}_i - \mathbf{y}_j}^{{}^{k+1}\tilde{\boldsymbol{y}}_i} - a\mathbf{d}_{ij}\sin{}^{\boldsymbol{k+1}}\boldsymbol{\beta}_{ij}\sin{}^{k+1}\boldsymbol{\alpha}_{ij} \\ \overbrace{{}^{k+1}\mathbf{z}_i - \mathbf{z}_j}^{{}^{k+1}\tilde{\boldsymbol{z}}_i} - b\mathbf{d}_{ij}\cos{}^{k+1}\boldsymbol{\beta}_{ij} \end{matrix} \right\|_2^2 \tag{3.24}$$

Each optimization in (3.24) is a single variable QP with simple bound constraints. We first obtain the symbolic formulae for the unconstrained version and then clip the resulting solution to $[0\ 1]$.

Remark 4 *Evaluating* (3.22)*,* (3.23) *and the solution of 3.24 requires no matrix factorization/inverse or even matrix-matrix products. We just need element-wise operation that can obtained for all the sub-problems in one shot. In other words, we obtain* $(\boldsymbol{\xi}_{2,1}, \boldsymbol{\xi}_{2,2}, \dots \boldsymbol{\xi}_{2,n_r})$*,* $(\boldsymbol{\xi}_{3,1}, \boldsymbol{\xi}_{3,2}, \dots \boldsymbol{\xi}_{3,n_r})$*, and* $(\boldsymbol{\xi}_{4,1}, \boldsymbol{\xi}_{4,2}, \dots \boldsymbol{\xi}_{4,n_r})$ *in parallel.*

3.4 Results

The objective of this section is twofold. First, to validate that a distributed approach augmented with our custom batch optimizer can indeed generate collision-free trajectories for tens of robots in highly cluttered environments. Second, to compare our approach with the existing state-of-the-art (SOTA) multi-robot trajectory optimizer in terms of solutions quality and computation time.

Implementation Details: We built our optimizer in Python using JAX [22] as our GPU accelerated linear algebra back-end. We considered static obstacles as robots with fixed zero velocity. We modeled each robot by a sphere and each obstacle by its circumscribing sphere. We experiment with a diverse range of radii of both robots and obstacles. Simulations were run on a desktop computer with 32 GB RAM and RTX 2080 NVIDIA GPU.

3.4.1 Benchmarks and Convergence

Our optimizer is tested using the following benchmarks.

- The robots' start and goal positions are sampled along the circumference of a circle.
- The robots are initially located on a grid and are tasked to converge to a line formation.

By changing the number and positions of robots and static obstacles, we created several variations of the mentioned benchmarks and utilized them to validate our optimizer. Fig. 3.2(A-I) presents a few qualitative results in a diverse set of environments. Fig. 3.2(A-C) shows an environment with 32 robots and 20 obstacles. Interestingly, we observe a circular pattern formation among the robots while passing through narrow passages between static obstacles. In Fig. 3.2(D-F), 36 robots initially arranged in a grid are given the task to navigate to a line formation while avoiding collisions with each other and with the four static obstacles in the environment. Fig.3.3 shows the execution of the computed trajectories in a high-fidelity physics engine called Gazebo available in Robot Operation System [46].

A conceptually simple way of validating the convergence of the proposed optimizer is to observe the trends in residual of constraints (3.9c) over iterations. If the residuals converge to zero, the computed trajectories are guaranteed to be collision-free. Fig. 3.4 empirically provides this validation. It presents $\|\mathbf{F}\boldsymbol{\xi}_{1,i} - \mathbf{g}_i\|$ averaged over all i. Furthermore, we average the residuals over different trials in various benchmarks. We can observe from Fig.3.4 that, on average, 100 iterations are sufficient to obtain residuals around 0.01. A further increase in residuals can be obtained by increasing the number of iterations but at the expense of increasing the computation time. In our implementation, we adopt a heuristic wherein we inflate the size of the robots with four times the typical residual observed after 100 iterations.

For a further sanity check, we check for inter-robot and robot-obstacle distances at each point along the computed trajectories (Fig. 3.5). Collisions are considered to have happened if the distances are less than sum of the robots' (blue line in Fig.3.5) or robot-obstacles' (yellow line in Fig.3.5) radii. Fig 3.5 summarizes the average behavior observed across several trials, which validates the satisfaction of the collision avoidance requirement.

3.4.2 Comparisons With State-of-the-Art

This subsection compares our optimizer with existing state-of-the-art approaches [2] and [1]. We use the metrics Smoothness Cost, Arc-Length and Computation Time as discussed in Section 2.3.4.

3.4.2.1 Comparison with [1]

Fig. 3.6 presents a qualitative comparison between the trajectories obtained by our optimizer and [1] in two different benchmarks. Both approaches are successful; however, ours results in shorter trajectories. This trend is further confirmed by Fig.3.7. Our optimizer achieves an average reduction of 3.90% and 13.72% in arc-length in 16 and 32 robots benchmarks, respectively. Furthermore, the performance gap

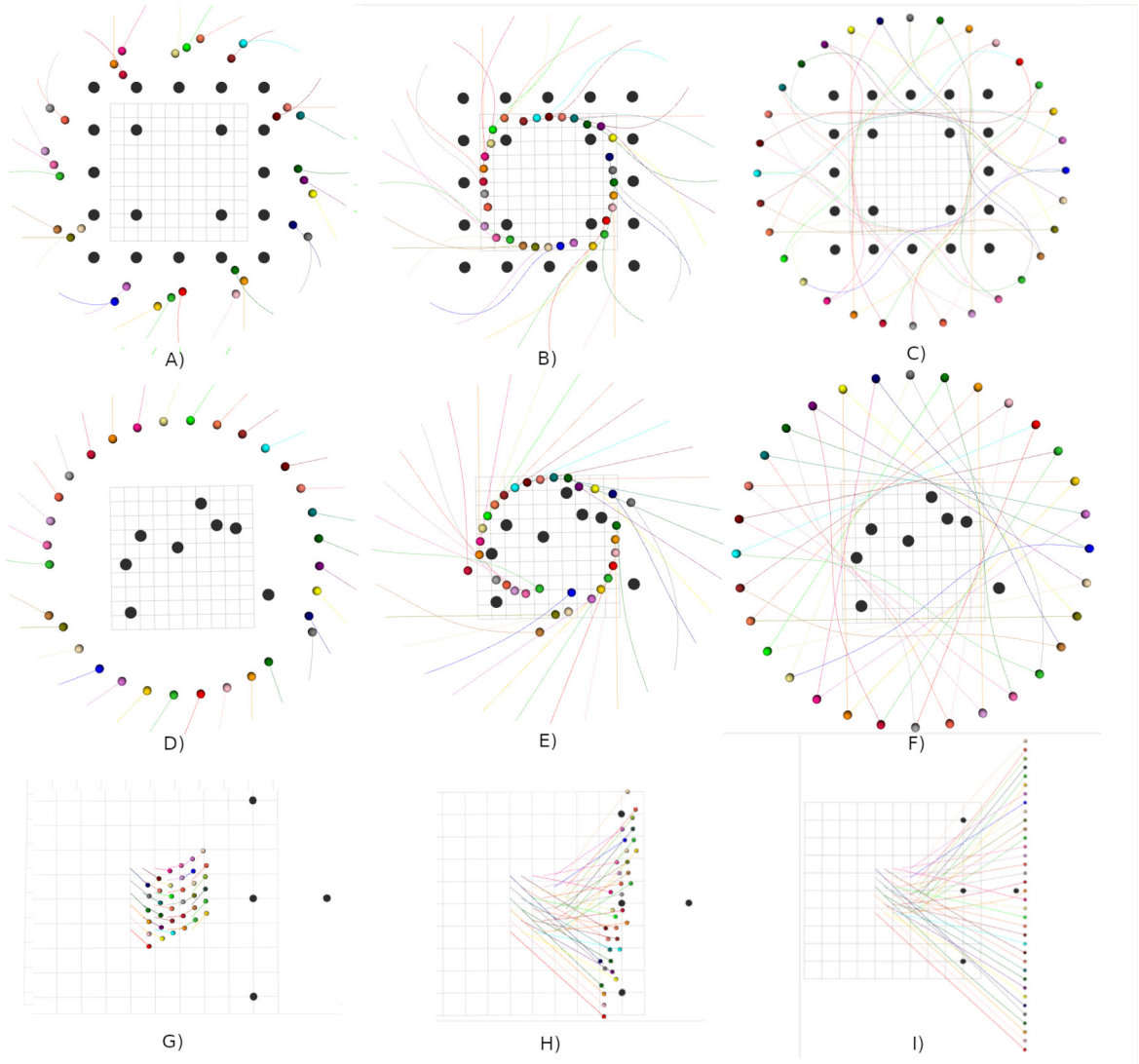


Figure 3.2: Trajectory snapshots for (A-C) 32 robots, each of radius 0.3m arranged in a circle and 20 obstacles of radius 0.4m, (D-F) 32 robots, each of radius 0.3m arranged in a circle and 8 randomly placed obstacles of radius 0.4m, (G-I) 36 robots, each of radius 0.1m arranged in a grid configuration are required to move to a line formation. Also, the environment has 4 static obstacles, each of radius 0.15m.

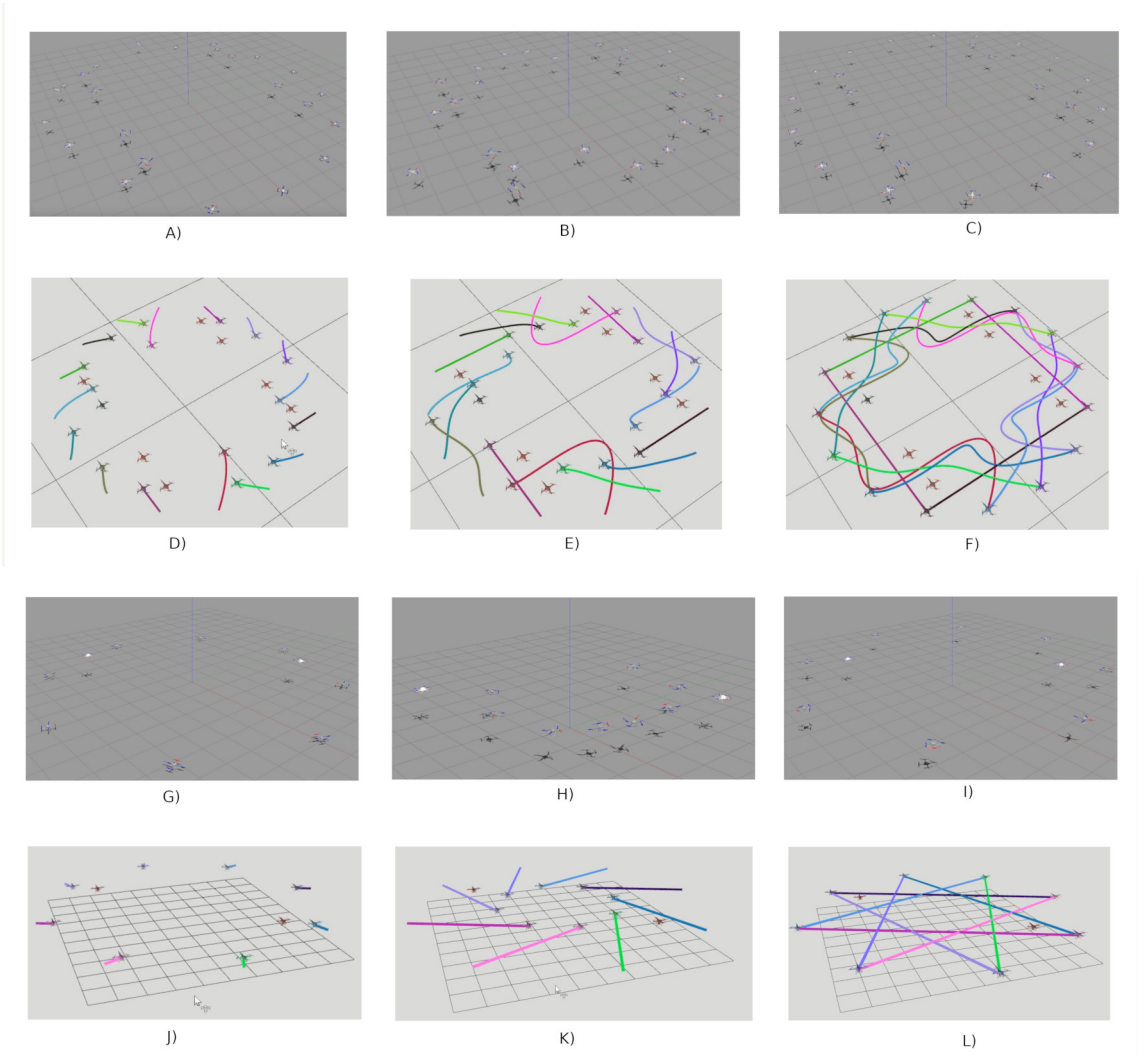


Figure 3.3: Simulation snapshots for (A-F) 16 drones and 8 static obstacles of radius 0.3m, and (G-L) 8 drones and 2 static obstacles of radius 0.3m. (A-C) and (G-I) are screenshots of simulations on Gazebo. In (A-C), the gray static hovering drones represent static obstacles while in (G-I) white hovering drones represent static obstacles. (D-F) and (J-L) are screenshots of RViz simulations with the brown hovering drones representing static obstacles. For the full simulation videos, please refer to the following link: `https://www.dropbox.com/scl/fo/xnostapkvf72uudyb840t/h?dl=0&rlkey=02gjjllohbomzi0kcifbqezt9`

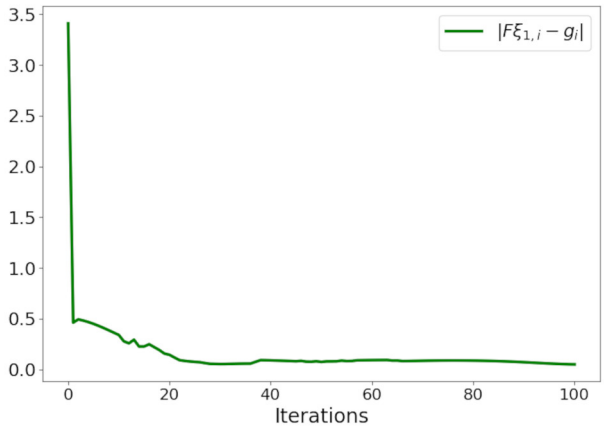


Figure 3.4: Empirical validation of convergence of our optimizer. The figure shows the residual of $\|\mathbf{F}\boldsymbol{\xi}_{1,i} - \mathbf{g}_i\|$ averaged over all i (agent index) and across different benchmarks.

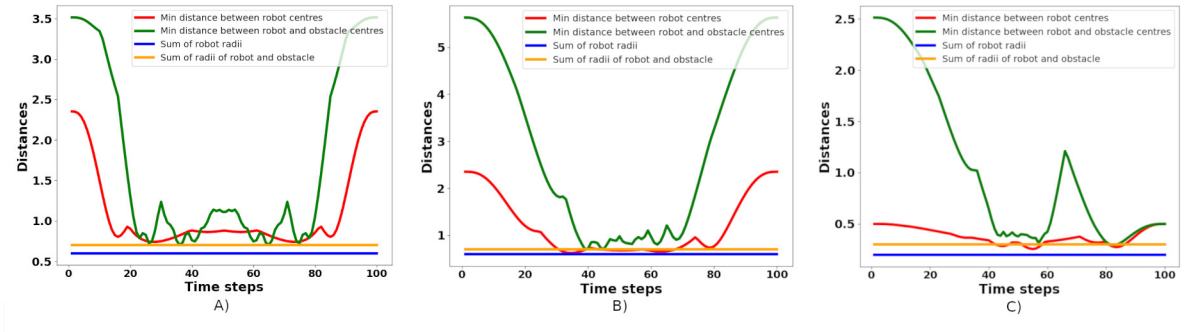


Figure 3.5: Figure showing the minimum of the pair-wise distances between the robots averaged across different benchmarks, some of which are shown in Fig.3.2. The pair-wise distance is always greater than the lower bound shown in blue. Similarly, we also show the average minimum distance between the robots' and obstacles' centers. The corresponding lower bound is shown in yellow which is always respected by the computed trajectories.

between our optimizer and [1] increases as the environment becomes more cluttered with static obstacles. We also observe similar trends in the smoothness metric, with the performance gap being even starker. Our optimizer achieves an average reduction of 35.86% and 59.06% in smoothness cost in 16 and 32 robots benchmark, respectively.

Table 3.2 compares the computation time of our optimizer and [1]. Our optimizer shows better scaling with the number of robots and obstacles in the environment. On the considered benchmark, our optimizer shows a worst and best case improvement of 74.28% and 98.48% respectively. The trends in computation time can be understood in the following manner. The approach of [1] uses sampling-based multi-agent pathfinding algorithms to compute initial guesses for the robot trajectories. As the environment becomes more cluttered, the computational cost of computing the initial trajectories increases dramatically. Moreover, their sequential optimization also becomes increasingly more computationally intensive as the number of robots and obstacle increase.

In contrast, our optimizer only requires matrix-matrix products, and the dimension of these matrices increases linearly with the number of robots and obstacles. This linear scaling along with GPU parallelization explains our computation time.

3.4.2.2 Comparison with [2]

Table. 3.3 compares the performance of our optimizer with [2]. Our core difference with [2] stems from the fact that we break a large optimization problem into smaller distributed sub-problems. In contrast, [2] retains the original larger problem itself. However, both our optimizer and [2] use GPUs to accelerate the underlying numerical computations. Thus, unsurprisingly, [2] shows a decent scaling with the number of robots and obstacles. Nevertheless, our approach still outperforms [2]. Specifically, in 16 robot benchmarks, our optimizer shows a worst-case improvement of 2 times over [2] in computation time. As the environment becomes more cluttered, this factor increases to almost 10. In 32 robot benchmarks, the difference between our optimizer and [2]'s computation time is around nine times.

In terms of the arc-length and the smoothness metrics, our optimizer shows an improvement of around 57% and 58% respectively over [2]. However, both approaches provide comparable results in the more challenging 32 robot benchmarks. The arc-length and smoothness cost difference decreases as the environment becomes more cluttered.

3.5 Ablation Study

3.5.1 Initializations using Reciprocal Velocity Obstacle(RVO)[3]

RVO is a local, reactive multi-robot trajectory planner that has been shown to generate collision-free trajectories in real-time for hundreds of robots[3]. Even though it runs in real-time, RVO trajectories cannot be directly executed by Robots as they are jittery and jerky. We use a trick to precompute trajectories using RVO for given start and goal positions of multiple Robots and use the RVO trajectories

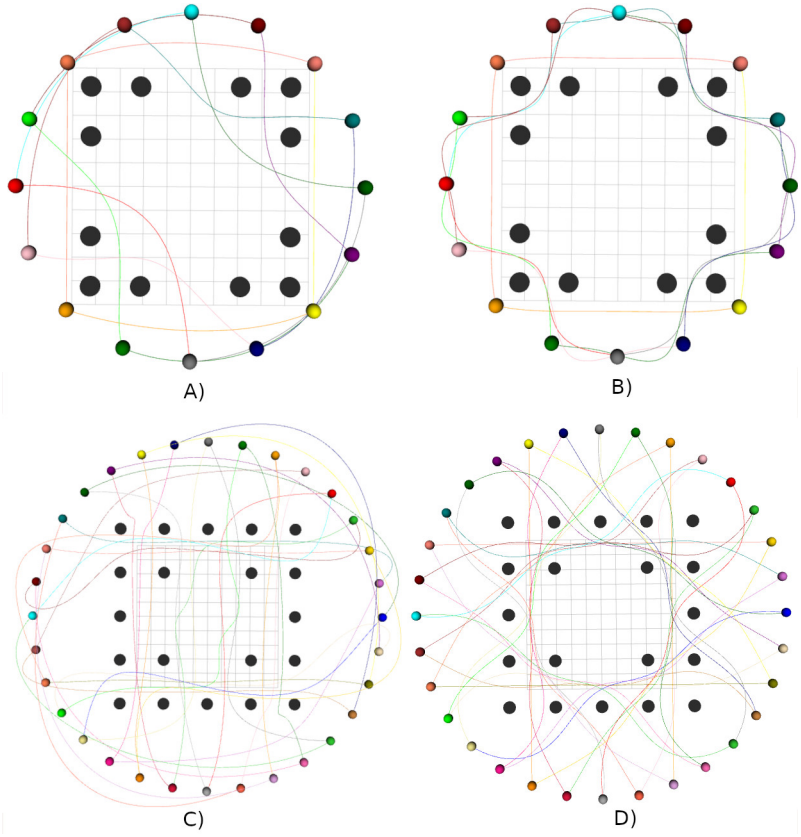


Figure 3.6: Comparison of trajectories generated by [1](A),C)) and our optimizer(B),D)) for 16 robots-12 obstacles (upper row) and 32 robots-20 obstacles (bottom row) benchmarks. Black spheres denote static obstacles and colored spheres denote robots. Our optimizer generates trajectories with smaller arc-length than [1].

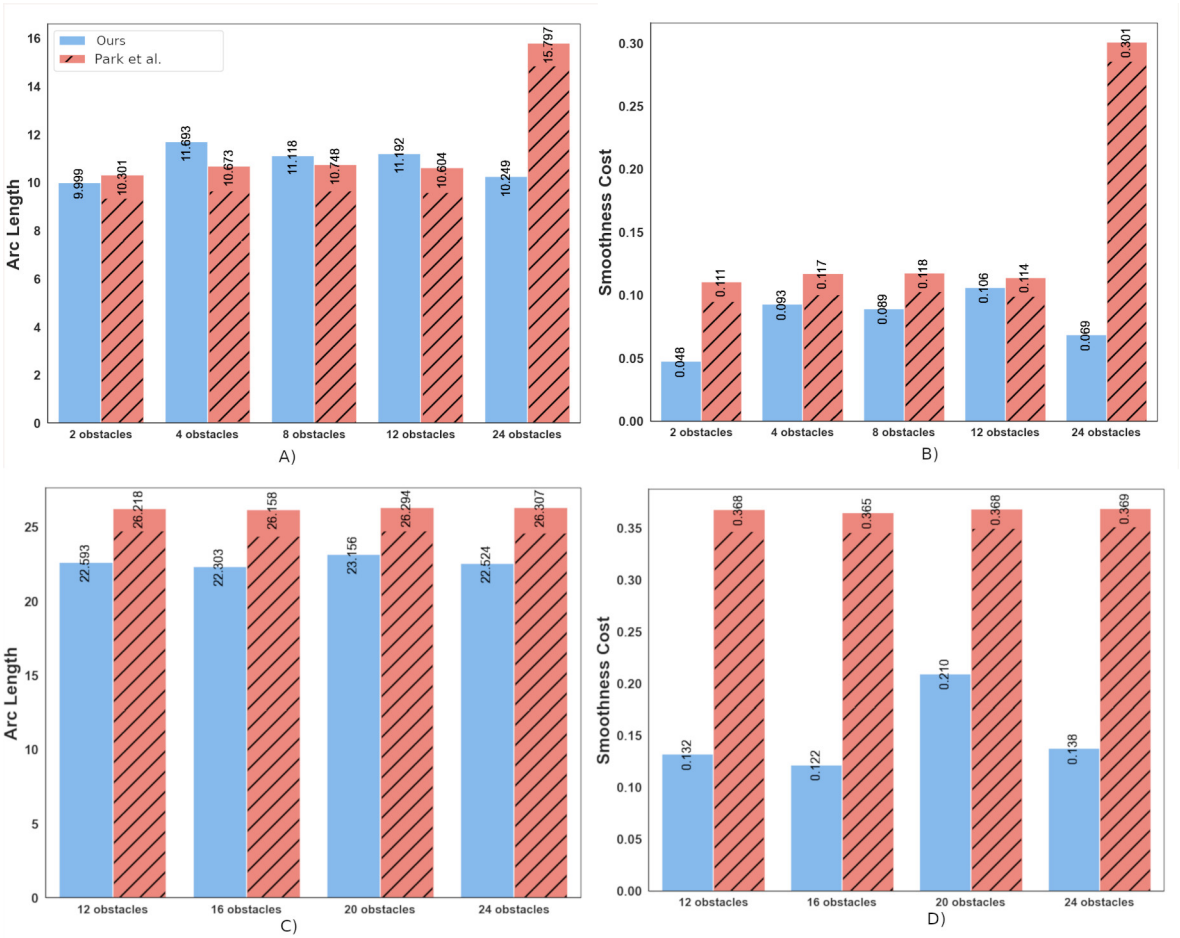


Figure 3.7: Comparison of our optimizer with [1] in terms of arc-length and smoothness of obtained trajectories in 16 robots (A,B) and 32 robots (C,D) benchmarks. Our optimizer generates trajectories with not only better smoothness, but also with shorter arc-lengths. Moreover, the performance gap between our approach and [1] increases as the environment becomes more cluttered.

along with other optimizer parameters computed from them to initialize our optimizer. Thus the task of eliminating collisions is first performed by RVO and further enhanced by our optimizer. We consider the velocity along the straight line joining the start and goal positions of the Robots as the desired velocity in RVO and sample the RVO trajectory obtained at evenly spaced intervals to match the number of time samples expected by the optimizer. We define the following set of additional configurations to test our optimizer with different types of initializations:

- **Square Antipodal:** Robots are placed on the edge of a square and are needed to reach their antipodal positions. This scenario is particularly challenging since if the robots were moving in straight lines between the start and goal positions, they would all collide together at the center of the workspace.

- **Circle:** Robots are distributed evenly along a circle and are needed to rotate by k positions. (default $k = 5$).

- **Ellipse:** Robots are placed uniformly along a straight line, and the i^{th} robot is needed to reach the position of the $(n - i)^{th}$ robot where n is the total number of robots.

Figure 3.8 shows the trajectories generated by RVO alone for 16 Robots in the Square Antipodal scenario vs trajectories generated by our optimizer with RVO initialization. We observe a significant improvement in the smoothness of the trajectories generated by our optimizer with RVO initialization compared to RVO alone. The improvement in trajectory quality can be explained as follows: the optimizer starts with an already low residual due to RVO initialization and further reduces it with iterations while smoothing the trajectories at the same time. However, in terms of computation time, the RVO initialization does not offer significant benefits compared to a straight-line initialization of trajectories.

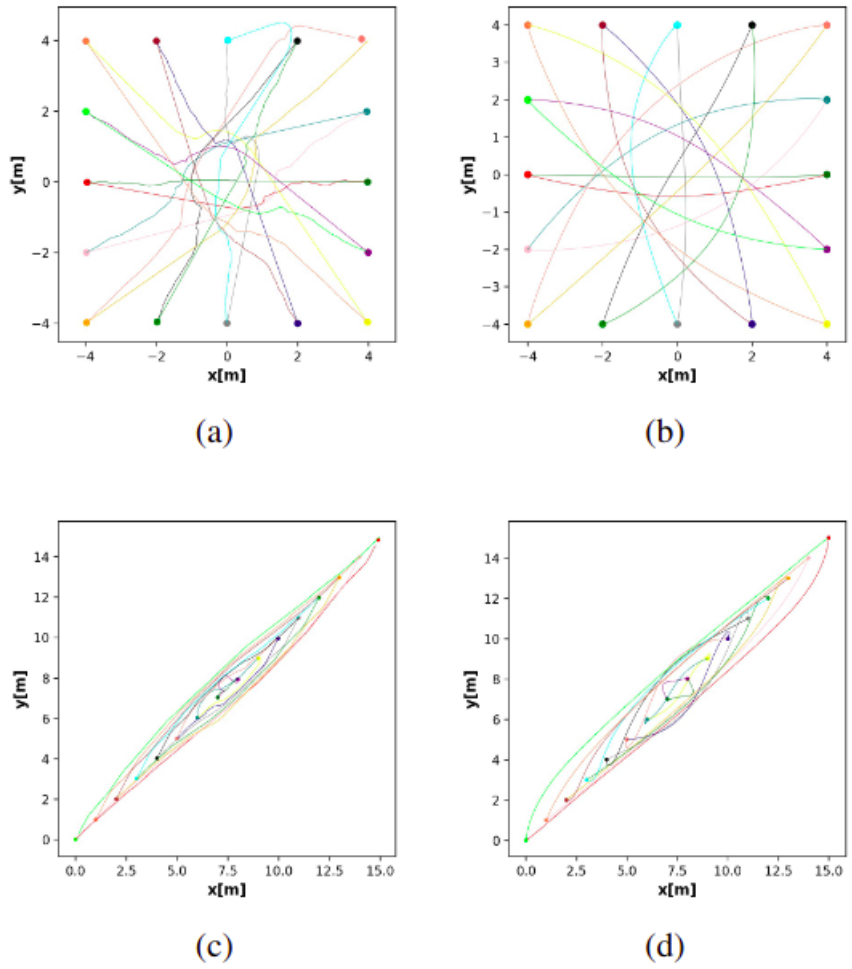


Figure 3.8: Comparison of trajectories from a) RVO alone in 16 Robots 2D Square Antipodal b) Optimizer + RVO initialization for 16 Robots 2D Square Antipodal c) RVO alone in 16 Robots 2D Ellipse b) Optimizer + RVO initialization for 16 Robots 2D Ellipse.

3.5.2 Initializations using Multi-robot Pathfinding(MAPF)[4]

Similar to RVO, we can also couple graph-search-based Multi-robot Pathfinding (MAPF) algorithms discussed in Section 2.4.3 for initializing our optimizer. For this purpose, we will use the `cbs-mapf` PyPI package that implements the high-level Conflict Based Search Algorithm based on [4] and low-level space-time A*(STA*), which is similar to normal A*(from Section 2.4.3.1 with an additional time dimension. The trajectories generated by CBS-MAPF are piecewise linear and exhibit jerks and sharp turns, so these cannot be used practically for multi-robot navigation. We scale up the start and goal coordinates depending on grid size (we use grid size = 10) as well as the robot radii and precompute

multi-robot trajectories using CBS-MAPF. Similar to the RVO trajectories, we sample points from the trajectories obtained from CBS-MAPF to calculate the optimizer parameters and initialize our optimizer.

We introduce a new scenario to test MAPF for 14 Robots arranged in a 2D grid and are needed to move to the opposite end of the grid. Figure 3.9 shows a comparison of trajectories between CBS-MAPF alone and the optimizer using CBS-MAPF for initialization. We again observe a significant improvement in the smoothness of the trajectories generated by our optimizer with CBS-MAPF initialization compared to CBS-MAPF alone. However, similar to RVO, there is no benefit to using MAPF initialization in terms of computational time.

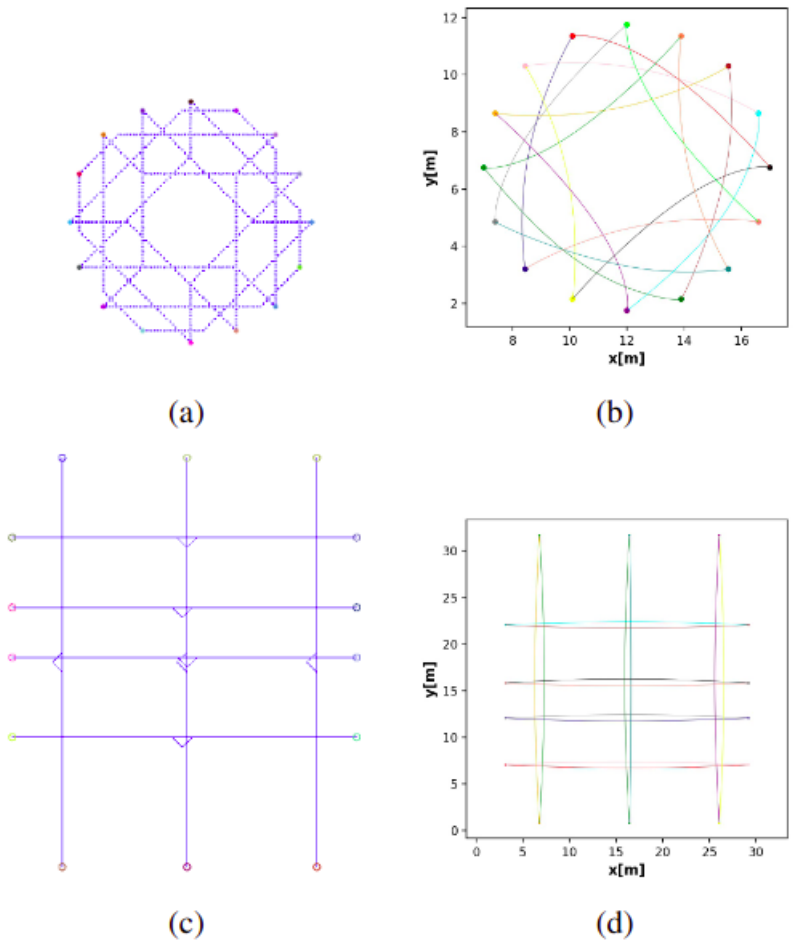


Figure 3.9: Comparison of trajectories generated by CBS-MAPF alone and our optimizer with CBS-MAPF initialization for a)-b) 16 Robots in the Circle(k=5) scenario, c)-d) 14 Robots in 2D Grid orientation.

Table 3.1: Important Symbols used in our optimizer

n_p, n_v, n_r	Planning steps, number of variables parameterizing trajectory along each motion axis, and number of robots, respectively.
a, b	Spheroid dimensions
$x_i(t), y_i(t), z_i(t)$	Position of i^{th} robot at time t.
$\overline{x}_i(t), \overline{y}_i(t), \overline{z}_i(t)$	Predicted position of j^{th} agent.
$\boldsymbol{\lambda}_i$	Lagrangian multiplier

3.6 Discussions

Joint multi-robot trajectory optimizations are generally considered intractable beyond a small number of robots. This is because the number of pair-wise collision avoidance constraints increases exponentially with the number of robots. Moreover, even the best optimization (QP) solvers show polynomial scaling with the number of constraints. We have fundamentally altered this notion through the discussion in this chapter. By employing a clever set of reformulations and parallelism offered by modern computing devices such as GPUs, we managed to compute trajectories for tens of robots in highly cluttered environments in a fraction of a second. Our formulation is simple to implement and involves computing just matrix-matrix products. Such computations can be trivially accelerated or parallelized on GPUs using off-the-shelf libraries like JAX ([22]). We benchmarked our approach against two strong state-of-the-art baselines and showed substantial improvement over them in terms of computation time and trajectory quality.

Our work has potential beyond multi-robot coordination for interaction-aware trajectory prediction. We show in Sections 3.5.1 and 3.5.2 how our multiagent optimizer may be used to improve the quality of trajectories obtained from multi-agent collision avoidance methods, such as Reciprocal Velocity Obstacle(RVO)[3] and Multi-Agent Pathfinding(MAPF)[4].

It must be noted here that we were able to approximate collision constraints and solve the multi-robot path planning problem through gradient-based optimization due to the simplicity of the kinematics governing the motion of each robot. In this case, all the robots in the multi-robot system were holonomic. However, to extend this framework for car-like vehicles (non-holonomic robots), we would have to obey more complicated kinematic equations as discussed in Section 2.1.2. When the robot kinematics becomes as complex as for manipulators, discussed in Section 2.5.1, even solving the trajectory optimization problem for a single robot may become intractable using gradient-based optimizers. This raises the need for a stochastic optimizer for robots with high-dimensional articulation, such as manipulators. In the next chapter, we will discuss how stochastic optimization can be used to generate collision-free trajectories for manipulators, and how the optimization can be decoupled into a bi-level framework, consisting of a global path planner and low-level motion planner.

Table 3.2: Comparison with current state-of-the-art [1] in terms of computation time.

Number of robots	Number of Obstacles	ours[s]	[1][s]
32	24	0.21	12.897
32	20	0.20	11.827
32	16	0.20	12.423
32	12	0.19	12.504
16	24	0.17	0.795
16	12	0.17	0.661
16	8	0.16	0.680
16	4	0.16	0.702
16	2	0.15	0.621

Table 3.3: Comparison with [2] in terms of computation time, arc-length and smoothness cost

Number of robots	Number of Obstacles	Benchmark	Computation time[s]	Arc-length[M]	Smoothness cost
16 robot	2	[2]	0.34	13.488	0.102
	2	Ours	0.15	9.999	0.048
	4	[2]	0.37	14.257	0.114
	4	Ours	0.16	11.693	0.093
	8	[2]	0.70	15.539	0.140
	8	Ours	0.16	11.118	0.089
	12	[2]	0.79	15.931	0.159
	12	Ours	0.17	11.192	0.106
	24	[2]	1.49	24.198	0.164
	24	Ours	0.17	10.249	0.069
32 robots	12	[2]	1.688	23.855	0.157
	12	Ours	0.19	22.593	0.132
	16	[2]	1.752	24.04	0.164
	16	Ours	0.20	22.303	0.122
	20	[2]	1.804	24.14	0.170
	20	Ours	0.20	23.156	0.210

Chapter 4

Stochastic Trajectory Optimization for Robot Manipulators

Manipulating a target object using a fixed-base robot manipulator presents a complex problem that requires addressing multiple constraints, such as optimizing the manipulator joint cost, finding a collision-free trajectory, and modeling the object dynamics. Previous approaches have relied on contact-rich manipulation, where the object moves predictably while attached to the manipulator, thereby avoiding the need to model the object's dynamics, which does not generalize to multiple types of end-effectors. Further, collision avoidance for manipulators in existing approaches rely on context-specific neural network-based object representation, which need to be retrained for each specific arrangement and type of obstacles and robots. In this chapter, we try to solve the research problem **T3** from Section 1.1.1. We will first discuss a stochastic trajectory optimizer, Via-point Stochastic Trajectory Optimization and adapt it for context-agnostic path planning in the joint space for commonly-used robot manipulators, such as the Franka Panda and UR5e. Next, we propose a novel framework that disentangles the non-prehensile long-horizon manipulation problem into path planning and motion planning. The planning component uses the aforementioned context-agnostic Via-Point stochastic trajectory optimizer, generating a collision-free trajectory from the start to the goal position, consisting of multiple via-points. We pair the high-level via-point planner with a low-level motion planner based on Deep Reinforcement Learning, that is gripper-agnostic.

4.1 Introduction

Object manipulation is an essential task for robots to perform that has many diverse downstream applications, for example in the service industry (manipulating kitchenware [47]), industrial setting (packaging [48]) and household (tabletop rearrangement [49]). Broadly, object manipulation can be of two types - *Prehensile*, which involves continuous contact-rich gripping of the object using the manipulator's end-effector(gripper or a hand-like structure), and *Non-Prehensile*, which involves moving the object without grasping it. Prehensile actions are often used in manufacturing, packaging, and assembly processes where the robot needs to handle and manipulate objects of various shapes and sizes. On the other hand, non-prehensile actions are typically used when the object is too fragile or too large

to be picked up by the robot's gripper. Instead, the robot uses other methods such as pushing, pulling, sliding, or rolling to move the object around. Non-prehensile manipulator actions are commonly used in applications such as warehouse automation, sorting, and material handling.

Path planning for manipulators involves finding a collision-free path for a robot arm or manipulator to follow as it moves from its current state to a desired state. This state could be a set of predefined joint angles, or end-effector positions. Any trajectory sent to a manipulator to be followed must satisfy the dynamic and kinematic constraints, such as joint angle and velocity limits, as well as the workspace bounds in which it operators. These constraints come from the motion planner, which calculates the joint velocities necessary for executing the high-level path. A few commonly-used approaches to path planning for manipulators include the following:

- *Sampling-based Methods:* These methods randomly sample the configuration space of the robot (i.e., the set of all possible positions and orientations) and attempt to find a collision-free path by connecting the samples together. Examples of sampling-based methods include Rapidly-exploring Random Trees(RRTs)[17] and Probabilistic Roadmaps (PRMs)[50].

- *Search-based Methods:* These methods search for a path through the configuration space using a search algorithm, such as A*[18] or Dijkstra's algorithm. Search-based methods can be combined with heuristics to reduce the search space and improve performance.

- *Optimization-based Methods:* These methods formulate the path planning problem as an optimization problem, where the objective is to find the shortest path that satisfies certain constraints (e.g., collision avoidance). Optimization may be performed by gradient-based updates or by sampling potential candidates from a distribution and updating the distribution itself after evaluating the cost of the candidates.

In this chapter, we will explore optimization-based methods for path planning for manipulators. The intuition behind this choice is derived from the following ideas:

- Sampling-based methods or search-based methods may take a very long time to return an end-to-end path for complex scenes, such as cluttered tabletops. Optimization-based approaches, especially long-horizon planning, can return sub-optimal end-to-end trajectories very quickly and iteratively improve on them.

- Optimization-based methods are able to handle a variety of objectives, including collision avoidance, joint cost, and kinematic limits, which would merely be heuristics for the other methods. Search-based and sampling-based methods are restricted to a discrete solution space by virtue of their design, and if our samples do not include the most optimal solution, they will return sub-optimal trajectories. Optimization-based techniques, however, can adapt along the iterations to ultimately converge to the most optimal solution.

Sections 2.5.4 and 2.5.5 provide a background of standard gradient-based, and sampling-based optimization approaches for robot manipulation, respectively. Stochastic optimization includes the speed of sampling-based planners and the optimality guarantee of optimization frameworks to generate trajectories for manipulators. We adopt an improved version of stochastic trajectory optimization from Section 2.5.5 for our purpose and incorporate it into a bi-level push planner for manipulators. The contributions of this chapter are twofold:

1. Designing a fast, collision-aware high-level global path planner using stochastic trajectory optimization for manipulators

2. Coupling this high-level planner with a low-level push planner to enable the manipulator to perform precise long-horizon non-prehensile actions, such as pushing an object on a table.

4.2 High-level Global Planning for Manipulators

In this section, we introduce the basics of a high-level path planning algorithm, Via-Point Stochastic Trajectory Optimization(VP-STO)[13], and design a suitable joint-space cost function incorporating collision avoidance. We show a few simulation results of our high-level global planner on the Universal Robots, UR5e robot arm and the Franka Emika Panda robot arm, both of whose kinematics have been discussed in Section 2.5.1.

4.2.1 Via-Point Stochastic Trajectory Optimization(VP-STO)

Via-Point Stochastic Trajectory Optimization(VP-STO)[13] fits different candidate trajectories satisfying the start and goal constraints to perform stochastic optimization over a cost function. VP-STO computes N via-points between the start and the goal by sampling them from a multivariate Gaussian distribution, and fits trajectories satisfying the kinematic constraints of the robot and the workspace passing through them. The cost corresponding to these trajectories are then evaluated, and the trajectories are ranked in their order of cost. Finally the parameters of the sampling distribution are updated using the Covariance-Matrix Adaptation Method (CMA-ES). The benefit of the CMA-ES approach over standard Cross-Entropy Methods(CEM) lies in the following factors:

- *Better handling of non-linearities*: CMA-ES is known for its ability to handle non-linear optimization problems, which are common in manipulator planning. CEM, on the other hand, tends to struggle with non-linearities, which can result in slower convergence and suboptimal solutions.

- *Adaptive step size:* CMA-ES uses an adaptive step size to adjust the search direction and step size during optimization, which helps to avoid getting stuck in local optima. CEM, on the other hand, uses a fixed step size, which can lead to premature convergence and suboptimal solutions.

- *Better exploration of search space:* CMA-ES uses a probabilistic approach to explore the search space, which helps to avoid getting stuck in local optima and to find the global optimum. CEM, on the other hand, uses a deterministic approach, which can lead to poor exploration of the search space.

The overall optimization pipeline in VP-STO is denoted in Figure 4.1. The cost function can be represented as the following:

$$min \int_0^1 q''(s)^T q''(s) ds \tag{4.1a}$$

$$\text{s.t} \qquad q(s_n) = q_n, \qquad n = 1, .., N \tag{4.1b}$$

$$q(0) = q_0, q'(0) = q'_0, q(1) = q_T, q'(1) = q'_T \tag{4.1c}$$

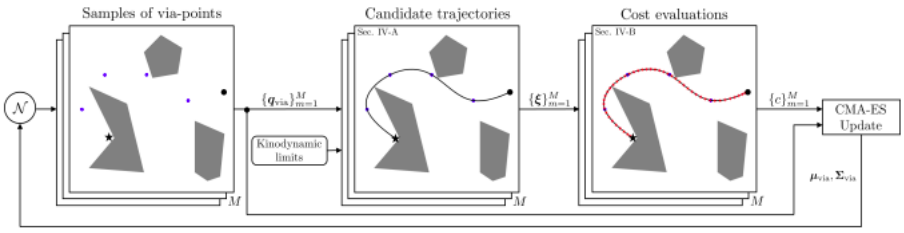


Figure 4.1: The end-to-end pipeline for Via-Point Stochastic Trajectory Optimization(VP-STO). Source: [13]

.

Here s_n denotes the scaled via point timings $s_n = t_n/T$, which are uniformly distributed between $(0, 1)$. $q(s)$ is a weighted sum of basis functions. T is the total movement duration. Now, given the via-points q_{via} and the boundary conditions q_0, v_0, q_T, v_T, the computation of the explicit continuous trajectory only depends on T. VP-STO approximates T by solving for the minimum positive duration such that the resulting velocity and acceleration limits are satisfied over a discrete set of evaluation points uniformly distributed in the continuous time-space.

For each evaluation point s_k, there exists a closed-form solution T_k such that the motion happens through q_0, q_T, q_{via}, and the robot arm reach either the velocity limit or the acceleration limit at time $t = s_k$. VP-STO then picks the most conservative duration among the K solutions for $T : T(q_{via}) = max(T_k)$ in order to make sure that the velocity and acceleration constraints are satisfied at all evaluation points. Once T is known, $q(s)$ is found using the equations for q_0, q_T, q_{via}, v_0, and v_T.

VP-STO optimizes the trajectory by minimizing a cost function that captures the tradeoff between smoothness, efficiency, and safety. Based on the design of the cost function that VP-STO minimizes, we can adapt it for a wide variety of applications. In this dissertation, we adapt the VP-STO algorithm for manipulator pushing tasks on a tabletop and pair it with a low-level push planner in Section 4.3.

4.2.2 Collision Detection

Collision detection for a robot manipulator needs to be performed on two fronts - self-collision, which involves collisions among the links of the robot itself, and environment collision, which includes collisions between the robot links and objects in the environment. Note that the contact between the robot's end-effector and the target object to be moved on the table is not considered as a collision. We have already discussed standard distance-based collision avoidance for manipulators in Section 2.3.3.1. Let us now analyze the drawbacks of these existing approaches.

As discussed in Section 2.3.3.1, one way of checking for collisions is to sample points uniformly along the robot body and compute pairwise distances among them(for self-collision) and with other objects in the environment. If the computed distance exceeds the threshold separation determined by the geometries of the robot body and the objects, the point is collision-free. The difficulty with this approach arises due to its reliance on the knowledge of the exact geometries of the robot and the other objects. This makes it hard to generalize this approach across a range of objects of different shapes and sizes on the table, as well as to robots with different dynamics. Learning implicit representations of objects also runs into the same generalizability issue, and thus the necessity of a standardized approach capable of detecting collisions agnostic to object and robot geometries becomes evident.

4.2.2.1 PyBullet Mesh Overlap:

In this chapter, we will adopt a different approach called *Mesh Overlap*, that attempts to solve this problem of collision avoidance in a simulator setting. The simulator of our concern is PyBullet, which is a physics simulation engine that can be used to simulate robots and other complex systems in 3D. To use PyBullet to load 3D robots into a simulation scene, one can follow this general sequence of steps:

1. *Define the robot model:* Start by defining the geometry, mass, and other properties of each link and joint in the robot model. This can be done using the URDF (Unified Robot Description Format) file format.

2. *Load the robot model:* Once we have defined the robot model, we can load it into PyBullet using the loadURDF function. This function takes as input the path to the URDF file and returns a unique identifier for the robot in the simulation.

3. *Visualize the robot:* we can visualize the robot in PyBullet using the render function. This function generates a 3D mesh of the robot model and displays it in a window.

4. *Simulate the robot:* We can simulate the robot's behavior by applying forces and torques to its joints using the `'setJointMotorControl2'` function. This function allows we to specify the desired position, velocity, or torque for each joint in the robot model.

5. *Collect data:* During the simulation, we can collect data on the robot's position, velocity, and other properties using the `'getBasePositionAndOrientation'` and `'getJointState'` functions. This data can be used for analysis and control purposes.

As discussed in point 3, the simulator contains the robot model as a set of geometric meshes. Similarly, for objects in the environment, the simulator loads geometric meshes into the scene. An overlap between two object meshes indicates a collision between them. We can, thus, detect collisions using PyBullet's inbuilt collision detection feature based on the overlap between the meshes of the robot and the obstacles or between the meshes of robot links. In fact, PyBullet is equipped with the feature to detect collision by detecting mesh overlap and penetration depth between meshes through the function `{getCollisionFn}`. An example of mesh overlap and subsequent collision detection in PyBullet is depicted in Figure 4.2.

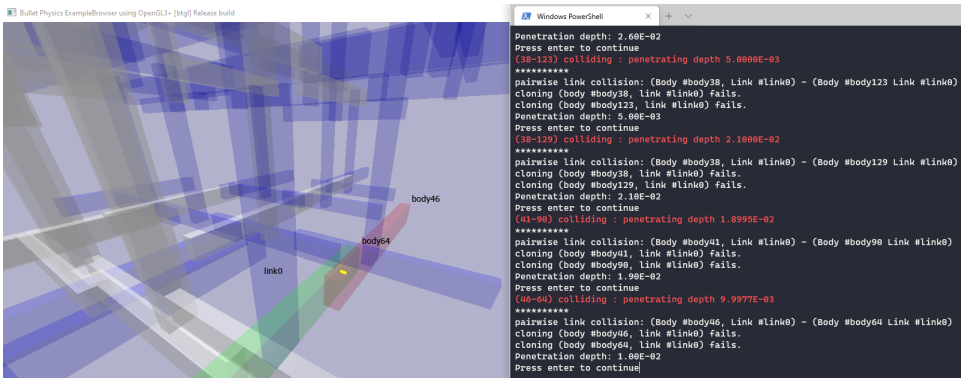


Figure 4.2: Collision Detection through Mesh Overlaps in the PyBullet simulator. Source: `https://github.com/yijiangh/pybullet_planning_tutorials`

To explain PyBullet's inbuilt collision detection through code, let us analyze the standard use of the `getCollisionFn` as follows:

```
collision_fn = get_collision_fn(robot, ik_joints, obstacles,
self_collisions=True, disabled_collisions=self_collision_links)
print (collision_fn(joint_poses, diagnosis = True))
```

We send the robot model, list of movable joints, and list of obstacle models to the `getCollisionFn`, along with a few Boolean flags. `selfCollisions` indicates whether to detect self-collisions among the robot link meshes, `disabledCollisionFn` relaxes the collision detection between mesh pairs that are not counted as a collision, for example, between the end-effector and the target object being moved. `getCollisionFn` returns a Boolean flag for a given snapshot of joint poses, indicating whether it has been able to detect a collision or not. The `diagnosis` flag indicates whether we want to visualize the detected collision in the PyBullet GUI or not.

4.2.3 Joint-Space Path Planning

The design of the cost function for VP-STO determines its application to the task at hand. For non-planar end-effector motion, such as grasping and pick-and-place operations, we perform global path-planning in the joint angle space. Recall from Section 2.5.1 that the Franka Emika Panda robot has 7 movable joints, while the UR5e robot has 6 movable joints. To represent the joint angle movements and perform path-planning in the joint space, therefore, we would need to plan in 7-D for the Franka Panda manipulator and in 6-D for the UR5e manipulator. Joint-space planning allows us to also introduce joint limits as constraints into the optimization problem based on the robot dynamics. Note that we are allowed to design discontinuous non-differentiable cost functions since VP-STO never computes the gradient of the cost function but relies on stochastic optimization. We design a cost function for VP-STO as a sum of the following terms:

1. **Cost Limits**: This is defined as the frequency of violation of joint limits while executing a joint-space trajectory.

2. **Cost curvature**: Curvature cost aims to shorten the arc length (refer to Section 2.3.4 for arc-length definition) of the joint-space trajectory to ensure short trajectories.

3. **Joint Cost**: Joint cost is defined as the norm of the difference in joint angles over successive timestamps. In essence, joint cost captures the effort expended by the robot to perform some task by changing its joint angles.

4. **Cost Collision**: This is a discrete cost, consisting of a very high-cost value C if a collision is detected, or 0 otherwise.

$$\text{Collision Cost} = \begin{cases} C & \text{if collision = True} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

4.2.4 Simulation Results for Joint-Space Path Planning

We test the VP-STO-based global joint-space path planner using the cost function defined in Section 4.2.3 in PyBullet for a few scenarios involving one or two obstacles of varied sizes. The results of a few test simulations can be found at: `https://www.dropbox.com/scl/fo/t1dy47cgz2rvmedj9a87o/h?dl=0&rlkey=ou4losxv0i9lawc2wffnk60vl`. Figure 4.3 shows an example trajectory from our simulator runs for the Universal Robots UR5e robot arm.

4.2.5 Path Planning for Pushing Objects on a Table

For the planar motion of the manipulator's end-effector, such as pushing an object along a table, we can discard the high-dimensional joint space path planning in favor of a simpler Cartesian-space motion of the end-effector. Thus the path planning for the end-effector can be performed in the Cartesian

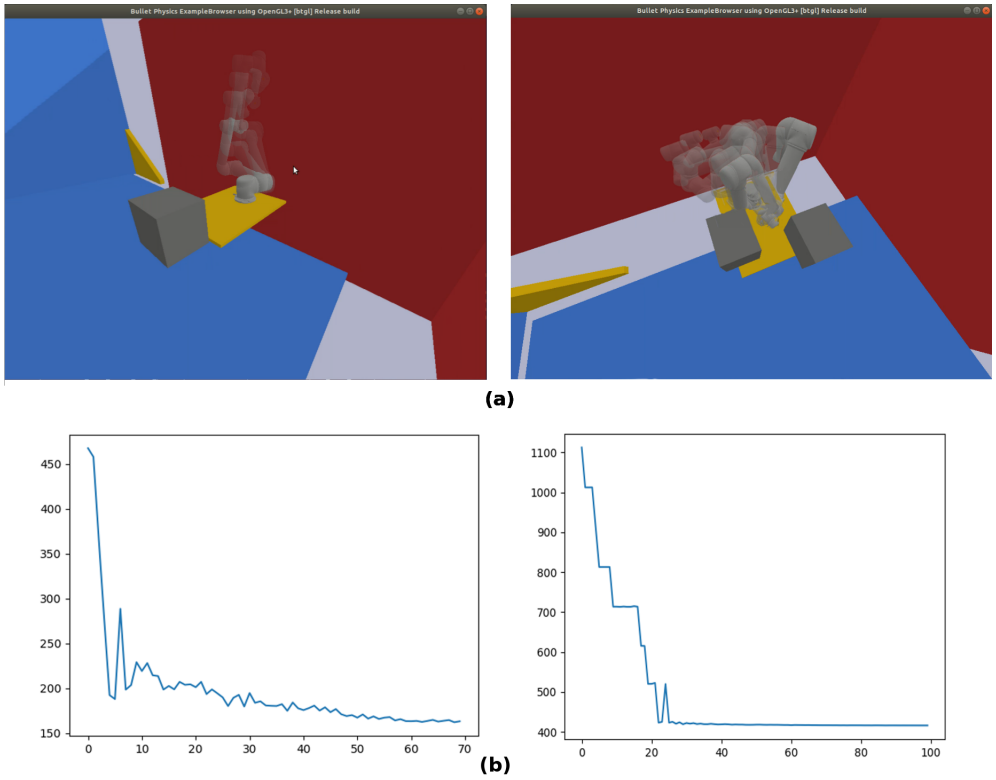


Figure 4.3: (a) A few examples of trajectories obtained after joint-space trajectory optimization using VP-STO simulated in PyBullet. (b) The associated cost function plots during the optimization iterations. VP-STO is able to minimize the cost function, which can be denoted by the reducing trend in the cost plots.

end-effector space, which would be 2-D in the case of planar tabletop motion. The end-effector can be thought of as a circular holonomic robot in 2-D of diameter equivalent to the separation between the fingers of the gripper plus the finger widths. This approximation makes it possible to use standard holonomic robot path-planning algorithms, as well as stochastic optimizers like VP-STO, to plan the end-effector trajectory. We can further simplify this by planning 2-D trajectories for the center of the object being pushed on the table instead of directly the end-effector, particularly in cases where only brief contacts or pushes by striking are allowed between the end-effector and the object. In such cases, it is not enough for the end-effector to reach the desired goal position. Rather, the object should be able to reach the goal position, and an efficient push planner should be designed to determine the controls necessary for the manipulator to push the object along the planned trajectory. An example of a 2-D trajectory for an object being pushed on a table by a manipulator is shown in Figure 4.4.

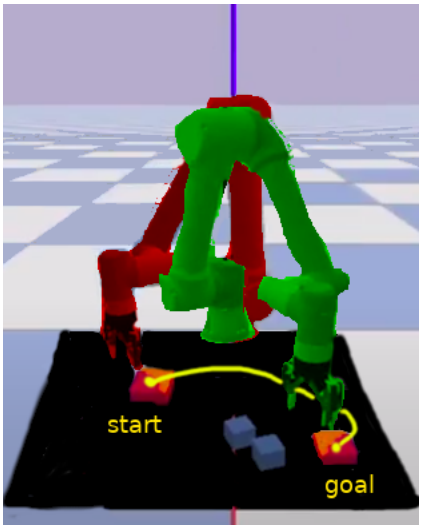


Figure 4.4: 2-D Trajectory of an object being pushed by a Manipulator. Here the red-shaded manipulator denotes its start pose, and the green-shaded manipulator denotes its final pose. The blue cubes indicate obstacles (collision objects) on the table. The end-effector motion happens along the plane of the black table as the object moves from the start to the goal position. The yellow line denotes the trajectory of the center of the object being pushed.

4.2.5.1 Analysis of Joint Costs associated with Cartesian-space trajectories for pushing objects

For manipulators, in addition to the performance metrics discussed in Section 2.3.4, joint cost becomes a very important metric for benchmark comparison. Joint cost is defined as the norm of the difference in joint angles over successive timestamps. In essence, joint cost captures the effort expended by the robot to perform some task by changing its joint angles, and therefore, the goal of an efficient trajectory optimizer would be to minimize the joint cost incurred by the robot arm to complete a given task. For joint-space path planning, it is easy to incorporate the differences between successive joint angles for each joint-space trajectory. However, if we are planning in the Euclidean end-effector space, or in the space of the center of the object being pushed on the table, minimizing just the Euclidean length of the trajectory may not be sufficient; it is important to check for the joint effort to execute an end-effector trajectory or push an object along a planned trajectory. Table 4.1 demonstrates that the shortest object-center trajectory in terms of Euclidean length, i.e. straight-line trajectory, does not always guarantee the least joint cost; in fact, the trajectory returned by VP-STO optimizing over joint-cost is more optimal in terms of joint effort for the same number of waypoints. To compute joint cost, we need to execute the planned object-center trajectory using a push planner and obtain the joint angles. For real robots, we can use manipulator-specific analytical Inverse Kinematics(IK) solvers[51] to convert end-effector positions to joint angles.

Table 4.1: Comparison of Joint Costs associated with different types of object-center trajectories using our push planner from Section 4.3.4.1.

Start Position(m,m)	End Position(m,m)	Trajectory Type	Num WayPoints	Joint Cost
(0.35, -0.05)	(0.5, 0.1)	Straight line	8	48.0866966147044
(0.35, -0.05)	(0.5, 0.1)	2-piecewise straight lines	8	53.2750370301563
(0.35, -0.05)	(0.5, 0.1)	VP-STO	8	39.5423031834928
(0.4, -0.15)	(0.63, 0.05)	Straight line	8	83.0072021138549
(0.4, -0.15)	(0.63, 0.05)	2-piecewise straight lines	8	113.146020007934
(0.4, -0.15)	(0.63, 0.05)	3-piecewise straight lines	8	115.673830244566
(0.4, -0.15)	(0.63, 0.05)	4-piecewise straight lines	8	120.5008919817
(0.4, -0.15)	(0.63, 0.05)	VP-STO	8	62.1512468250956
(0.5, 0.1)	(0.65, -0.05)	Straight line	18	88.1713855670522
(0.5, 0.1)	(0.65, -0.05)	2-piecewise straight lines	18	100.889302118451
(0.5, 0.1)	(0.65, -0.05)	VP-STO	18	77.346373

4.3 Bi-Level Optimization for Non-Prehensile Actions

4.3.1 Introduction

Non-prehensile manipulation has typically received less attention in the literature compared to grasp manipulation. A standard mechanism to robustly push an object from start to goal position while avoiding colliding with objects on the table does not exist.

A recent work [52] tackles this problem but assumes contact-rich manipulation in which the target object is attached to the manipulator. This reduces the complexity of the problem by assuming predictable dynamics - the object moves along with the manipulator. Contact-rich manipulator, however, assumes a certain type of gripper that may not be feasible for a given use case. We concentrate on non-prehensile manipulation in an end-effector agnostic manner that assumes the object dynamics to be independent of the end-effector dynamics. The primary challenge in developing a policy to execute such an action is the stochastic outcome of pushing an object: in the absence of privileged information like friction, the weight of the target object, it is impossible to accurately predict the dynamics of the object on a push action. To tackle this, several works have tried to explicitly model the outcome by training a deep neural network on a large amount of push-outcome pairs collected via simulation [53, 54, 55]. However, such explicit modeling results in low generalization, making multi-step long-horizon planning of pushing an object from start to goal challenging.

In recent years, deep reinforcement learning (RL) has shown promising results in various robotics applications. RL algorithms learn to perform a task by maximizing a reward signal, which can be a scalar value that reflects the task's success or failure; or a dense reward indicating the distance from the goal.

However, RL algorithms have high sample complexity [56], and designing the perfect reward function is tricky, limiting their application in long-horizon scenarios [57] needing complex reward design.

Moreover, RL algorithms trained for long-horizon tasks are hard to generalize across a wide variety of scene configurations and out-of-distribution object shapes and sizes.

In this work, we tackle the problem of object rearrangement using non-prehensile actions by disentangling the space of control and planning. The path planning module generates a feasible trajectory for the robot's end-effector, while the low-level RL motion planner learns to plan the robot's controls to achieve the desired task. In our framework, RL acts as an efficient mechanism for understanding the dynamics of an object on different push actions. Here, the RL motion planner is trained using a simple objective to push the object quickly to a given goal location.

In this setting, the low-level motion planner is unaware of the collision objects and is dependent on a high-level planning module to obtain an optimal collision-free trajectory. At the same time, the high-level planning module predicts the most optimal trajectory for the given RL motion planner. The framework is trained using bi-level optimization: the high-level planning module is optimized on the cost of execution of the low-level motion planner.

A global RL policy solves the task of manipulating an object from start to goal position end-to-end using a single policy. Such a global policy simultaneously solves the control task (understanding the dynamics of the system) and the planning task (a collision-free optimal trajectory from start to goal). Compared to training a global RL model for non-prehensile object manipulation, our framework, based on a bi-level optimization objective can have several advantages:

1. **Better task-specific performance:** Non-prehensile object manipulation tasks can be highly varied and complex, and a single global RL model may not be able to handle all tasks equally well. In contrast, a bi-level optimization approach can generate task-specific plans that are optimized for each individual task, resulting in better performance.

2. **Improved sample efficiency:** Non-prehensile object manipulation tasks typically require a large number of samples to train an RL model effectively. A bi-level optimization approach can reduce the number of samples required by using the high-level planning module to generate a task-specific collision-free plan, which can simplify the RL objective to only push to a goal location.

3. **Better interpretability:** A bi-level optimization approach separates the generation of the task-specific plan and the low-level actions, making it easier to understand how the system operates and diagnose issues. In contrast, a global RL model can be more opaque and difficult to interpret.

4. **Ability to handle constraints:** Non-prehensile object manipulation tasks often have constraints, such as avoiding collisions or maintaining balance. A bi-level optimization approach can incorporate these constraints into the high-level planning module and use them to guide the low-level actions.

In contrast, a global RL model may struggle to handle constraints effectively, as designing an appropriate reward can be tricky.

4.3.2 Related Work

Push-based non-prehensile manipulation can be divided into push-to-grasp, such as pushing objects in clutter to make them graspable [53, 54] or sliding an object to the edge of the table [58, 59] and push-to-goal to push an object from a start to a goal position [52, 60, 61, 55]. The latter line of work can further be classified as contact-rich manipulation either by sliding by the top [62, 63, 60, 52] or by the side [61].

We aim to tackle push-to-goal through push-by-striking manipulation. Push-by-striking loses the contact-rich assumption and thus disentangles the dynamics of the object from the manipulator dynamics removing the constraints in the type of end-effector - as the end-effector strikes an object, the outcome of the object state is independent of the end-effector state.

RL has been applied in various robotics applications, including grasping [64, 65, 66], manipulation [67, 68], and locomotion [69, 70]. Recently there has been a lot of work [71, 72, 73] on combining RL with classic controllers and primitives to improve their performance. RL-based methods have also shown promising results in performing non-prehensile object manipulation tasks. For instance, Tan et al.[74] proposed a hierarchical RL method for non-prehensile object manipulation, where the high-level policy generates a sequence of subtasks, and the low-level policy learns to execute each subtask. Zhang et al.[75] proposed a hierarchical RL framework that combines the advantages of both model-based and model-free methods for non-prehensile object manipulation.

Path planning algorithms are also widely used in robotics applications, including object manipulation =[76, 77, 78]. Path planning algorithms generate a feasible trajectory for the robot to perform the task. Various path-planning algorithms have been proposed, including sampling-based methods[76] and optimization-based methods[77]. While other path-planning approaches would build a path without taking into account the capabilities of the lower-level motion planner, our trajectory optimization model samples optimal trajectories by considering the capability of our low-level RL motion planner. Our neural network is aware of the limitations and capabilities of the lower-level RL motion planner.

4.3.3 Task Specification

We consider the task of pushing a target object from an initial pose to a goal pose on a tabletop scene with multiple movable collision objects. The environment consists of a planar surface with dimensions of 0.8m $\times$ 0.8m, upon which up to four movable objects are randomly placed. The target object is a rectangular block with dimensions 0.05m $\times$ 0.05m $\times$ 0.1m, and the robot manipulator is a Franka-Panda arm with a two-fingered gripper; however, our framework is independent of the type of gripper and can be extended to any end-effector setting (point contact, area contact, contact-rich, etc.). The manipulator

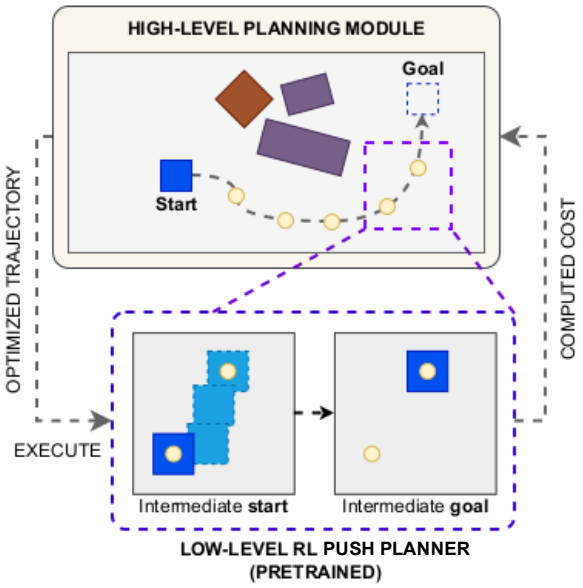


Figure 4.5: Our bi-level optimization framework solves the task of non-prehensile object manipulation in a cluttered tabletop rearrangement scene. We make use of a low-level RL motion planner that is trained to reach a short goal. The high-level planning module predicts a set of waypoints that is most optimal for the low-level RL motion planner to execute. The purple and brown objects indicate obstacles (collision objects); dark-blue squares indicate the target object, and the light-blue squares indicate the trajectory taken by the RL motion planner between a set of intermediate waypoints predicted by the high-level planning module.

interacts with the environment by pushing the target object, using the tip of the arm as the pushing point. The goal location is specified as the pose of the target object on the given tabletop scene.

4.3.4 Proposed Framework

Our proposed approach involves two modules: a path-planning module and a low-level RL motion planner. The path planning module generates an optimal collision-free trajectory for the robot's end-effector, while the low-level RL motion planner plans the robot's controls to accomplish the task objectives. Fig. 4.5 illustrates the proposed approach.

4.3.4.1 Low-level RL Push Planner

Our RL motion planner learns to simply push a target object from a start to a goal location on a tabletop scene. Our push planner is trained without any obstacles on the table and is optimized to approach the goal location quickly. Our state space consists of the set of poses taken by the target object,

which is being pushed by the RL motion planner. The action space is defined as a set of 8 discrete equidistant points on the perimeter of the target object and 8 different push angles at each of these 8 points. The objective of this RL push planner is to learn a policy that maximizes the expected cumulative reward, where the reward is the dense negative distance from the current pose to the goal location. The distance metric is the Euclidean distance between the centroid of the target object and the goal location.

Formally, the state space is represented by $S \in \mathbb{R}^6$, where each element of the state vector represents the position and orientation of the target object. The action space is represented by $A \in \mathbb{R}^{16}$, where each action is a combination of a point on the object and a push angle. Let $s_t \in S$, and $a_t \in A$ denote the state and action at time step t, respectively. The state transition is deterministic, and the next state s_{t+1} is computed as the result of applying the push action to the current state s_t.

The reward function is defined as

$$r(s_t, a_t) = -\gamma || \ \mathcal{C}(s_t) - \mathcal{G} \ ||_2 \tag{4.3}$$

where γ is a discount factor, $\mathcal{C}(s_t)$ is the centroid of the target object in the current state, and $\mathcal{G}$ is the goal location. The negative distance metric is used to encourage the push planner to minimize the distance to the goal location. The discount factor is used to balance the trade-off between short-term and long-term rewards.

The RL motion planner learns a policy $\pi(s_t)$ that maps states to actions by maximizing the expected cumulative reward. The optimal policy is obtained by solving the Bellman equation, which is given by:

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) \\ &+ \ \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \end{aligned} \tag{4.4}$$

where $Q(s_t, a_t)$ is the state-action value function, $P(s_{t+1}|s_t, a_t)$ is the transition probability, and $\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ is the maximum expected future reward. The policy is then derived from the optimal value function as:

$$\pi(s_t) = \arg\max_{a_t} Q(s_t, a_t) \tag{4.5}$$

To learn the optimal policy, we employ a deep Q-learning algorithm that uses a neural network to approximate the value function. The network takes the state as input and outputs the value for each action in the action space. We use the Adam optimizer to minimize the mean squared error between the predicted and target Q-values.

4.3.4.2 High-Level Path Planning Module

The high-level planner generates a feasible trajectory for the object being pushed by the manipulator, taking as input the initial and final configurations of the object and producing a sequence of intermediate via points that the low-level motion planner must push it along. In our case, it is important to plan the high-level trajectory for the object instead of the manipulator itself, as is done in the case of contact-rich

manipulation. This is because the behavior of the manipulator and the object being pushed are different. Pushing in a contact-impoverished manipulation task, the robot's end-effector only contacts the object briefly to give it a push, and the object continues to move without being directly contacted by the manipulator. Due to this, it is not straightforward to predict the object's trajectory based solely on the manipulator's motion, making it more challenging to plan a trajectory directly for the manipulator. It is noteworthy that planning the high-level trajectory for the object, as opposed to the manipulator, provides a robust and gripper-agnostic algorithm that accounts for the mismatches between the manipulator's motion and the object's movement due to striking.

The predicted via points are collision-free and ensure that the object can be pushed toward the goal location without encountering any obstacles. For high-level planning, we use the VP-STO algorithm, a stochastic sampling-based optimization method. VP-STO evaluates candidate trajectories by estimating the per-iteration cost function on a simulator running the low-level RL motion planner, which effectively captures the manipulator and environment dynamics. The cost function incorporates joint cost, collision cost, and boundary cost, as explained in the following section.

4.3.5 Designing the bi-level optimization objective

As can be seen in Fig. 4.5, the high-level planning module predicts an optimal trajectory for the end-effector. In other words, the predicted trajectory, $\mathcal{T}$, comprising of a set of via-points, $\mathcal{V}$, is optimal in terms of the manipulator's joint cost, $\mathcal{J}$, as it tries to execute the trajectory.

To obtain the optimal trajectory for a given scene made of multiple obstacles, VP-STO optimizes the following cost objective, $\mathbb{J}$ computed as:

$$\mathbb{J}(\pi, \mathcal{V}) = \alpha\mathcal{J} + \beta\mathcal{X} + \gamma\mathcal{B} \tag{4.6}$$

where π is our low-level RL policy as defined in Sec. 4.3.4.1. $\mathcal{X}$ and $\mathcal{B}$ are the collision and out-of-boundaries cost, respectively. α, β, and γ are the scaling parameters to normalize the three different metrics. $\mathcal{J}$ is calculated based on the first-order change in the manipulator's joint angles, while the collision cost, $\mathcal{X}$, reflects the change in obstacle positions in the scene. The out-of-boundary cost, $\mathcal{B}$, is calculated as the frequency of workspace boundary violations. Note that the joint angles are obtained after simulating the manipulator's movement using the low-level RL motion planner in PyBullet. For each of the via-point $\{v_i\}_{i=1}^{\mathcal{K}-1} \in \mathcal{V}$, the RL motion planner is executed to push the object from v_i to v_{i+1}; here $\mathcal{K}$ is the number of via-points predicted by the planning module. The total joint cost is then computed as:

$$\mathcal{J} = \sum_{i=1}^{\mathcal{K}-1} j(v_i, v_{i+1}) \tag{4.7}$$

where j indicates the local cost of moving the target object between the intermediate via points. Furthermore, our approach mainly focuses on the pushing behavior of the manipulator on the plane of the table; collisions can be assumed to be marked by positional displacements of obstacles according to the task specification in Section 4.3.3.

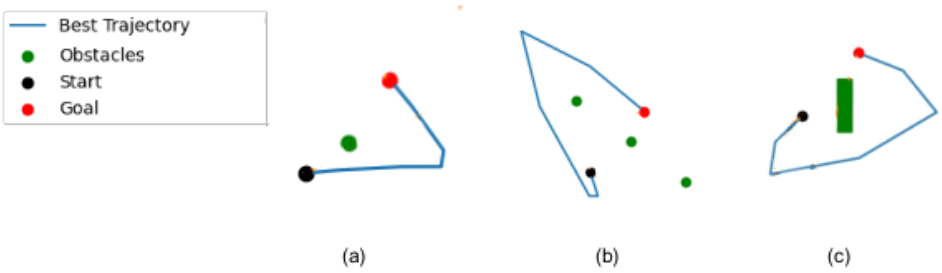


Figure 4.6: Trajectories predicted by our framework. Our framework successfully avoids trajectories and is conservative when choosing a path. Note that the positions shown here are placeholders to indicate the locations of the centers of various objects. (a) shows a trajectory with a simple case of a single collision object, and (b) shows a more complicated case when there are multiple collision objects. The obstacle sizes are so big that it is not possible for the manipulator to go between them, so it takes an alternate route around the obstacles. (c) shows a case with an elongated obstacle. Even though the trajectories may not look the most optimal in terms of Euclidean distance, these trajectories are near-optimal with respect to joint and collision costs. Refer to Table 4.1 for the analysis of joint costs vs Euclidean distance for 2-D trajectories.

4.3.6 Simulation Results

We test our bi-level optimizer on a few tabletop scenarios involving one or more obstacles of varied shapes and sizes in the PyBullet simulator. Figure 4.6 shows the trajectories predicted by our bi-level approach while avoiding collisions and minimizing joint cost within the workspace bounds. Figure 4.7 indicates the predicted trajectory when run in the PyBullet simulator using our low-level push planner. Additional simulation results on other test cases can be found at `https://www.dropbox.com/sh/mf8j0kq4mv53wct/AABVdITIHCfNBtqHMphnJcSia?dl=0`.

4.4 Discussions

In this chapter, we adapted the Via-point-based Stochastic Trajectory Optimization(VP-STO) trajectory optimizer for manipulator path planning tasks, using PyBullet's mesh overlap technique to detect collisions. We presented a few simulation results obtained from joint-space trajectory planning using VP-STO. We then proposed a framework for disentangling non-prehensile long-horizon manipulation into path planning and motion control. Our approach predicts a collision-free trajectory using VP-STO and simplifies the motion planning component's task by reducing it to moving the object toward the next via point. In the future, one can reduce the planning time for our offline bi-level optimizer by curating a dataset of start-goal pairs and their optimal trajectories. A neural network can be trained to

Figure 4.7: (a) Trajectories predicted by our framework simulated in PyBullet with our low-level RL push planner. The blue boxes on the black table indicate collision objects(obstacles), and (b) The top-view of the object motion as a result of executing the trajectory.

learn these configuration-to-trajectory mappings and can be used in real-time during inference to quickly obtain optimal manipulator trajectories. Further, more ablation studies can be performed by replacing the optimizer and the push planner in our bi-level optimization framework. Overall, our proposed framework is a promising step towards achieving more efficient and flexible non-prehensile manipulation.

Chapter 5

Conclusions

This dissertation discusses various optimization techniques for the high-dimensional trajectory optimization of robots. We tackle two complex robot systems - multi-robot systems with a large number of robots planning start-to-goal collision-free trajectories and robot manipulators operating in n-dimensional joint spaces formed by n joints. By suitable design of objective functions, choice of optimization paradigm, and mathematical approximations, our proposed path-planning methods aim to solve otherwise computationally intractable trajectory optimization problems. We present two major contributions - a GPU-accelerated distributed multi-agent trajectory optimizer and a stochastic trajectory optimizer for robot manipulators.

For the GPU-accelerated multi-agent trajectory optimizer, we leverage parallelism offered by GPUs and mathematical reformulations to convert a complex Quadratic Programming(QP) optimization problem into a simple set of matrix-matrix products. We have outperformed existing state-of-the-art algorithms in various multi-robot planning scenarios with varied numbers of robots, obstacles, and their configurations. Beyond just computational acceleration, we also achieve comparable trajectory quality against our benchmarks and show how our optimizer achieves an improving performance gap with respect to them as the complexity of the planning task increases. [1]

For manipulator trajectory generation, we adapt the existing Via-Point based Stochastic Trajectory Optimization(VP-STO) method with a suitably-designed cost function to plan collision-free paths in the joint space. We leverage PyBullet's mesh overlap-based collision detection to design our collision cost function. We demonstrate the efficiency of our approach through simulations in PyBullet for different manipulators such as UR5e and Franka Emika Panda. We then couple this high-level path planner with a low-level Deep Reinforcement Learning(RL) based push planner to solve the non-prehensile task of pushing an object on a 2-D tabletop. We present results obtained from the bi-level trajectory optimizer for different shapes and numbers of obstacles on the table.

Beyond computational speed-up, we demonstrate in Section 3.5.1 and 3.5.2 a few additional applications of our multi-agent optimizer, in improving the quality of trajectories given by multi-agent

[1]The software package for our GPU-accelerated distributed multi-robot optimizer has been made publicly available at `https://github.com/susiejojo/distributed_GPU_multiagent_trajopt`.

collision-avoidance methods such as Reciprocal Velocity Obstacle(RVO) and Multi-agent Pathfinding(MAPF) Methods. This shows that our algorithm, when coupled with other planning algorithms as initializations, can improve the quality of trajectories planned by them. Beyond holonomic robots, our algorithm can also be extended to non-holonomic robot systems and even to high-dimensional robots such as manipulators. Similarly, our bi-level trajectory optimization algorithm for manipulators can be used as a teacher to train artificial neural networks to learn optimal paths from a given start position to an intended goal position. This approach could ultimately make the goal of real-time path planning for robot manipulators a reality.

In the entirety, we believe that the contributions of this dissertation will help solve path-planning problems for a wide variety of complex robot systems and will serve as inspiration to enable robots to solve tasks ranging from simplifying our daily household work to industrial labor and defense applications.

Appendix A

Distributed GPU-accelerated Multi-Agent Joint Trajectory Optimizer: JAX NumPy and Initializations

A.1 JAX NumPy: Usage Tutorial

JAX is a Python library developed by Google that provides an easy and efficient way to perform numerical computations, particularly those involving large matrices and tensors, on a GPU. JAX uses a technique called *Just-In-Time (JIT) compilation*, which compiles the Python code into optimized machine code that can run on the GPU. This allows JAX to take advantage of the parallelism offered by the GPU, making computations much faster than if they were performed on the CPU.

One of the key features of JAX is its ability to automatically differentiate functions. JAX also provides a set of linear algebra operations that are optimized for GPU acceleration, including matrix multiplication, matrix-vector multiplication, and matrix inversion. To perform matrix operations using JAX, one first needs to create arrays or tensors that contain the matrix data and then use the JAX linear algebra functions to perform matrix operations on these arrays.

The usage of the JAX NumPy library involves a minor change in standard NumPy Python code by replacing most NumPy function calls with JAX NumPy calls. For example, to define a 2-D tensor of zeroes of shape $(5, 3)$ in JAX NumPy, we would do the following:

```
import jax.numpy as jnp
new_arr = jnp.zeros((5,3))
```

To use the JIT feature of JAX NumPy, we can think of JIT as a decorator that accepts functions operating on JAX arrays and their arguments as static arguments. For example, if we have a function 'addJAX' defined over JAX NumPy arrays, we would tell the Python interpreter to use JIT compilation for this function as follows:

```
from jax import jit
add_jit = jit(addJAX)
```

Table A.1: Comparison of multi-agent path planning times using different mathematical libraries

Number of Robots	Planning time(in sec)	Library
4	0.0050	NumPy
8	0.0049	NumPy
16	0.0046	CuPy
32	0.0046	JAX NumPy
32	0.267	NumPy
32	0.0049	CuPy
64	0.0047	JAX NumPy

JAX NumPy does not allow tensor slicing or direct indexing but rather makes use of the C++ style 'at' function to access tensor elements by index. In addition, JAX tensors are immutable, i.e. they cannot be edited in place. Please refer to the official documentation of JAX NumPy for further details on JAX methods at `https://jax.readthedocs.io/en/latest/jax.numpy.html`

A.2 Comparison of different off-the-shelf GPU-based tensor manipulation libraries

For the centralized joint trajectory optimization problem for multiple holonomic robots, we implement the path planning algorithm using three different off-the-shelf open-source Python libraries - vanilla NumPy, JAX NumPy, and CuPy. The latter two libraries use accelerated tensor computations to achieve computational speed-up in computing large tensor operations, such as matrix multiplications, inverses, and so on. As discussed in Appendix A.1, JAX NumPy leverages GPU acceleration for CUDA-based tensor computations, and so does CuPy. Table A.1 compares the time taken to plan start-to-goal offline trajectories for a varying number of robots using the same core centralized trajectory optimization approach.

We observe from Table A.1 that compared to NumPy, JAX NumPy achieves a computational acceleration of around 60 times over vanilla NumPy by leveraging GPU computations. Further, the planning time for 32 as well as 64 robots are similar, indicating that the JAX-based implementation of the algorithm remains nearly constant regardless of the increase in the number of robots, or, in other words, an increase in the size of the matrices used in the optimization process. For CuPy, we observe a similar trend in the planning time, where between 16 and 32 robots, the computation time remains nearly constant as well.

Related Publications

1. Guhathakurta, D., Rastgar, F., Sharma, M. A., Krishna, K. M., & Singh, A. K. (2022). Fast Joint Multi-Robot Trajectory Optimization by GPU Accelerated Batch Solution of Distributed Sub-Problems. *Frontiers in Robotics and AI, 9*. doi:10.3389/frobt.2022.890385

2. Guhathakurta, D., Rastgar, F., Sharma, M. A., Krishna, M., & Singh, A. K. (2022). GPU Acceleration of Joint Multi-Agent Trajectory Optimization. Poster presented at IROS 2022 workshop on *Decision Making in Multi-Agent Systems(DMMAS)*, Kyoto, Japan.

Bibliography

[1] Jungwon Park, Junha Kim, Inkyu Jang, and H. Jin Kim. Efficient multi-agent trajectory planning with feasibility guarantee using relative bernstein polynomial, 2020.

[2] Fatemeh Rastgar, Houman Masnavi, Jatan Shrestha, Karl Kruusamäe, Alvo Aabloo, and Arun Kumar Singh. Gpu accelerated convex approximations for fast multi-agent trajectory optimization. *IEEE Robotics and Automation Letters*, 6(2):3303–3310, 2021.

[3] Jur van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. pages 1928–1935, 05 2008.

[4] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 02 2015.

[5] Thi Thoa Mac, Cosmin Copot, Duc Trung Tran, and Robin De Keyser. Heuristic approaches in robot path planning: A survey. *Robotics and Autonomous Systems*, 86:13–28, 2016.

[6] Yufan Chen, Mark Cutler, and Jonathan P How. Decoupled multiagent path planning via incremental sequential convex programming. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5954–5961. IEEE, 2015.

[7] Yufan Chen, Mark Cutler, and Jonathan P How. Decoupled multiagent path planning via incremental sequential convex programming. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5954–5961. IEEE, 2015.

[8] Carlos E Luis, Marijan Vukosavljev, and Angela P Schoellig. Online trajectory generation with distributed model predictive control for multi-robot motion planning. *IEEE Robotics and Automation Letters*, 5(2):604–611, 2020.

[9] Nathan Ratliff, Matt Zucker, J. Andrew Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *2009 IEEE International Conference on Robotics and Automation*, pages 489–494, 2009.

[10] John Schulman, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, and Pieter Abbeel. Trajopt.

[11] Mohak Bhardwaj, Balakumar Sundaralingam, Arsalan Mousavian, Nathan Ratliff, Dieter Fox, Fabio Ramos, and Byron Boots. Storm: An integrated framework for fast joint-space model-predictive control for reactive manipulation. 2021.

[12] Cristina Pinneri, Shambhuraj Sawant, Sebastian Blaes, Jan Achterhold, Joerg Stueckler, Michal Rolinek, and Georg Martius. Sample-efficient cross-entropy method for real-time planning. In *Conference on Robot Learning 2020*, 2020.

[13] Julius Jankowski, Lara Brudermüller, Nick Hawes, and Sylvain Calinon. Vp-sto: Via-point-based stochastic trajectory optimization for reactive robot behavior, 2022.

[14] Mehmet Acikgoz and Serkan Araci. On the generating function for bernstein polynomials. *AIP Conference Proceedings*, 1281:1141–1143, 09 2010.

[15] Arun Kumar Singh and K. Madhava Krishna. Reactive collision avoidance for multiple robots by non-linear time scaling. In *52nd IEEE Conference on Decision and Control*, pages 952–958, 2013.

[16] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. Stomp: Stochastic trajectory optimization for motion planning. pages 4569–4574, 05 2011.

[17] Steven M. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998.

[18] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[19] Huanwei Wang, Shangjie Lou, Jing Jing, Yisen Wang, Wei Liu, and Tieming Liu. The ebs-a* algorithm: An improved a* algorithm for path planning. *PLOS ONE*, 17:1–27, 02 2022.

[20] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[21] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *International Joint Conference on Artificial Intelligence*, 2011.

[22] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. Jax: composable transformations of python+ numpy programs, 2018. *URL http://github. com/google/jax*, 4:16, 2020.

[23] Claudio Gaz, Marco Cognetti, Alexander Oliva, Paolo Giordano, and Alessandro Luca. Dynamic identification of the franka emika panda robot with retrieval of feasible parameters using penalty-based optimization. *IEEE Robotics and Automation Letters*, PP:1–1, 07 2019.

[24] Rosen Diankov and James Kuffner. Openrave: A planning architecture for autonomous robotics. 04 2011.

[25] Reuven Y. Rubinstein and Dirk P. Kroese. *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-Carlo Simulation (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2004.

[26] Dipanwita Guhathakurta, Fatemeh Rastgar, M. Aditya Sharma, K. Madhava Krishna, and Arun Kumar Singh. Fast joint multi-robot trajectory optimization by gpu accelerated batch solution of distributed sub-problems. *Frontiers in Robotics and AI*, 9, 2022.

[27] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. Swarm robotic behaviors and current applications. *Frontiers in Robotics and AI*, 7:36, 2020.

[28] Ali Bolu and Ömer Korçak. Adaptive task planning for multi-robot smart warehouse. *IEEE Access*, 9:27346–27358, 2021.

[29] Juncheng Li, Maopeng Ran, and Lihua Xie. Efficient trajectory planning for multiple non-holonomic mobile robots via prioritized trajectory optimization. *IEEE Robotics and Automation Letters*, PP:1–1, 12 2020.

[30] Yuan Zhou, Hesuan Hu, Yang Liu, and Zuohua Ding. Collision and deadlock avoidance in multirobot systems: A distributed approach. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(7):1712–1726, 2017.

[31] Federico Augugliaro, Angela P Schoellig, and Raffaello 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*, pages 1917–1922. IEEE, 2012.

[32] Enrica Soria, Fabrizio Schiano, and Dario Floreano. Predictive control of aerial swarms in cluttered environments. *Nature Machine Intelligence*, 3(6):545–554, 2021.

[33] Carlos E Luis and Angela P Schoellig. Trajectory generation for multiagent point-to-point transitions via distributed model predictive control. *IEEE Robotics and Automation Letters*, 4(2):375–382, 2019.

[34] José Bento, Nate Derbinsky, Javier Alonso-Mora, and Jonathan S Yedidia. A message-passing algorithm for multi-agent trajectory planning. *Advances in neural information processing systems*, 26, 2013.

[35] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics Automation Magazine*, 4(1):23–33, 1997.

[36] Vivek K. Adajania, Aditya Sharma, Anish Gupta, Houman Masnavi, K Madhava Krishna, and Arun K. Singh. Multi-modal model predictive control through batch non-holonomic trajectory optimization: Application to highway driving. *IEEE Robotics and Automation Letters*, 7(2):4220–4227, 2022.

[37] Moritz Werling, Julius Ziegler, Sören Kammel, and Sebastian Thrun. Optimal trajectory generation for dynamic street scenarios in a frenet frame. In *2010 IEEE International Conference on Robotics and Automation*, pages 987–993. IEEE, 2010.

[38] Trevor Halsted, Ola Shorinwa, Javier Yu, and Mac Schwager. A survey of distributed optimization methods for multi-robot systems. *arXiv preprint arXiv:2103.12840*, 2021.

[39] Sudhir Kylasa, Fred Roosta, Michael W Mahoney, and Ananth Grama. Gpu accelerated sub-sampled newton's method for convex classification problems. In *Proceedings of the 2019 SIAM International Conference on Data Mining*, pages 702–710. SIAM, 2019.

[40] Michael Hamer, Lino Widmer, and Raffaello D'andrea. Fast generation of collision-free trajectories for robot swarms using gpu acceleration. *IEEE Access*, 7:6679–6690, 2018.

[41] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Gpu matrix multiplication. 05 2012.

[42] Fatemeh Rastgar, Arun Kumar Singh, Houman Masnavi, Karl Kruusamae, and Alvo Aabloo. A novel trajectory optimization for affine systems: Beyond convex-concave procedure. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1308–1315. IEEE, 2020.

[43] Gavin Taylor, Ryan Burmeister, Zheng Xu, Bharat Singh, Ankit Patel, and Tom Goldstein. Training neural networks without gradients: A scalable admm approach. In *International conference on machine learning*, pages 2722–2731. PMLR, 2016.

[44] Laura Ferranti and Tamas Keviczky. Operator-splitting and gradient methods for real-time predictive flight control design. *Journal of Guidance, Control, and Dynamics*, 40(2):265–277, 2017.

[45] Laura Ferranti, Rudy R Negenborn, Tamás Keviczky, and Javier Alonso-Mora. Coordination of multiple vessels via distributed nonlinear model predictive control. In *2018 European Control Conference (ECC)*, pages 2523–2528. IEEE, 2018.

[46] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154 vol.3, 2004.

[47] Deheng Zhu, Hiroaki Seki, Tokuo Tsuji, and Tatsuhiro Hiramitsu. Tableware tidying-up robot system for self-service restaurantndash;detection and manipulation of leftover food and tableware-. *Sensors*, 22(18), 2022.

[48] L.-H Hu, X.-P Li, and Wansheng Tang. Paper roll packing automatical by using industrial robots. 34:48–51, 04 2015.

[49] Aditya Agarwal, Bipasha Sen, Shankara Narayanan V, Vishal Reddy Mandadi, Brojeshwar Bhowmick, and K Madhava Krishna. Approaches and challenges in robotic perception for table-top rearrangement and planning. *arXiv preprint arXiv:2205.04090*, 2022.

[50] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[51] Yanhao He and Steven Liu. Analytical inverse kinematics for franka emika panda – a geometrical solver for 7-dof manipulators with unconventional design. In *2021 9th International Conference on Control, Mechatronics and Automation (ICCMA)*, pages 194–199, 2021.

[52] Nils Dengler, David Großklaus, and Maren Bennewitz. Learning goal-oriented non-prehensile pushing in cluttered scenes. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1116–1122, 2022.

[53] Baichuan Huang, Shuai D Han, Abdeslam Boularias, and Jingjin Yu. Dipn: Deep interaction prediction network with application to clutter removal. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4694–4701. IEEE, 2021.

[54] Baichuan Huang, Shuai D Han, Jingjin Yu, and Abdeslam Boularias. Visual foresight trees for object retrieval from clutter with nonprehensile rearrangement. *IEEE Robotics and Automation Letters*, 7(1):231–238, 2021.

[55] Fan Bai, Fei Meng, Jianbang Liu, Jiankun Wang, and Max Q-H Meng. Hierarchical policy for non-prehensile multi-object rearrangement with deep reinforcement learning and monte carlo tree search. *arXiv preprint arXiv:2109.08973*, 2021.

[56] Wenlong Mou, Zheng Wen, and Xi Chen. On the sample complexity of reinforcement learning with policy space generalization, 2020.

[57] Andrew C. Li, Pashootan Vaezipoor, Rodrigo Toro Icarte, and Sheila A. McIlraith. Challenges to solving combinatorially hard long-horizon deep rl tasks, 2022.

[58] Kaiyu Hang, Andrew S. Morgan, and Aaron M. Dollar. Pre-grasp sliding manipulation of thin objects using soft, compliant, or underactuated hands. *IEEE Robotics and Automation Letters*, 4(2):662–669, 2019.

[59] Jennifer King, Matthew Klingensmith, Christopher Dellin, Mehmet Dogar, Prasanna Velagapudi, Nancy Pollard, and Siddhartha Srinivasa. Pregrasp manipulation as trajectory optimization. In *Proceedings of Robotics: Science and Systems (RSS '13)*, June 2013.

[60] Weihao Yuan, Johannes A. Stork, Danica Kragic, Michael Y. Wang, and Kaiyu Hang. Rearrangement with nonprehensile manipulation using deep reinforcement learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 270–277, 2018.

[61] João Moura, Theodoros Stouraitis, and Sethu Vijayakumar. Non-prehensile planar manipulation via trajectory optimization with complementarity constraints. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 970–976. IEEE, 2022.

[62] Zhuo Xu, Wenhao Yu, Alexander Herzog, Wenlong Lu, Chuyuan Fu, Masayoshi Tomizuka, Yunfei Bai, C Karen Liu, and Daniel Ho. Cocoi: Contact-aware online context inference for generalizable non-planar pushing. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 176–182. IEEE, 2021.

[63] Changkyu Song and Abdeslam Boularias. Object rearrangement with nested nonprehensile manipulation actions. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6578–6585. IEEE, 2019.

[64] Shirin Joshi, Sulabh Kumra, and Ferat Sahin. Robotic grasping using deep reinforcement learning. *CoRR*, abs/2007.04499, 2020.

[65] Andrej Orsula, Simon Bøgh, Miguel Olivares-Mendez, and Carol Martinez. Learning to Grasp on the Moon from 3D Octree Observations with Deep Reinforcement Learning. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4112–4119, 2022.

[66] Wenxuan Zhou and David Held. Learning to grasp the ungraspable with emergent extrinsic dexterity. 2022.

[67] Tao Chen, Megha Tippur, Siyang Wu, Vikash Kumar, Edward Adelson, and Pulkit Agrawal. Visual dexterity: In-hand dexterous manipulation from depth. *arXiv preprint arXiv:2211.11744*, 2022.

[68] Tao Chen, Jie Xu, and Pulkit Agrawal. A system for general in-hand object re-orientation. *Conference on Robot Learning*, 2021.

[69] Ananye Agarwal, Ashish Kumar, Jitendra Malik, and Deepak Pathak. Legged locomotion in challenging terrains using egocentric vision. *CoRL*, 2022.

[70] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. *RSS*, 2021.

[71] Shikhar Bahl, Mustafa Mukadam, Abhinav Gupta, and Deepak Pathak. Neural dynamic policies for end-to-end sensorimotor learning. In *NeurIPS*, 2020.

[72] Mohammad Nomaan Qureshi, Ben Eisner, and David Held. Deep sequenced linear dynamical systems for manipulation policy learning. In *ICLR 2022 Workshop on Generalizable Policy Learning in Physical World*, 2022.

[73] Tobias Johannink, Shikhar Bahl, Ashvin Nair, Jianlan Luo, Avinash Kumar, Matthias Loskyll, Juan Aparicio Ojea, Eugen Solowjow, and Sergey Levine. Residual reinforcement learning for robot control, 2018.

[74] Shubham Pateria, Budhitama Subagdja, Ah-Hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys*, 54:1–35, 06 2021.

[75] Jesse Zhang, Haonan Yu, and Wei Xu. Hierarchical reinforcement learning by discovering intrinsic options, 2021.

[76] Nathan Ratliff, Matt Zucker, J. Andrew Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *2009 IEEE International Conference on Robotics and Automation*, pages 489–494, 2009.

[77] Mohak Bhardwaj, Balakumar Sundaralingam, Arsalan Mousavian, Nathan D. Ratliff, Dieter Fox, Fabio Ramos, and Byron Boots. Fast joint space model-predictive control for reactive manipulation. *CoRR*, abs/2104.13542, 2021.

[78] John Schulman, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, and Pieter Abbeel. Finding locally optimal, collision-free trajectories with sequential convex optimization. In *Proceedings of Robotics: Science and Systems*, Berlin, Germany, June 2013.