Motion Planning for Multiple Autonomous Robotic Vehicles using Optimization Tools

João Manuel de Vieira Garcia

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors:  Prof. Rita Maria Mendes de Almeida Correia da Cunha
              Prof. António Manuel dos Santos Pascoal

Examination Committee

Chairperson: Prof. João Fernando Cardoso Silva Sequeira
Supervisor:  Prof. Rita Maria Mendes de Almeida Correia da Cunha
Member of the Committee: Prof. Fernando Lobo Pereira

November 2019
Declaro que o presente documento é um trabalho original da minha autoria e que cumpre todos os requisitos do Código de Conduta e Boas Práticas da Universidade de Lisboa.

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
To my grandparents Manuel and Benvinda...
Acknowledgments

It would not have been possible to produce this thesis without the endless support of my colleagues Bruno Alves and Inês Cruz and specially of Bahareh Sabetghadam and my supervisors António Pascoal and Rita Cunha.

First, I want to thank António Pascoal, a professor who got me interested in control theory, modelling and simulating, and gave me confidence in my capabilities, as no other professor was able to do. It was with great joy that I received the news that professor Pascoal had accepted me as the student to research about such an important topic as optimal motion planning of autonomous vehicles, which was by far my first choice: not only because of the subject but also the supervisor.

Secondly, I want to thank Professor Rita Cunha, in her quality as supervisor, also helped me in the definition and execution of this work. The professor gave me assistance in solving mathematical problems especially when striving to find out if Medusa was differentially flat.

I really appreciate the approach my supervisors lead me to take, in order to solve this problem. From the first studies and background on all the methods available, until the end results obtained in my program, I always felt like I was learning something very innovative and elegant.

I would also like to thank IST-ID for supporting this work with a scholarship.

Lastly, I want to thank Bahareh Sabetghadam, who helped me tremendously every day and was always available to answer any questions regarding motion planning and Bézier curves. She was instrumental in guiding me towards the best methods of optimization and how they work, especially regarding collision avoidance and GJK algorithm. Furthermore, she helped me to solve every day problems of optimization related issues and not get stuck in my research.

Thank you all.
Abstract

Motion planning algorithms for autonomous vehicles are becoming increasingly relevant in the general area of robotics.

Cooperative robotics is also a very timely and challenging topic, in view of the large spectrum of applications that can benefit from the use of multiple robots working together to perform scientific and commercial missions in an efficient and reliable manner.

Motivated by the current trend, this thesis addresses the problem of cooperative motion planning for multiple vehicles. The work is motivated by an application example that requires steering a group of vehicles from desired initial to final positions with simultaneous times of arrival. Other objectives include avoiding collisions with external obstacles and among the vehicles themselves, while meeting constraints imposed by the vehicle dynamics. Temporal and energy-related objectives are also addressed.

To place the topic in perspective, the thesis first gives a general view of the problem of motion planning. This is followed by the development of a specific class of cooperative motion planning algorithms that build upon the parametrization of Bézier curves and differentially flat systems.

Using these tools, the problem of motion planning is cast in the form of an optimization problem that can be solved sufficiently fast to be used in real-time applications. The method proposed decreases significantly the complexity of the optimization problem, making it possible to compute, for a given cost criterion, optimal trajectories for 10 vehicles in approximately 2 seconds.

Keywords: Motion planning, autonomous vehicles, optimization, Bézier curves, flat systems
Resumo

O planeamento de movimento para veículos autónomos estão cada vez mais relevantes na área geral da robótica.

Cooperação robótica é também um tema muito oportuno e desafiante, em vista do largo espectro de aplicações que podem beneficiar do uso de múltiplos veículos robóticos trabalhando juntos para realizar missões científicas e comerciais de uma maneira eficiente e fiável.

Motivado pela tendência corrente, esta tese aborda o problema de planeamento de movimento cooperativo para múltiplos veículos. O trabalho é motivado por uma aplicação-exemplo que requer dirigir um grupo de veículos de certas posições iniciais a finais com tempos de chegada simultâneos. Outros objetivos incluem prevenção de colisões com objetos exteriores e entre os próprios veículos, enquanto cumprem restrições impostas pela dinâmica do veículo. Objetivos temporais e relacionados com a energia são também abordados.

Para meter o tópico em perspectiva, a tese dá uma visão geral do problema de planeamento de movimento. Isto é seguido pelo desenvolvimento de uma classe específica de algoritmos de planeamento cooperativo de movimento que se construem da parametrização de curvas e Bézier e sistemas que são differentially flat.

Usando estas ferramentas, o problema de planeamento de movimento é descrito na forma de um problema de otimização que pode ser resolvido suficientemente rápido para ser usado em aplicações em tempo real. O método proposto diminui significativamente a complexidade do problema de otimização fazendo com que seja possível calcular, para um certo critério de custo, trajetórias ótimas para 10 veículos em aproximadamente 2 segundos.

**Palavras chave:** Planeamento de movimento, veículos autónomos, otimização, curvas de Bézier, flat systems
Contents

Acknowledgments ................................................................. vii
Abstract .............................................................................. ix
List of Tables .......................................................................... xvii
List of Figures .......................................................................... xix
List of Acronyms ....................................................................... xxi

1 Introduction ........................................................................... 1
  1.1 Motivation .......................................................................... 1
  1.2 Overview ............................................................................ 2
  1.3 Optimization methods: a brief survey .................................. 3
  1.4 Thesis Outline .................................................................... 5

2 Trajectory parameterization ...................................................... 7
  2.1 Different parameterization methods .................................... 7
  2.2 Bézier Curves ..................................................................... 10
  2.3 Splines ............................................................................... 14
    2.3.1 B-Splines ...................................................................... 17
    2.3.2 Comments ..................................................................... 18
  2.4 Collision avoidance ............................................................ 19
    2.4.1 Minimum distance between samples .......................... 19
    2.4.2 GJK Algorithm .......................................................... 19
    2.4.3 Minimum distance between two Bézier Curves .......... 23
    2.4.4 Separating hyperplane ............................................... 24
    2.4.5 Considerations .......................................................... 26

3 Optimization methodologies ..................................................... 27
  3.1 Constrained Optimal Control Problem ............................... 27
  3.2 Methodologies .................................................................. 28
  3.3 Direct optimization methods ............................................ 28
    3.3.1 Direct single shooting method ................................. 28
    3.3.2 Direct multiple shooting method .............................. 29
    3.3.3 Direct collocation method ....................................... 30
3.4 Polynomial methods .................................................. 32
3.4.1 Differential flatness .............................................. 32
3.5 Problem formulation using polynomial methods .................. 33

4 Optimization in MATLAB ............................................. 35
4.1 Overview .............................................................. 35
4.2 Constrained non-linear optimization: fmincon .................... 36
4.2.1 Sequential Quadratic Programming .......................... 36
4.2.2 Interior Point Method ............................................ 39
4.2.3 Additional comments .......................................... 42
4.3 Alternatives to fmincon ........................................... 42

5 Implementation ......................................................... 45
5.1 Cost function ......................................................... 45
5.1.1 Approximated energy .......................................... 45
5.1.2 Unicycle model .................................................. 46
5.1.3 Hovercraft model .............................................. 47
5.1.4 Medusa model ................................................ 50
5.2 Initialization .......................................................... 50
5.2.1 Hungarian assignment ........................................ 50
5.2.2 Fixed coefficients: Initial and final constraints ............ 51
5.2.3 Bounds ............................................................ 53
5.2.4 Additional comments ........................................ 54
5.3 Flow chart ............................................................. 54

6 Results ................................................................. 57
6.1 Hovercraft ............................................................. 57
6.2 Unicycle ............................................................... 62
6.3 Comparing parameters of optimization in MATLAB .......... 64

7 Conclusions ............................................................. 69
7.1 Achievements ........................................................ 69
7.2 Future Work ......................................................... 69

Bibliography ............................................................ 71

A Motivating optimization problem .................................... 1
A.1 Problem Description ............................................... 1
A.2 Analytic Approach .................................................. 2

B Calculation of barycentric coordinates for point-line case ....... 5
Convex unconstrained optimization

C.1 Gradient Descent Method
C.2 Conjugate Gradient Method
C.3 Newton method
  C.3.1 Quasi-Newton method
C.4 Backtracking subroutine
  C.4.1 Comments

MATLAB code

D.1 Bézier operations
D.2 fseminf nonlinear and infinite constraints function
## List of Tables

6.1 Bounds and constants used for hovercraft simulations. .............................................. 57
6.2 Run time and energy of the obtained trajectories without using Hungarian assignment. . 58
6.3 Run time and energy of the obtained trajectories with Hungarian assignment with $T = 4000$ s. ........................................................................................................ 64
6.4 Constant inputs used for unicycle simulations. ................................................................. 64
6.5 Run time and energy of the obtained trajectories using SQP without Hungarian assignment with $T = 4000$ s. ................................................................. 65
6.6 Run time and energy of the obtained trajectories using interior-point with factorization subalgorithm without Hungarian assignment with $T = 4000$ s. ................................. 67
List of Figures

2.1 Parameterization of a trajectory in 1D using ZOH with different number of steps. 8
2.2 Parameterization of a trajectory in 1D using polynomials in monomial form. 9
2.3 Example of Bézier curve of order 17 in the parameterization of a trajectory. Solution obtained in 17 seconds. 10
2.4 Bernstein bases of orders 0 to 5. 11
2.5 Example of a high order Bernstein basis. 11
2.6 An example of a Bézier curve in 2D space. 12
2.7 Evaluation of point at time $\tau = 0.5$ using de Casteljau algorithm. 15
2.8 An example of a spline curve in B-form. 16
2.9 Spline in pp-form. 17
2.10 Spline in BB-form. 17
2.11 Optimal trajectories for two vehicles using discretized constraints. 19
2.12 Optimal trajectories for two vehicles using the method in Section 2.4.3. 23
2.13 Medusa bounded by a rectangle and a circle. 25
2.14 Optimal trajectories for two vehicles bounded by rectangles. 25
3.1 Example of transcription using trapezoidal collocation. 31
3.2 Example of transcription using Hermite-Simpson collocation. 31
5.1 Simplified hovercraft. 48
5.2 Program layout 55
6.1 Optimal trajectory for 3 hovercrafts and no obstacles. 60
6.2 Time evolution of all the physical variables for each hovercraft in the $T = 4000$ s case. 61
6.3 Optimal trajectory for 3 hovercrafts and 2 obstacles. The vehicles’ initial and final positions are represented by the colored dots in the edges of the trajectories. The vehicle starts in the bottom and goes up. The large dark blue circles represent two obstacles. The one in the bottom is static and the one in the top is moving. The time frame being depicted is at $t = 232$ s. The grey area is the fixed unfeasible region. 62
6.4 Time evolution of all the physical variables for each hovercraft in the $T = 400$ s case. 63
6.5 Optimal trajectory for 3 unicycles and 2 obstacles. The vehicles' initial and final positions are represented by the colored dots in the edges of the trajectories. The vehicle starts in the bottom and goes up. The large dark blue circles represent two obstacles. The one in the bottom is static and the one in the top is moving. The time frame being depicted is at $t = 208$ s. The grey area is the fixed unfeasible region.

6.6 Time evolution of all the physical variables for each unicycle for the $T = 400$ s case.

6.7 Run time and minimum cost function obtained by interior-point method using conjugate gradient sub-algorithm, sequential quadratic programming, and interior-point method with direct step (factorization) sub-algorithm, for the simulations using $T = 4000$ s.

A.1 Analytic results for the optimal control signals.

C.1 Example of iterations in Gradient Descent with optimal step size (green) and Conjugate Gradient Method (red) for minimizing a quadratic function.
List of Acronyms

**AUV**  Autonomous unmanned vehicle.

**BFGS**  Broyden-Floyd-Goldfarb-Shanno.

**CG**  Conjugate gradient.

**DFP**  Davidon-Fletcher-Powell.

**FLOPS**  Floating point operations per second.

**GJK**  Gilbert-Johnson-Keerthi.

**KKT**  Karush-Kuhn-Tucker.

**LBFGS**  Limited Broyden-Floyd-Goldfarb-Shanno.

**LP**  Linear programming.

**LQR**  Linear quadratic regulator.

**NLP**  Nonlinear programming.

**OCP**  Optimal control problem.

**ODE**  Ordinary differential equation.

**QP**  Quadratic programming.

**SQP**  Sequential quadratic programming.

**SR1**  Symmetric Rank 1.

**ZOH**  Zero-order hold.
Chapter 1

Introduction

1.1 Motivation

Motion planning is a key subject of utmost importance in autonomous robotic applications. There are several fields in which cooperative planning is crucial. As examples we cite the WiMUST\(^1\) [1] and MORPH [2] projects that aimed to endow groups of vehicles with the capability to map the seabed and perform geotechnical surveys by acting in cooperation. There are also very good prospects for the use of autonomous cooperative drones in different applications such as agriculture, security surveillance, areal mapping, fault inspection, and unmanned light cargo transportation, in which two or more drones must perform synchronized cooperative motions to successfully accomplish such tasks. Drones are also being used in cinematography to create 3D imagery autonomously [3], [4].

Automatic warehouses have also become pervasive in the past years, with numerous companies automating all their product storage management. Amazon\(^2\) and Alibaba\(^3\) own automatic warehouses in which there are several ground robots that plan their trajectories in a cooperative manner, fetching products for delivery 24/7 in an optimized manner, massively increasing the work flow and productivity.

One of the most prominent features of good trajectory planning for autonomous vehicles is the capability to optimize desired aspects related to motion, such as minimizing the time to maneuver, and reducing energy consumption, or a compromise thereof. Accomplishing these goals for multiple vehicles means that it will be possible to perform missions where energy consumption of a whole fleet of autonomous vehicles is as low as possible, while ensuring adequate cooperation between all vehicles to complete their mission. The constraints to be met include obstacle and inter-vehicle collision avoidance, as well as stringent constraints imposed by the dynamics of the vehicles.

With the development of small computers it is now possible to have more computational power on board of these vehicles, which allows for the solution of increasingly more complex motion planning problems, fast enough to meet real-time application needs.

---

\(^1\)WiMUST: [https://www.youtube.com/watch?v=DhihaZs-4DQ](https://www.youtube.com/watch?v=DhihaZs-4DQ)

\(^2\)Amazon: [https://www.youtube.com/watch?v=Ox05Eks2Q3s](https://www.youtube.com/watch?v=Ox05Eks2Q3s)

\(^3\)Alibaba: [https://www.youtube.com/watch?v=FB14Y55V2Z4](https://www.youtube.com/watch?v=FB14Y55V2Z4)
1.2 Overview

Computing the best trajectories to be tracked by single or multiple vehicles, with a view to meeting desired operational objectives in the presence of vehicle and environment related constraints translates into solving an optimal control problem. The solution of the latter is a set of trajectories embodying the requirements or needs expressed by the end-user. General needs usually consist in preventing collisions with the dynamic environment, in-between vehicles, and guaranteeing that all the vehicles specifications such as maximum velocity, torque, etc, are fulfilled. Other requisites can be more specific, such as making the vehicles go through desired waypoints at specific times, guaranteeing that the vehicles can communicate over an aerial or underwater communication network, etc. The latter requirements reflect on the stringent constraint that the vehicle’s topology allows for the establishment of good quality communication links.

Clearly, in the above context, optimization problems are tailored for the problems at hand. Important factors that deeply affect the optimization approach have to do with the fact of whether or not the environment is known and there are no unknown elements, if there are favorable physical conditions such as absence of drag and slide slip, if the model of the vehicles is simple and accurate, etc. All the above factors are true in the case of both Amazon and Alibaba automatic warehouses (known favorable environment and a simple accurate model), which can rely on discretized graph approaches. Other types of conditions are related to the occurrence of unpredictable strong winds or currents, unfavourable conditions such as significant drag in a marine vehicles, and whether or not the vehicle models capture the different operating conditions adequately. This is the case with outdoor drones and marine vehicles, which usually require a more careful approach. The problem addressed in this thesis concentrates on the latter case. The main goal of the thesis is the study and development of a multiple vehicle trajectory planning algorithm capable of being adapted to drones or marine vehicles of the Medusa class available at ISR. The vision is to use the algorithm as a tool to generate feasible trajectories in the fastest way possible and incorporate other objectives, such as active navigation. In the context of sea mapping missions, for example, the Medusa vehicles need to start their joint missions in completely aligned positions. This alignment is currently done manually and is very time consuming. However, the orderly motion of the group of the vehicles from initial to final positions may be done by tracking the trajectories that result from solving a so-called go-to-formation maneuver planning problem. This seemingly simple problem is indeed quite challenging and serves as the key motivation for the work in this thesis. Because the vehicles are required to operate for long periods of time, one of the goals is to generate minimum energy trajectories. Moreover, fast motion planning is required, since the vehicle’s initial positions are only known before the immediate start of a mission, and at sea it is very difficult to maintain a vehicle standing still. For this reason, it is also important that the vehicles arrive at the destination at the same time, i.e., go-to-formation maneuver includes a simultaneous time of arrival objective.

It is important to remark that the trajectories generated by a motion planner are to be used as references to be tracked by a group of vehicles. Thus, in practice, the vehicles must be equipped with the control systems responsible for closed-loop trajectory tracking. Another alternative is to make use of
optimization techniques so that it will be possible for the vehicles to react to external disturbances in real
time. In this context, if a vehicle is to be pushed out of the planned trajectory by a large disturbance,
instead of trying to go back to the original trajectory, the vehicle may try and adapt to this unexpected
situation by performing motion re-planning. The tools used to tackle this problem fall in the general
category of receding horizon, model predictive control [5]. The results of this thesis are also fast and
accurate enough to be implement this algorithm in a receding horizon manner, in real-time situations.
However, the main focus will be on the offline planning of trajectories.

1.3 Optimization methods: a brief survey

Optimization is a vast field of study leading to a plethora of techniques and algorithms, all with their pros
and cons. The first step in the adoption of a particular method is explained to obtain a general idea of
the different optimization methods.

Optimization methods, can be roughly classified into two major categories: direct and indirect meth-
ods. Indirect methods compute analytical first-order necessary conditions for optimality. These methods
are derived using the calculus of variations or, in a more elaborate form, using Pontryagin maximum
principle, and lead to the formulation of a two-point boundary-value problems. Direct methods transform
the original optimization problem into a finite non-linear programming (NLP) problem and are considered
to be more effective and numerically stable; however, they cannot guarantee optimality of the results.

In [6] and [7], indirect methods are used to compute low thrust optimal trajectories for spacecraft
applications. In [8], direct and indirect methods are used for trajectory optimization of multi-asteroid
rendezvous mission using solar sailing, depending on the number of visiting targets.

Most direct methods deal with discretized problems such as direct single shooting, direct multiple
shooting, and direct collocation methods that use pseudospectral methods, also known as orthogo-
nal collocation to solve the ordinary differential equation (ODE) that characterizes the dynamics of the
vehicle’s model. In [9], Legendre pseudospectral methods are compared with the Bernstein polyno-
mial approximations of the problem. In particular, it is shown that pseudospectral methods achieve fast
speed of convergence, but they do not guarantee that constraints are fulfilled at all times. Other collo-
cation methods that use trapezoidal collocation and Hermite-Simpson collocation can use splines and
their properties to ensure collision free trajectories; however, when the underlying dynamical systems
are differentially flat, the method is usually outperformed by polynomial methods.

In [10], a direct multiple shooting method is used for motion planning that includes active navigation
constraints. Even though the method can be applied to non-differentially flat systems, there is no guaran-
tee that the vehicles will not collide; furthermore, there is the need to solve an ODE at each iteration. The
method also uses FORCES PRO, which is a state of the art optimization software, but requires a paid
license, which would create a hurdle to be utilized as a tool in other users applications, and also requires
the problem to be in a specific formulation that does not encompass all general forms of optimization
problems. In [11], direct collocation is used for optimizing motion of MPI CyberMotion Simulator which
is a vehicle that moves in a very confined space with the intent to emulate the movement of specific real
Other discrete methodologies based on dynamic programming are also described in the literature. These approaches usually transform the original problem into a graph-related problem and make use of graph algorithms. Discrete methodologies are usually very fast and efficient. The most common approach is the celebrated A* algorithm, for which existence (completeness) and optimality of a solution is guaranteed if a consistent heuristic can be found. However, these discrete trajectories require the description of the environment in terms of a graph, which in turn requires complete knowledge and full characterization of every static and dynamic obstacle of the environment. Moreover, most A* algorithms do not consider the dynamics of the vehicle. In [12], A* search is used to find the best trajectory in 3D for an autonomous spacecraft with known obstacles. In [13], A* is used in marine applications in ice covered waters, for long distance trajectory planning. In this case, however, there is a need for human or other type of external supervision to ensure the feasibility and anti-collision of the trajectories.

In [13], D*, a dynamic version of A*, is used to plan the trajectories of multiple autonomous vehicles in partially known environments. This technique still requires the construction of a graph. The main difference between D* and A* is that D* uses, at each instant of time, knowledge of the trajectory computed thus far, in contrast with A*, that requires recalculating the whole trajectory, thus out performing A* in this situation. In [14], a graph planning approach is designed for quadrotor swarms with the use of roadmap generation with conflict annotation to detect collisions, followed by a discrete schedule constructed to prevent these collisions. A continuous trajectory optimization method is then implemented by using a support vector machine algorithm to construct separating hyperplanes which will construct "safe corridors". These corridors are used as bounds for splines, which guarantee that the trajectories will stay inside the corridors at all times, generating suboptimal smooth trajectories. To improve the results, an iterative refinement of the trajectories is performed. This permits to quickly compute collision free and dynamically feasible trajectories and then use the remaining time and computational power to enhance the obtained result. This remarkable approach brings together graph discrete planning, machine learning, and continuous trajectory optimization theory, and is shown to be able to compute trajectories for 200 robots in computer simulations and with 32 quadrotors in real experiments. However, a model of the environment must be provided using an octree, thus being suitable only for deterministic environments.

Focusing on differentially flat approaches, the usual parameterizations of trajectories consist of Bernstein polynomials, also known as Bézier curves, or piecewise polynomials, known as splines. For an introduction to differential flatness the reader is referred to [15].

In [16], trajectories for quadrotors are defined using spline parameterizations and exploit their properties to reduce the number of constraints. Separating hyperplanes, also defined as splines, are used to define anti-collision constraints. This avoids bounding vehicles by a safety radius and allows using other convex shapes to bound the vehicle, doing this in continuously by using spline coefficients which ensures that there are no collisions between samples of the trajectories.

In what concerns optimization algorithms in MATLAB, several options are presented that require the user to choose carefully which set of methods to use. The most relevant option is the minimization algorithm. In the present context, the most suitable MATLAB's algorithms are sequential quadratic pro-
gramming and interior point methods. However, there are other parameters to choose from in these algorithms that can greatly affect their performance.

Besides being easily accessible, the use of MATLAB’s built-in Optimization Toolbox was motivated by the results shown in [17], where a motion planning problem for 1000 vehicles in 3D was solved in 20 minutes. The simulation used minimal time of arrival of each vehicle to its designated destination, also subject to bounds on acceleration and velocity, with the added constraint of minimal temporal deconfliction between agents. The method used Hungarian assignment for the final positions. These results were obtained for a much simpler problem than the one being addressed here, for which the planning problem is nonlinear, the cost function includes the vehicles’ total energy spent and nonlinear, angular-related constraints, and anti-collision constraints. Nevertheless, the above paper showed that MATLAB’s optimization tool is not only highly flexible in terms of implementation, but can also be sufficiently fast for such applications.

In this thesis, an optimization algorithm is proposed for a general class of systems that includes two differentially flat vehicle models: a unicycle and a disk-like hovercraft, serving as a proxy for the Medusa AUV. Optimization using built-in MATLAB’s Optimization Toolbox is also reviewed in some depth in order to adopt the best possible results from this tool. Optimization using energy minimization is, surprisingly, very hard to find in literature. In [18], the energy minimization of the Medusa vehicle is examined in detail, and mission planning is performed first using a path planning approach that uses polynomials, followed by a time deconfliction strategy discretizing the problem and using A* algorithm, and also using PRONTO algorithm (Projection Operator Newton Trajectory Optimization) as another approach trajectory optimization which also did not discretize the problem. No explicit information is given on the time of computation that took this method to run completely, but it was reported to be too slow for real-time applications, even for a single vehicle.

In the present work, the dynamics of AUVs models are explicitly calculated and used in a cost function, and several MATLAB programs that apply specific Bézier properties not found in any Toolbox are also given. Moreover, a program example of MATLAB function \textit{fseminf} is given as one of the very few existing examples for the use of this function, which is specifically designed to solve semi-infinite optimization problems. A compilation of the main methods used in trajectory planning for multiple autonomous vehicles is also included.

1.4 Thesis Outline

This thesis is organized as follows. In Chapter 2, a number of trajectory parameterization approaches are presented, together with a simple motivating optimization example for comparison of the referred approaches. Moreover, collision avoidance methods are explained and compared in terms of efficacy and time complexity.

In Chapter 3, the most relevant optimization methodologies are compared, and the concept of differential flatness is introduced. The problem of motion planning is cast in the form of an equivalent optimal control problem, which is then greatly simplified exploiting the concept of differential flatness.
In Chapter 4, the optimization methods related to the computation of the optimal solution are presented. The goal is to present an overview on how MATLAB's optimization minimizer works, how to choose the best optimization algorithm, and how to choose the best options for the algorithm. To this end, the methods behind the MATLAB's solver \textit{fmincon} are briefly presented. Alternatives to this algorithm are also mentioned, and in particular, \textit{fseminf} is explained.

In Chapter 5, the implementation of the program is presented. Firstly, the cost function of the motion planning optimization problem is defined as an approximation of the true energy spent by the system. Secondly, the models for unicycle and hovercraft are introduced and shown to be differentially flat, and remarks are made on the Medusa model. Thirdly, the initialization of the optimization problem is discussed, in particular on finding the best set of final conditions by means of the Hungarian assignment method, and also how to manipulate the control points to achieve initial and final conditions outside the optimization problem. Finally, bounds on the physical variables are defined as linear constraints, followed by additional comments and a flow chart of the overall implementation.

In Chapter 6, the run time and energy consumption results obtained with the developed algorithm are presented for the hovercraft and the unicycle models and also, a comparison is made between the different algorithms for minimization in MATLAB.

Finally, Chapter 7 summarizes the main achievements of the thesis and discusses topics that warrant future work.
Chapter 2

Trajectory parameterization

2.1 Different parameterization methods

*First things first. What is a trajectory exactly?*

The word “trajectory” was best defined by Artur Wolek and Craig Woosley in Model-based Path Planning - “A trajectory is a time-parameterized state history that identically satisfies the equations of motion. (...) A path is simply a continuous, parametric curve through configuration space. While a path is parametric, it may not be parameterized by time (...)".

*But how can we translate this into an optimal trajectory?*

In order to find the best parameters that originate an optimal trajectory, a parameterization of the trajectory is required. In this section, different types of parameterizations will be described, along with an example of a solution of a small optimization problem, and later, anti-collision strategies will be discussed.

By definition, parameterization is a method to find the parametric equations of some shape. In a trajectory, this parameter is the time, and the parametric equations are functions of time which describe the evolution of a number of variables such as position, velocity, force, orientation, angular velocity and torque. Because these variables are needed to compute the energy associated with the execution of a trajectory, they must be computed as a function of some candidate trajectory.

**Discretization**

There are several ways to describe a trajectory. The most straightforward way is to divide it into small steps.

A simple example of such a parameterization using zero-order hold (ZOH), for a simple problem in 1D, is shown in Figure 2.1. In this example, the goal is to stop the vehicle in the least amount of time possible at the position 0, by finding the optimal input \( u \), which is also the variable being parameterized. A constrained optimization problem regarding a vehicle moving in a single direction is considered with
initial position at the coordinate 3 meters and final position at the origin. Given $u$, computing the position and the velocity of the vehicle are straightforward. This problem is described in Appendix A. The red line shows the analytical solution to the problem that is obtained using linear-quadratic programming (LQR), whereas the blue line shows the result obtained by optimizing each ZOH step of the trajectory. The LQR result is obtained for an infinite time horizon, thus, the optimal result with 10 seconds of time horizon will be an approximation of the infinite time horizon solution. However, since 10 seconds is a sufficiently long time horizon, both solutions are very close, so it is possible to use LQR to approximately solve this finite time horizon problem. The software used is \textit{fmincon} of the built-in MATLAB’s Optimization Toolbox.

![Graphs](image1.png)

(a) 20 step parameterization. Solution obtained in 7 seconds.  
(b) 200 step parameterization. Solution obtained in 119 seconds.

Figure 2.1: Parameterization of a trajectory in 1D using ZOH with different number of steps.

It is observed that for the 20 step case, the trajectory is not quite near the optimal one, whereas in the 200 step case it is. This shows that for this parameterization to be close to the actual optimal trajectory, a large amount of parameters may be required, which leads to more complex optimization problems and may lead to slow algorithms.

**Polynomials**

Another way to parameterize a trajectory is by approximating it using polynomials, which are parameterized by their coefficients.

Using this approach, one can use very few parameters to describe an optimal trajectory. This also gives differentiable results, and facilitates solving ordinary differential equations.

8
A polynomial can have several forms. The most common one is the monomial form described by

\[ P(t) = \sum_{k=0}^{n} a_k t^k, \]  

(2.1)

with \( P(t) \) being the polynomial of order \( n \), described as a function of time \( t \), with coefficients \( a_k \), and \( k \in \{0, \ldots, n\} \).

However, as the order increases, small variations in the high order coefficients may lead to large variations in the resulting trajectories. It is proved that increasing the order of the polynomial in an interpolation problem over a set of equidistant points will create an oscillation at the edges of that interval. This is called the Runge’s phenomenon [19]. Thus, there is limited accuracy when using polynomials in monomial form, and forcing significant accuracy can lead to very slow computations. Results for the previously described problem, with polynomials in monomial form, using its coefficients as optimization variables, are shown in Figure 2.2.

![Graphs showing polynomial and analytic solutions for different orders of polynomials.](image)

(a) 4th order polynomial. Solution obtained in 2 seconds.
(b) 18th order polynomial. Solution took a significant amount of time (above three hours).

Figure 2.2: Parameterization of a trajectory in 1D using polynomials in monomial form.

When comparing figures 2.1 and 2.2, it can be seen that with a small number of parameters one can get very close to the optimal trajectory using the monomial form, at the expense of very slow results.

However, there are other polynomial forms called Bernstein polynomials or Bézier curve that can, in certain circumstances, yield trajectories close to the to the optimal ones (Figure 2.3) and have very good properties, relevant for trajectory optimization problems, which will be described in the next section.
2.2 Bézier Curves

Bézier curves are also known as Bernstein polynomials. Sergei Natanovich Bernstein was an Ukrainian mathematician born in 1880 who invented this polynomial as a way to facilitate the constructive proof of Weierstrass approximation theorem. Later, Pierre Bézier, a French engineer working for Renault used curves constructed from Bernstein polynomials to model the geometry of car bodies. Publishing extensively his ideas caused these curves to be conventionally known as Bézier curves. To read more about their history, the reader is referred to [20].

Bernstein polynomials are defined as

\[ P(\tau) = \sum_{k=0}^{n} p_k B_{k,n}(\tau), \quad \tau \in [0,1], \]  

(2.2)

where \( p_k \) are the Bernstein polynomial coefficients, also called control points, \( B_{k,n}(\tau), k \in \{0, \ldots, n\}, \) are the Bernstein basis polynomials of degree \( n \), and \( \tau \in [0,1] \) denotes normalized time.

The Bernstein polynomial basis are given by

\[ B_{k,n}(\tau) = \binom{n}{k} (1 - \tau)^{n-k} \tau^k, \]

(2.3)

where \( \binom{n}{k} \) is a binomial coefficient. Using \( t = \tau T \), the time-scaled version of (2.2) can be written as

\[ P(t) = \sum_{k=0}^{n} p_k B_{k,n}(t), \quad t \in [0,T], \]

where

\[ B_{k,n}(t) = \binom{n}{k} \frac{t^k(T-t)^{n-k}}{T^n}. \]
The Bernstein bases using $\tau \in [0, 1]$ for several orders are shown in Figure 2.4\(^1\).

![Bernstein bases of orders 0 to 5.](image)

For higher orders, the Bernstein basis will be similar to those shown in Figure 2.5, with the Bernstein envelope shown in red.

![Example of a high order Bernstein basis.](image)

**Property 1: Initial and final points**

The initial and final values of a Bernstein polynomial are equal to the first and last coefficient, respectively. Thus, for a polynomial of order $n$,

\[
P(0) = p_0, \\
P(1) = p_n.
\]

\(^1\)http://mathworld.wolfram.com/BernsteinPolynomial.html
**Property 2: Convex hull**

A Bernstein polynomial is completely confined inside the convex hull defined by its coefficients.

These two properties are shown in Figure 2.6. The first control point is (0,1) and the final control point is (0,-0.5).

![Figure 2.6: An example of a Bézier curve in 2D space.](image)

**Property 3: Derivative and integral**

The derivative and integral of a Bézier curve are easily computed as

\[
P(t) = \sum_{k=0}^{n-1} (p_{k+1,n} - p_{k,n}) B_{k,n-1}(t),
\]

(2.4)

\[
\int_0^T P(t) dt = \frac{T}{n+1} \sum_{k=0}^{n} p_{k,n},
\]

(2.5)

Also, as a consequence of property 1, the derivatives at the initial and final points are given by \(p_1 - p_0\) and \(p_n - p_{n-1}\), respectively.

**Property 4: Summing/Subtracting two Bézier curves**

The sum of two Bézier curves

\[
f(t) = \sum_{i=0}^{m} f_{i,m} B_{i,m}(t),
\]

(2.6)

and

\[
g(t) = \sum_{i=0}^{n} g_{i,n} B_{i,n}(t)
\]

(2.7)
is also a Bézier curve described by

\[ f(t) \pm g(t) = \sum_{i=0}^{m} \left( f_{i,n} \pm \sum_{j=\max(0,i-m+n)}^{\min(n,i)} \binom{n}{j} \binom{m-n}{i-j} g_{j,n} \right) B_{i,m}(t). \]

**Property 5: Multiplying two Bézier curves**

The multiplication of two Bézier curves (2.6) and (2.7) is also a Bézier curve described by

\[ f(t)g(t) = \sum_{i=0}^{m+n} \left( \sum_{j=\max(0,i-n)}^{\min(m,i)} \binom{m}{j} \binom{n}{i-j} f_{j,m}g_{i-j,n} \right) B_{i,m+n}(t). \]

**Property 6: Alternating between Bernstein form and monomial form**

It is possible to convert the coefficients for a monomial basis \( a_k \) into those for a Bernstein basis from \( p_k \) according to

\[ p_k = \frac{1}{\binom{n}{k}} \sum_{i=0}^{k} \binom{n}{k} \binom{k}{i} a_i, \]

and convert from Bernstein form to monomial form by performing the computation

\[ a_k = \sum_{i=0}^{k} \binom{n}{j} \binom{j}{i} (-1)^{j-i} p_i. \]

**Property 7: Degree elevation**

For every curve of degree \( n \) and every \( m \geq n \), there exists a Bézier representation of degree \( m \) of that same curve \([21]\), such that

\[ P_{i}^{n+r} = \sum_{j=\max(0,i-r)}^{\min(n,i)} \binom{n}{j} \binom{r}{i-j} P_{j}^{m}, \quad i = 0, 1, \ldots, n + r \]

**De Casteljau’s Algorithm**

Paul de Faget de Casteljau was a French physicist and mathematician who worked at Citroën. Against the philosophy of their co-workers, he eagerly tried to describe car bodies mathematically. His persistence led to an increasing adoption of computer-aided design within Citroën from 1963 onward \([20]\) and the elaboration of so-called de Casteljau algorithm.

It is a recursive method used to evaluate a Bernstein polynomial \( P(\tau) \) at a specified value of \( \tau \). It is efficient, numerically stable, and used to divide the polynomial into two independent Bernstein
polynomials. This property will prove itself very useful when solving optimal problems with inter-vehicle collision avoidance constraints.

The method performs a series of line interpolations done in the convex hull, and generates a new set of outer lines with minus one dimension than the first one, as shown in Figure 2.7. These line interpolations are made between a set of new control points that generate a new convex hull. By doing this until there is only one line, a new control point is obtained exactly inside the curve. This point corresponds to the point evaluated at the specified $t_{div}$. This is mathematically translated by (2.8) evaluated at $t_{div} \in [0, T]$

$$P_i^{[k]} = p_{i,n}^{[k-1]} \frac{T - t_{div}}{T} + p_{i+1,n}^{[k-1]} \frac{t_{div}}{T}, \quad i = 0, \ldots, n - j, \quad k = 1, \ldots, n$$

which means that

$$P(t_{div}) = p_{0,n}^{[n]}.$$

The de Casteljau's algorithm not only evaluates the value of the curve at a given time, but also gives the coefficients for computing the two parts of the curve divided by the evaluation point.

Performing the algorithm in the example of Figure 2.6, which is defined in the interval $\tau \in [0, 1]$, and plotting the several coefficients of a Bézier curve with 8 control points obtained at each iteration, it can be seen in Figure 2.7 that, when performing an iteration, the number of control points (coefficients) decreases by one. Moreover, the required point is computed after 7 iterations, corresponding to the order of the curve. It can also be seen in the figure that de Casteljau's algorithm manages to split the Bézier curve into two new curves. This property will prove to be very useful in the collision avoidance method between Bézier curves, described in Section 2.4.3.

Another interesting conclusion is that one can get a much tighter convex hull by performing the algorithm. This means that if one wanted to ensure the distance between two curves using only the distance between two convex hulls, getting more coefficients would be a way to decrease the conservatism that this method would imply.

2.3 Splines

Splines are generalizations of Bézier curves. They were first discovered by the Romanian mathematician Isaac Jacob Schoenberg and the name comes from the strips of wood used by boat builders to create curves. These builders would use thin splints pulled into place by weights called ducks or knots which controlled the shape of spline. This is a rough idea of what a spline stands for in our context. Mathematically, a spline $s(\tau)$ is represented by piecewise polynomial functions which can be expressed in terms of a linear combination of the so-called B-splines [22] (see Section 2.3.1) and can be defined as

$$s_k(t) = \sum_{i=1}^{n} N_{i,k}(t) P_i,$$  

(2.9)
where $N_{i,k}$ are de B-splines of order $n$, $k$ is the $k^{th}$ interval with $k \in 1, \ldots, m$. Having a non-decreasing sequence called the knot vector $T = \{t_0, t_1, \ldots, t_m\}$ and control points $P_0, \ldots, P_n$, the knots define the time at which another polynomial piece is considered in the spline².

There are several ways to define a spline. In the next sections the B-form, pp-form, and BB-form are explained. The following examples were obtained using the Spline library in MATLAB³.

**B-form**

The B-form has become the most standard way to represent a spline during its construction since it is easy to build splines in B-form and it guarantees smoothness between each polynomial piece. The B-form describes a spline as

$$bs(x) = \sum_{j=1}^{n} B_{j,k}(x)a_j,$$

which corresponds to a weighted sum of B-splines (Section 2.3.1) of order $k$ and total number of coefficients $n$ that has to obey the rule

$$n > m + k,$$

explained in more detail in [19].

An example is shown in Figure 2.8 with $m = 10$ knots, and the order $n = 6$ in each of two dimensions, and order of spline $p = 10 - 6 - 1 = 3$, thus corresponding to a cubic spline.

This B-form is similar to Bézier curves. It requires few coefficients to have a smooth curve with a varied shape. It is, as mentioned, a good way to construct a spline and the one which uses less coefficients.

²http://mathworld.wolfram.com/B-Spline.html
However, the coefficients are not tied to a certain knot interval. This means that changing a coefficient changes the whole curve, as in Bézier curves. It is possible to work around this by transforming the B-form spline into pp-form or a BB-form.

**pp-form**

As with Bézier curves, polynomials in monomial form of degree $k$ can be seen as a spline of order $k$ or higher, with an arbitrary sequence of knots. Thus, one can express the monomials as linear combinations of B-splines over any knot sequence $T$ using the polar forms of monomials (explained in [21]), i.e., the pp-form provides a description in terms of its breaks $\xi_i, \xi_{i+1}$ (in which the two dot notation refers to the interior of a polynomial piece) and the local polynomial coefficients $c_{ji}$ of $l$ pieces given by

$$p_j(x) = \sum_{i=1}^{k} (x - \xi_j)^{k-i} c_{ji}, \quad j = 1, \ldots, l.$$ 

Transforming the previous example into pp-form gives the result of Figure 2.9. It can be seen that the B-form and the pp-form are equivalent. Since this is a cubic spline case, 4 coefficients are required in order to specify a cubic polynomial at each interval between knots. The pp-form is convenient for the evaluation of a spline, used in MATLAB's spline library, although there are other alternatives such as the de Boor’s algorithm [22] which is the generalization of de Casteljau algorithm for splines.

**BB-form**

It is also possible to easily convert a spline into several Bézier curves divided by the spline knots, which is called the BB-form (Bernstein-Bézier form). This gives the same curve, but with more coefficients. Although this is not ideal in terms of complexity, it permits to use Bézier curve algorithms which can prove to be very useful in later implementations and improvements of this thesis.

The equivalent example of Figure 2.8 in BB-form is shown in Figure 2.10.
In this Figure, the curve is the same, but the coefficients are now 4 per knot interval and the knots repeat themselves 4 times at each different time instant. More details about the transformation from B-form to BB-form in [19].

### 2.3.1 B-Splines

The term B-splines stands for basis spline. The way to compute them is by recurrence. First, the B-spline of order 1 is defined as

\[
N_{i,1}(t) = \begin{cases} 
1, & \text{if } t_i \leq t < t_{i+1} \\
0, & \text{otherwise}
\end{cases}
\]

In particular, if \( t_i = t_{i+1} \), \( N_{i,1} = 0 \).
The higher order B-splines are then computed as

\[ N_{i,j}(t) = \frac{t - t_i}{t_{i+j} - t_i} N_{i,j-1}(t) + \frac{t_{i+j+1} - t}{t_{i+j+1} - t_{i+1}} N_{i+1,j-1}(t) \]

where \( j \in [1, p] \) and \( p \) is the degree of the B-spline given by \( p \equiv m - n - 1 \).

Since the highest knot order required by the recursion formula (2.10) is \( m + p + 1 \), this means that for one of the most standard cases, which are cubic splines, where \( p = 3 \), taking the example in B-form, \( m = p + n + 1 = 3 + 6 + 1 \). Thus, the number of knots has to be higher than the actual knots used. This is solved by adding at least \( p \) equal knots to knots already defined, increasing their multiplicity, which means, in this example, that the knot vector is \((0, 0, 0, 0, 0.25, 0.5, 0.75, 1, 1, 1, 1)\).

The following properties of B-splines can now be stated [21], [23]:

- \( N_{i,j}(t) \) is piecewise polynomial of degree \( p \);
- \( N_{i,j}(t) \) is positive in \([t_i, t_{i+1}]\);
- \( N_{i,j}(t) \) is zero outside \([t_i, t_{i+1}]\);
- A B-spline is invariant under a translation and/or scaling of its knot sequence.

Properties regarding spline curves are similar to Bézier curves. Initial and final points are also given by the first and last control point respectively, the convex hull property is guaranteed at each knot interval and the derivative and integral are also fairly easy to compute (see [24], [23], [21]). Additionally, [24], [21] and [19] explain the property of knot insertion that permits to add knots to an already defined curve.

2.3.2 Comments

Splines, together with Bézier curves, are one of the most used curves in optimization. They both parameterize a trajectory fairly accurately using few parameters and avoid Runge’s phenomenon that appears when using monomial form polynomials. The fact that \( N_i \) is zero outside \([t_i, t_{i+1}]\) shows that altering a segment of a spline will not affect the other segments, which is not the case in Bézier curves. This makes splines easier to construct graphically, thus being used for graphical design applications. Moreover, splines offer the advantage of concatenation of curves and offer more control of shape which permits the curve to pass through waypoints by defining knots, constituting a fast and simple implementation in an optimization problem, since constraints could be avoided. They also offer flexibility in their behaviour by the control of the order of the splines.

However, in the proposed problem there are no waypoints to be achieved, since the goal is only to perform a go-to-formation with minimal energy and no collisions, and splines are slightly harder to formulate and to work with.

Contrarily to splines, Bézier curves do not have a MATLAB Toolbox that permits to use properties of these curves such as the ones listed in Section 2.2. For this reason, they were implemented in this thesis, and since no similar external MATLAB program or toolbox seemed to have implemented such properties explicitly, these operations are present in Appendix D.1.
2.4 Collision avoidance

Collision avoidance is one of the most important aspects one must guarantee in the optimization problem. In this section, three methods are explained and compared. Further, we discuss how to make the best use of Bézier curves to ensure anti-collision trajectories.

2.4.1 Minimum distance between samples

The fastest and most straightforward way to avoid collisions is to sample the trajectory with a sufficiently fine discretization and then measure the Euclidean norm between each of the samples of a pair of trajectories. However, it cannot be guaranteed that collisions will never occur between time samples, or that a minimum distance is always respected. This is less likely to happen when the sampling is increasingly finer, but doing so will lead to increasingly slower computations of the optimal solution.

The resulting trajectories in 2D space are shown in Figure 2.11 for a simple optimization problem of a simultaneous arrival problem, regarding two vehicles with initial and final conditions, bounded by a security radius and parameterized by Bézier curves. The vehicles are modelled as unicycles and the cost function is the total energy consumption. The vehicles’ initial points are the control points in the bottom and their final positions are the upper points. There are no additional constraints being considered at this point.

Figure 2.11: Optimal trajectories for two vehicles using discretized constraints.

It can be seen in Figure 2.11 that the vehicles collide, even though the solver signals this as a feasible solution. For that reason, other algorithms must be considered.

2.4.2 GJK Algorithm

To prevent collisions between vehicles one can use the convex-hull property of splines and Bézier curves. One conservative approach would be to compute the distance between two convex hulls which can be done efficiently using Gilbert–Johnson–Keerthi (GJK) Algorithm [25].
Invented in 1988, this particular algorithm computes the Euclidean distance between two convex shapes in linear time complexity, dependent on the total number of vertices of the two shapes. Furthermore, the algorithm can be used for spaces of any dimension.

To understand the GJK algorithm one needs to explore concepts like the simplexes, barycentric coordinates, Minkowski difference, the support operation and some basic geometric mathematics [26].

**Simplex**

A simplex can be seen as a triangle in a certain dimension. For example, a 0-simplex is a point, 1-simplex a line, 2-simplex a triangle, 3-simplex a tetrahedron. At each iteration, the GJK algorithm assesses if the the two shapes intersect or not. In this thesis, we only consider 2D cases, so the maximum simplex order needed is 2. GJK algorithm assesses collisions using simplexes constructed using the two convex shapes and measuring its distance to the origin, as explained further.

**Barycentric coordinates**

To compute distances between simplexes one can use the so called barycentric coordinates [27]. Given $n$ points in space $p_1, p_2, ..., p_n$, an $n - 1$ simplex is the set of points

$$\{p : p = \lambda_1 p_1 + \lambda_2 p_2 + \ldots + \lambda_n p_n \}$$

where each $\lambda_i$ is the barycentric coordinate of $p_i$. There are some useful properties regarding this coordinates which will be used in the future:

- All the barycentric coordinates sum to 1;
- If all the barycentric coordinates are between 0 and 1, then the point $p$ is inside the simplex;
- If there is at least one negative barycentric coordinate, then the point $p$ is outside the simplex.

**Minkowski difference**

The Minkowski difference of two sets, also called Minkowski sum, is formed by subtracting each point in one set to each point in the other set and is mathematically defined as

$$A - B = \{a - b \mid a \in A, \quad b \in B\}.$$

For example, if one took all the points inside a certain shape and subtract them from all the points enclosed in another shape, a new one would be obtained. One important consequence of this operation is that when the two figures overlap, there will be a set of points that are the same from each shape. Therefore, the shape resulting from the Minkowski difference will include the origin in case the two shapes ever intersect and will not include the origin if they do not intersect.
Support operation

The support operation eliminates the need to compute all the differences between each point that is enclosed by each shape which is, by itself, unfeasible to compute. It consists on finding the furthest point of a shape in a certain direction. This point is called “support point”. Given two sets $A$ and $B$ and a direction $d$, the support point of the Minkowski difference $A - B$ satisfies

$$p = \text{support}(d, A_i - B_i) = \max\{d(A_i - B_i)\}$$

$$= \max\{dA_i - d(B_i)\}$$

$$= \max\{dA_i\} - \max\{-d(B_i)\}$$

$$= \text{support}(d, A_i) - \text{support}(-d, B_i).$$

In fact, with two convex shapes one does not need to compute all the points from the Minkowski difference. It is sufficient to construct a convex hull of support points such as $p$ that enclose the origin.

General description

The GJK algorithm, not only computes the minimum Euclidean distance between two convex shapes $A$ and $B$, $d(A, B) = \min\{||x - y|| : x \in A, y \in B\}$, but also provides the two points from each shape which are closest to each other.

Let $W_k$ the set of vertices of the simplex in each iteration $k$ and $S = A - B$, i.e., the Minkowski difference between the shapes $A$ and $B$. In each iteration, GJK algorithm constructs a simplex contained in $S$ which is nearer to the origin than the previous iteration. It starts with a random point of $S$ which is $W_0$. This point $v_k$ is already the closest point of the simplex to the origin. Then, the vector pointing to the origin is computed from that point, which is easily done, since the point $v_k$ will have coordinates relative to the origin, so the vector pointing to the origin be $d = -v_k$. This is used to compute the support points in each convex hull that are furthest apart, by computing the support point $p_1$ in the first polygon in direction $d$, and the support point $p_2$ in the second polygon in direction $-d$. The resulting support point for the Minkowski difference is $p = p_2 - p_1$. It is important to note that the extremes of $S$ will coincide with one of the vertexes of $A$ or $B$, so, faster computations are achieved, if instead of computing the support point, one computes the nearest vertex to that support point. This will constitute the closest point to the origin of $S$, and will be added as a point to the current simplex. With the new line segment which is a 1-simplex, one needs to check if the line passes through the origin. This is done by computing the closest point in the line to the origin. If it does not intersect, then that point is used again to find a direction towards the origin, by a constructing a 2-simplex and computing the distance to the origin. In the 2-simplex there is also the need to check if the point is inside the triangle, because if it is, it should not be added to the new simplex, and another direction should be searched. The stopping condition will be either finding that the closest point of the simplex to the origin is the origin itself, or, if the support operation indicates a repeated a pair of vertices, which means that the origin is never attained.
Distance from a point to a line segment

The approach used to compute the distance from a point \( p \) to a line segment with \( A \) and \( B \) as extremes, is to calculate the closest point \( P \) of the line to the point \( Q \).

The point can be calculated as

\[
P = \lambda_1 A + \lambda_2 B.
\]

The barycentric coordinates are calculated in Appendix B and are given by

\[
\lambda_1 = \frac{\overrightarrow{AB} \cdot \overrightarrow{QB}}{||\overrightarrow{AB}||^2}, \quad \lambda_2 = \frac{\overrightarrow{AB} \cdot \overrightarrow{AQ}}{||\overrightarrow{AB}||^2}
\]

If the two coordinates are between 0 and 1, then the point \( P \) is inside the line, if \( \lambda_1 < 0 \), \( Q \) is closest to \( B \), if \( \lambda_2 < 0 \), \( Q \) is closest to \( A \).

Distance from a point to a triangle

In this case, the goal is to find the closest point \( P \) in a triangle defined by the vertices \( A \), \( B \) and \( C \) to the point \( Q \). To do this, one first has to find the closest edge or vertex of the triangle to \( Q \), which can be done by interpreting each edge as a line segment and performing the “distance from a point to a line” algorithm, described previously.

The barycentric coordinates of a 2-simplex are given by

\[
a = \overrightarrow{AB} x \overrightarrow{AC} y - \overrightarrow{AB} y \overrightarrow{AC} x
\]

\[
b = \overrightarrow{QB} x \overrightarrow{QC} y - \overrightarrow{QB} y \overrightarrow{QC} x
\]

\[
c = \overrightarrow{QC} x \overrightarrow{QA} y - \overrightarrow{QC} y \overrightarrow{QA} x
\]

\[
d = \overrightarrow{QA} x \overrightarrow{QB} y - \overrightarrow{QA} y \overrightarrow{QB} x
\]

where \( x \) and \( y \) denote the correspondent Cartesian coordinate of the vector and

\[
\lambda_1 = \frac{b}{a}, \quad \lambda_2 = \frac{c}{a}, \quad \lambda_3 = \frac{d}{a}
\]

If all these barycentric coordinates are positive, then the point is inside the triangle. And

\[
P = \lambda_1 A + \lambda_2 B + \lambda_3 C,
\]

if not, the problem resumes to find the closest point to the closest edge or vertex.
2.4.3 Minimum distance between two Bézier Curves

The minimum distance between two curves algorithm implemented in this thesis is based on the algorithm presented in [28]. Computing the minimum distance between the two trajectories, using this algorithm, allows to assess if the vehicles collide with absolute certainty, since it does not discretize the trajectory, but instead uses properties of the Bézier curves in a way that does not overlook what happens between samples. The algorithm takes advantage of properties 1 and 2 of Bézier curves (Section 2.2) and de Casteljau’s algorithm to recursively break the curve down to the pieces that are closest in time or in space to the other curve being evaluated. In order to do this, the above reference suggests the calculation of an upper and lower bound for the distance between the two curves and then use those results to divide the curve. Results for a similar problem to the one presented in Figure 2.11, using the minimum distance algorithm, are shown in Figure 2.12. The vehicles’ initial points are the control points in the bottom and their final positions are the upper points.

![Figure 2.12: Optimal trajectories for two vehicles using the method in Section 2.4.3.](image)

**Dividing the curve**

From the minimum distance combination of each curve define in \( \tau \in [0, 1] \), it is possible to divide the curve into two smaller halves in time, starting at \( \tau = 0.5 \), then \( \tau = 0.25 \), and so on. Then, it is possible to compute the closest half of each curve, and apply the splitting process until the curves can be approximated by points. Splitting the curves using de Casteljau’s algorithm gives the coefficients of the new curves. This allows creating a repeatable process of shortening infinitely a Bézier curve by means of another Bézier curve. Space and time deconfliction anti-collision strategies are decided by which combination of divided curves are chosen for evaluation. To perform space deconfliction, the algorithm must consider all curves of the two previously divided curves. To perform time deconfliction, the algorithm only needs to consider the curve segments that belong to the same time window, i.e., the same half with respect to time. Therefore, using this method, time deconfliction is easier to compute than space deconfliction.
Lower bound

The lower bound of the distance between the convex hulls of the two curves is calculated using GJK algorithm, thus corresponding to the minimum Euclidean distance between the convex hulls of the two curves, explained in detail in Section 2.4.2.

Upper bound

From property 1, it is possible to take one of the initial or final points from each curve and compute the distance between these points, which will always be greater or equal to the lower bound. For space deconfliction, the upper bound corresponds to the highest difference between the all the initial and final point combinations, but in time deconfliction, the distance is measured as the highest of the one between the initial points or the final points.

As the curve segment gets shorter, the upper bound tends to the lower bound. A tolerance is used to define how close they can be, thus balancing two wishes: a fast computation versus an exact distance.

The result of the algorithm will always be the upper bound of the distance between the convex hull of the two curves, thus, it will also correspond to an upper bound of the true distance. This means that this algorithm is always safe to use when assessing collisions, since it always presents either true or false reports of collisions, and never false reports of no collisions.

Additional comments

The biggest advantage of the “minimum distance between two Bézier curves” algorithm is the capacity to assess with certainty that no collisions will happen, although if the tolerance is too big, it can report false collisions. However, this algorithm is too slow to be used in every iteration of the optimization process when in high dimension problems.

Moreover, since this algorithm can only measure Euclidean distances, it can only provide a radius of security. That can be a problem in case the vehicles are long. For example, if a long vehicle, was to be modeled as a circle, there would be a significant amount of unnecessary transverse space covered by the circle as seen in Figure 2.13. This approach is, to a certain extent, conservative since it will provide trajectories that are not optimal. It is important to note that if a vehicle or obstacle modelled by any convex shape is static, one could compute the minimum distance between that convex shape and a Bézier curve by slightly altering the algorithm.

2.4.4 Separating hyperplane

It is also possible prevent collisions by using the separating hyperplane theorem [29] that states that two non-intersecting convex sets can always be separated by a hyperplane. In [30], an hyperplane is modelled by splines, and its coefficients, for offset and slope, are introduced as optimization variables. The great advantage of this method is that it becomes possible to model a vehicle as a rectangle or any other convex shape, and ensure that no collisions occur, provided that two vehicles or vehicle and
obstacle are on different sides of the hyperplane. Results for a similar problem to the one presented in Figure 2.11, for two vehicles bounded by rectangles, using separating hyperplanes, are shown in Figure 2.14. It can be observed that the circles in fact intersect, whereas the rectangles do not.

Figure 2.14: Optimal trajectories for two vehicles bounded by rectangles.

**Implementation**

Following the implementation of [30] for splines, as referred above the hyperplane separation can be implemented by modelling the slope $a(\tau)$ and offset $b(\tau)$ of the hyperplane as Bézier curves described as

$$a(\tau) = \sum_{i=1}^{n} c_i^a B_i^a(\tau), \quad b(\tau) = \sum_{i=1}^{n} c_i^b B_i^b(\tau),$$

with coefficients $c_i^a$ and $c_i^b$ (similar to [30], where they use splines) as additional optimization variables.

According to the hyperplane theorem, for convex shaped bounds each vertex has to be accounted
for in the hyperplane condition. Thus, the additional constraints are given by

\[
\begin{align*}
a(\tau)^T z_{j}^{veh_1}(\tau) + b(\tau) & \leq 0, \\
a(\tau)^T z_{k}^{veh_2}(\tau) + b(\tau) & \geq 0, \\
|a(\tau)|_2 & \leq 1, \\
j = 1, \ldots, n, \quad k = 1, \ldots, m, \quad \tau \in [0, 1],
\end{align*}
\]

in which \(z_{j}^{veh_1}\) is the vertex \(j\) of vehicle 1 with \(n\) vertices and \(z_{k}^{veh_2}\) is the vertex \(k\) of vehicle 1 with \(m\) vertices.

**Additional comments**

Separating hyperplane, permits more flexibility in terms of the computation of feasible trajectories. However, proper initialization is very important for the algorithm to be as fast as possible. Moreover, there are cases in which the best separating hyperplane is vertical, such as the one considered in Figure 2.14. In this case, the slope and offset are ambiguous since they can tend for plus or minus infinity when the hyperplane is parallel to the y-axis in the Cartesian space. For simple cases, one can just rotate the coordinate system by 90° but there are several more cases where this will not work, which could lead to false collisions or very slow computations. Another problem with separating hyperplanes is the vertex number. Comparing the cases with circles and rectangles, the problem is much slower in the rectangle case, since there are 3 extra constraints at each time step.

**2.4.5 Considerations**

As seen in Figure 2.13, the best shape to bound Medusa is a rectangle with a ratio of 3:1. However, the most practical solution in terms of efficiency of the algorithm is actually to bound the vehicle as a circle and measure the distance Euclidean between samples using an additional tolerance that is added to the radius to compensate for the relaxations made between samples and, to likely guarantee a feasible result at the expense of getting nearly optimal results. This is the usual approach found in literature, however, without a way to properly validate the result, the safety distances between vehicles are usually overly conservative or the sampling is significantly finer, which leads to slower algorithms. After the result is obtained, the “minimum distance between two Bézier curves” algorithm, described in Section 2.4.3 is performed to make sure that there are no collisions between samples. If there are, then the algorithm reruns with another initialization, until a feasible solution is found. This means that separating hyperplane is not considered in the case at hand. It is also possible to prevent collisions with absolute certainty by using minimum distance algorithm between vehicles and hyperplanes. However, this implementation is slow, since every trajectory of each vertex has to be considered against all existing hyperplanes. If the problem being considered can’t handle a radius relaxation, then, separating hyperplane can follow an implementation similar to the radius bounded case, using discretization during the optimization process, and then having its result validated by using minimum distance algorithm.
Chapter 3

Optimization methodologies

3.1 Constrained Optimal Control Problem

The motion planning problem that we consider in this thesis aims to perform a go-to-formation maneuver with simultaneous time of arrival respecting initial and final value constraints, bound constraints, dynamic feasibility and no-collisions using time deconfliction, for multiple vehicles. This has to be ensured in an optimal way with respect to energy consumption of the whole fleet. Mathematically, this problem can be formulated as a constrained optimal control problem of the form

$$\begin{align*}
\text{minimize} & \quad \int_0^T \sum_{v=1}^{N_v} L_v(x^{(v)}(t), u^{(v)}(t)) \, dt + \sum_{v=1}^{N_v} \Psi_v(x(T)) \\
\text{subject to} & \quad x^{(v)}(0) = x_0^{(v)}, \quad v = 1, \ldots, N_v \\
& \quad x^{(v)}(T) = x_f^{(v)}, \quad v = 1, \ldots, N_v \\
& \quad \dot{x}^{(v)}(t) = f(x^{(v)}(t), u^{(v)}(t)), \quad t \in [0, T], \quad v = 1, \ldots, N_v \\
& \quad x^{(v)} \leq x^{(v)}(t) \leq \bar{x}^{(v)}, \quad v = 1, \ldots, N_v \\
& \quad u^{(v)} \leq u^{(v)}(t) \leq \bar{u}^{(v)}, \quad v = 1, \ldots, N_v \\
& \quad c_{\text{col}}(x^{(v_1)}, x^{(v_2)}, u^{(v_1)}, u^{(v_2)}) \leq 0, \quad v_1 = 1, \ldots, N_v, \quad v_2 = 1, \ldots, N_v, \quad v_1 \neq v_2,
\end{align*}$$

where $T$ is the final time of the trajectory, $x^{(v)}(0)$ is the initial state of vehicle $v; v = 1, 2, \ldots, N_v$, with $N_v$ the number of vehicles, $x_f$ a final state, $\dot{x}^{(v)}(t)$ are the system dynamics, $c_{\text{col}}$ are inter-vehicle collision constraints, and the lower and upper bars represent bounds on the state, $x^{(v)}$, and inputs $u^{(v)}$.

A problem that only has the integral term $\sum_{v=1}^{N_v} L_v(x^{(v)}(t), u^{(v)}(t))$ is said to be in Lagrange form, a problem that has only the boundary objective $\sum_{v=1}^{N_v} \Psi_v(x(T))$ is said to be in Mayer form and a problem with both terms is said to be in Bolza form.

There are several optimization methodologies one can opt to use to solve this problem that will be briefly described in the following sections.
3.2 Methodologies

There are several different optimization methodologies that are suitable for solving motion planning problems. Because of the complexity of the majority of applications, optimal control problems (OCPs) are usually solved using numerical methods.

*Indirect* optimization methods solve trajectory optimization problems in three steps: Firstly, an analytical definition of necessary and sufficient conditions for optimality is constructed; secondly, these conditions are discretized, which turns the problem into a constrained parameter optimization problem; and finally, the optimization problem is solved. The major drawback in indirect methods is the need for finding analytical necessary and sufficient conditions for optimality which can be very complex.

*Direct* optimization methods usually do not require an analytical analysis of the problem, which makes them the most widely used methods for general optimization problems. For this reason they will be further described in the following section. For a full discussion on the different numerical methods for OCPs, the reader is referred to [31].

*Polynomial* methods transform the OCP problem into an equivalent simpler optimization problem by means of differential flatness properties [15].

3.3 Direct optimization methods

Direct methods would yield accurate solutions if the corresponding algorithms could be implemented with infinite precision arithmetics. They are used as a way to simplify an infinite optimization problem, into a finite one by means of *transcription* methods. For perspective, iterative methods are, in contrast to direct methods, not expected to terminate in a finite number of steps and converge to an exact solution.

Generally, direct methods solve trajectory optimization problems in two steps. First, the trajectory is discretized, which turns the problem into a constrained parameter optimization problem (in parameter optimization, the optimization variables are described by numerically), and then a simpler optimization problem is solved. Three methods for solving this simpler optimization problem, in a direct manner, are described in the next sections.

3.3.1 Direct single shooting method

The first step is to parameterize the control input signal $u(t)$ which can be considered piece-wise constant in time, defined as

$$u(t) = q_i, \quad t \in [t_i, t_{i+1}]$$

where $[t_i, t_{i+1}]$ denotes the time-step between successive parameterizations.

The second step consists in solving an Ordinary Differential Equation (ODE) with a specified initial
value. Mathematically, the steps consist of solving

\[ \dot{x}(t) = f(x(t), u(t; q)), \quad t \in [0, T], \]
\[ x(0) = x_0, \]

(3.3)

After solving the above equation, the state vector \( x(t) \) can now be seen as dependent variables in the considered time interval. Since they depend on \( u(t; q) \), where \( q = (q_0, q_1, ..., q_{N-1}) \), the states will be denoted as \( x(t; q) \). Note that in a fixed-time horizon the last component \( N \) is not an optimization variable. The third step is to solve the resulting optimization problem, given by

\[
\begin{align*}
\text{minimize} & \quad \int_0^T L(x(t; q), u(t; q))dt + \Psi(x(T; q)) \\
\text{subject to} & \quad h(x_i(t; q), u(t; q)) \leq 0, \quad i = 0, ..., N, \\
& \quad r(x(T)) = 0,
\end{align*}
\]

where \( L(x(t; q), u(t; q)) \) is the Lagrangian (or running cost) of the problem and \( \Psi(x(T; q)) \) is the terminal cost penalty, \( h(x_i(t; q), u(t; q)) \) consists of equality and inequality constraints and \( r(x(T)) = 0 \) are terminal constraints. To avoid a semi-infinite problem with finite number of variables to optimize and infinite number of constraints [32], the path constraints will also be discretized. This can be done using a finer grid than the one used for \( u(t; q) \) provided that the ODE software solver is able to return intermediate values between each step of \( u \).

The advantages of this method are that the only optimization variables in this NPL are \( q \) and that only initial guesses for the control input signal parameters are needed. The main disadvantage is that since this is done in one shot, one cannot use the knowledge of the state trajectory \( x(t) \) in the initialization, which can lead to slower convergence. Also, the ODE solution \( x(t; q) \) may depend in a strong nonlinear manner on \( q \) as in [5] and also, unstable systems are difficult to treat using this method.

The direct single shooting method is a so-called sequential method: it firstly computes the ODE of the general problem and, secondly, solves an optimization problem based on the solution of the ODE.

An example of a problem solved using direct single shooting is illustrated via Figure 2.1.

### 3.3.2 Direct multiple shooting method

As in direct single shooting, the first step in this method is to parameterize \( u \) as in (3.2).

The second step is to solve an ODE for each time-step separately, starting with an artificial initial value at each step \( s_i \), which translates into

\[
\begin{align*}
\dot{x}_i(t; s_i, q_i) &= f(x_i(t; s_i, q_i), q_i), \quad t \in [t_i, t_{i+1}] \\
x_i(t_i; s_i, q_i) &= s_i
\end{align*}
\]

where \( q_i \) is the control input signal parameters and \( s_i \) the state parameters, and the solution for each
ODE is represented as \( x(t; s_i, q_i) \) for \( t \in [t_i, t_{i+1}] \). To ensure continuity of the state trajectory, an additional equality constraint of the form (3.4) must be added to the optimization problem, as follows

\[
s_{i+1} = x_i(t_{i+1}; s_i, q_i), \quad i = 0, 1, \ldots, N - 1
\]

which has the form of a discrete dynamic equation. This means that, at each step the Lagrangian function is

\[
 l_i(s_i, q_i) = \int_{t_i}^{t_{i+1}} L(x(t; s_i, q_i), q_i) dt
\]

and, the resulting NLP problem is

\[
 \begin{align*}
 \text{minimize} & \quad \sum_{i=0}^{N-1} l_i(s_i, q_i) + \Psi(s_N) \\
 \text{subject to} & \quad s_0 = x_0 \\
 & \quad s_{i+1} = x_i(t_{i+1}, s_i, q_i), \quad i = 0, 1, \ldots, N - 1 \\
 & \quad h(s_i, q_i) \leq 0, \quad i = 0, \ldots, N, \\
 & \quad r(x(T)) = 0.
\end{align*}
\]

As in single shooting, the path constraints are discretized and a finer sampling can be used.

The strong points of this method are that one can use the knowledge of state trajectory in the initialization and that the method shows superior local convergence properties, in particular for unstable systems. However, the parameters to minimize are \( s \) and \( q \) whereas in single shooting, only \( q \) is required, which represents an increase of complexity in the minimization algorithm.

The multiple shooting method is a simultaneous method: while the optimization is performed, the dynamic equations are considered simultaneously, since they are included as constraints.

### 3.3.3 Direct collocation method

In direct collocation, the transcription method is used to approximate all continuous functions in the optimization problem as splines. This permits to define the functions by a small finite set of coefficients, and also to easily compute integrals and derivatives. The collocation methods usually involve two types of collocation: trapezoidal and Hermite-Simpson.

**Trapezoidal collocation**

Trapezoidal collocation works by approximating the control trajectory and the system dynamics as piecewise linear functions defined by linear splines, as seen in Figure 3.1, taken from [33]. For trapezoidal collocation, the knots are coincident with the collocation points.

The integrals resulting from the dynamics and the cost function can then be solved using trapezoidal approximation, which saves computational time.
Hermite-Simpson collocation

The Hermite-Simpson collocation uses quadratic splines to approximate the objective function and system dynamics between collocation points, as shown in Figure 3.2.

An advantage of using Hermite-Simpson collocation when comparing to trapezoidal collocation is that the state trajectory in usual OCPs is $\dot{x} = f(x,u)$. In this situation, $x$ would be a cubic Hermite spline, that has a continuous first derivative. The integrals resultant from the dynamics and the cost function can be solved using Simpson's quadratic rule, which saves computational time.

Orthogonal collocation

The trapezoidal and Hermite-Simpson collocation methods use low-order splines, whereas orthogonal collocation uses high-order splines. To ensure numerical stability, one needs to ensure that the implementation is numerically stable. The collocation points are located at the roots of orthogonal polynomials (the inner product between polynomials is zero), usually Legendre or Chebyschev. Using high-order orthogonal polynomials can lead to exponential convergence rates with the order of the polynomial, provided the function being approximated is sufficiently smooth, which is usually the case in trajectory optimization. When the entire trajectory is approximated using a single high-order polynomial, the resulting method is called pseudospectral collocation, or global collocation. This method is one of the most successfully implemented methods in motion planning and is very fast. However, as remarked in [9], it uses discretized constraints for collision avoidance, which does not guarantee that the vehicles do not collide because the methods fails to detect what happens in between collocation points, and even
discretizing such polynomials, it cannot guarantee that no collisions happen in-between samples.

3.4 Polynomial methods

Motion planning can also be done by trying to find the best polynomial that approximates the optimal trajectory and satisfies all the specified constraints, which translates into computing its coefficients. The main advantage of this method is that the dimension of the search space is usually much smaller than that of the search space associated with trying to find every sample of the control input that leads to an approximation of the optimal trajectory. Moreover, as seen in Figures 2.1 and 2.3, using Bézier polynomials can lead to more accurate results with a low number of coefficients. However, if the problem solved using only coefficients as optimization variables, there is the requirement that the system be differentially flat (Section 3.4.1).

The type of problems generated by polynomial-based methods are generally semi-infinite, since the optimization is constituted by a finite number of optimization variables and an infinite number of constraints. An elegant way to solve this problem is to use Bézier curves which arise from Bernstein polynomials and have the nice geometric properties described in Section 2.2 and in [9].

Defining \( x \) and \( y \) as the flat outputs that correspond to the position of the vehicle in \( \mathbb{R}^2 \), each trajectory is described as

\[
P(\tau) = \begin{bmatrix} x(\tau) \\ y(\tau) \end{bmatrix} = \sum_{k=0}^{n} p_k B_{k,n}(\tau).
\]

For simple polynomials \( B_{k,n}(\tau) = \tau^k \) (monomial basis) whereas in the case of Bézier curves, \( B_{k,n} \) has the form of a Bernstein polynomial and, is given as (2.3).

3.4.1 Differential flatness

A nonlinear system of the form

\[
\dot{x}(t) = f(x(t), u(t)), \quad x(t) \in \mathbb{R}^n,
\]

is said to be differentially flat [15], or simply flat if there exists a set of variables \( y = (y_1, \ldots, y_m) \) called the flat output, such that

- the components of \( y \) are not differentially related over \( \mathbb{R} \) [34];
- every system variable, state or input, may be expressed as a function of the components of \( y \) and a finite number of their time-derivatives;
- conversely, every component of \( y \) may be expressed as a function of the system variables and of a finite number of their time-derivatives.

Many realistic nonlinear models share this flatness property. Aside from all linear controllable systems, many nonlinear systems are differentially flat. The flat output has, in general, a clear physical
interpretation and captures the fundamental properties of a given system and its determination allows to considerably simplify the control design.

This implies that there is a fictitious flat output that can explicitly express all states and inputs in terms of the flat output and a finite number of derivatives [32]. Therefore, the dynamic constraints can all be expressed in terms of the polynomial trajectories, i.e., the polynomial coefficients of the trajectory are enough to describe the whole optimization problem.

3.5 Problem formulation using polynomial methods

Consider a system governed by
\[
\dot{x} = f(x, u), \quad x(0) = x_0,
\]
where \(x(t) \in \mathbb{R}^{N_x}\) is the state vector, \(u \in \mathbb{R}^{N_u}\), \(N_x\) is the number of coefficients in \(x\), \(N_u\) is the number of coefficients in \(u\). Assume the goal of the optimization problem where the goal is to steer the system from an initial state \(x_0\) at \(t = 0\) to the final state \(x_T\) at \(t = T\).

The cost function used in this thesis was the approximated energy for two differentially flat models: the unicycle model and the hovercraft model. As it will be seen in Section 5.1.2 and Section 5.1.3, finding a closed-form solution that depends on the control points of the trajectory is very hard. It actually leads to significantly complex expressions, that are slower to solve by substituting the values in the expressions, than using numerical methods to approximate the cost function.

For that reason, the cost function of (3.1) remains unchanged; however, being a differentially flat system, it is possible to compute the cost function using only the control points for the position. Therefore, the cost function is described by
\[
\int_0^T \sum_{v=1}^{N_v} L_v(x^{(v)}(t), u^{(v)}(t)) \, dt + \sum_{v=1}^{N_v} \Psi_v(x(T)).
\]

It is important to note that, in simpler problems, it is possible to find closed-form solutions for the running cost of (3.1).

At the same time, the control law must obey state and input constraints given by
\[
h(x(t), u(t)) \geq 0 \quad \forall t \in [0, T],
\]
where \(h : \mathbb{R}^{N_x} \times \mathbb{R}^{N_u} \to \mathbb{R}^{N_h}\) is the the constraint function. Assuming the system is differentially flat, this means the flat output \(y \in \mathbb{R}^{N_y}\) is of the form
\[
y = \phi(x, u, u^{(1)}, \ldots, u^{(q)}),
\]
and the states and inputs are described by
\[
x = \psi_x(y, y^{(1)}, \ldots, y^{(r-1)}).
\]
for some \( q, r \).

Differential flatness is particularly interesting when applied to optimal control problems such as (3.1), since it avoids complicated computations of integrals and derivatives which are often a mathematical and computational challenging step.

Admitting that the system is differentially flat, the optimization problem to be solved becomes

\[
\begin{align*}
\text{minimize} & \quad \int_0^T \sum_{v=1}^{N_v} L_v(x^{(v)}(t), u^{(v)}(t)) \, dt + \sum_{v=1}^{N_v} \Psi_v(x(T)) \\
\text{subject to} & \quad y^{(v)}(0) = y^{(v)}_0, \quad v = 1, \ldots, N_v, \\
& \quad y^{(v)}(T) = y^{(v)}_f, \quad v = 1, \ldots, N_v, \\
& \quad h(x^{(v)}(t), u^{(v)}(t)) \geq 0, \quad \forall t \in [0, T], \quad v = 1, \ldots, N_v, 
\end{align*}
\]

with \( h(x(t), u(t)) \) consisting of the inequalities

\[
\begin{align*}
c_{col}(x^{(v)}, u^{(v)}) & \geq 0, \quad v = 1, \ldots, N_v, \\
\underline{x}^{[v]} & \leq x^{(v)}(t) \leq \bar{x}^{(v)}, \quad v = 1, \ldots, N_v, \\
\underline{u}^{[v]} & \leq u^{(v)}(t) \leq \bar{u}^{(v)}, \quad v = 1, \ldots, N_v.
\end{align*}
\]

In conclusion, the flatness property greatly simplifies the optimization problem. There is no need for solving the ODE of the system dynamics in order to compute the states, since they are directly computed from the position. Differential flatness is one of the key elements in order to achieve fast computations in the algorithm that was developed in the context of this thesis.
Chapter 4

Optimization in MATLAB

We chose to implement the motion planning optimization software algorithm in MATLAB, using its Optimization Toolbox because of the significant amount of MATLAB users, and because it is already a built-in library of MATLAB that does not require any form of extra license payment. This is also an important fact, because the algorithm is supposed to serve as a basis for other implementations, which is the one main objectives of this thesis. The use of MATLAB’s built-in Optimization Toolbox was motivated by the results shown in [17], where a motion planning problem for 1000 vehicles in 3D was solved in 20 minutes. The simulation used minimal time of arrival of each vehicle to its designated destination, also subject to bounds on acceleration and velocity, with the added constraint of minimal temporal deconfliction between agents. The method used Hungarian assignment for the final positions. These results were obtained for a much simpler problem than the one being addressed here, for which the planning problem is nonlinear, the cost function includes the vehicles’ total energy spent with respect to the trajectory and also nonlinear angular-related constraints, as well as nonlinear anti-collision constraints. Nevertheless, the above paper showed that MATLAB’s optimization tool is not only highly flexible in terms of implementation, but can also be sufficiently fast for such applications.

There is a significant amount of solvers in the Optimization Toolbox [35] which are used in several different situations. Since the problem being addressed is a nonlinear constrained optimization problem, which is also semi-infinite, the functions to use from the Optimization Toolbox are fmincon and fseminf. The functioning of these minimizers will be explained in the following sections.

4.1 Overview

The preferred way to parameterize a trajectory is using control points to define Bézier curves. Moreover, since the problems considered are in 2D, each control point has two Cartesian components independently responsible for movements in each Cartesian axis. The goal is to manipulate these coefficients in order to create trajectories that minimize a certain cost function, such as the energy expenditure for a certain type of vehicles, as fast and efficiently as possible.

The best way to program MATLAB is to avoid, as much as possible, loops such as for or while and
try use MATLAB's library built-in functions, since MATLAB's functions are expected to be fully optimized. For example, it is much faster to compute a sum using MATLAB's function \texttt{sum} than to use a \texttt{for} cycle. In a complex optimization problem that may require tens of thousands of iterations, a few milliseconds in each iteration will have a tremendous impact on the total time of the algorithm. Therefore, a considerable care must be taken to monitor how much time MATLAB takes to perform each operation, which is why, in the optimization vector $\bar{x}$, the control points were all put in one column as

$$\bar{x} = \begin{bmatrix} x_0 & y_0 & x_1 & y_1 & \ldots & x_i & y_i & \ldots & x_{n\cdot N_v} & y_{n\cdot N_v} \end{bmatrix}^T,$$

with $x_i$ and $y_i$ being the two Cartesian coordinates, $n$ the order of the Bernstein polynomial, $N_v$ the number of vehicles, and $i \in \{0, \ldots, n \cdot N_v\}$.

Since $\bar{x}$ is a column vector and not a matrix, several operations such as the declaration of linear constraint matrices are facilitated. Moreover, the optimization vector can be quickly decomposed for further computations.

The implementation in MATLAB requires knowledge of the existent minimizers. Each minimizer requires several parameters to be chosen, that can affect deeply the way the algorithm is solved. This means that, in order to extract the best result out of MATLAB’s Optimization Toolbox, one needs to understand what every parameter means and how it will affect the accuracy of final result and computational time.

Although it is not considered the best option among all the optimizers, MATLAB’s Optimization Toolbox is still state-of-art technology, and offers a big flexibility in terms of implementation in exchange for usually slower algorithms. In order to understand how the complex Toolbox works, one must first examine the basic optimization algorithms. Convex unconstrained optimization theory is briefly explained in Appendix C.

### 4.2 Constrained non-linear optimization: \texttt{fmincon}

The minimizer used to obtain the final results is \texttt{fmincon}. Therefore, this algorithm is explored in depth, as a means to understand the reasoning behind one of the most important parts of the optimization, which is the minimization. Since there is no way to alter \texttt{fmincon}, and the design of an optimization algorithm is out of scope of this thesis, exploring this “black box” is key to understand exactly the behavior of the program, and learn how to define the best parameters for each type of problem.

This section follows the MATLAB Optimization Toolbox user guide [35] and summarizes the parts that affect the performance of the algorithm.

#### 4.2.1 Sequential Quadratic Programming

Sequential Quadratic Programming (SQP) is a so-called active set method [36], and represents the state of the art in nonlinear programming methods for medium scale optimization problems. An active set at $x$ corresponds to the set of boundaries of non strict inequality constraints, where $x$ is currently at.
In constrained nonlinear optimization, the general approach is to transform the original problem into an easier subproblem that can be solved and used as basis of an iterative process. One simple approach to constrained problems would be to include all the constraints in the cost function, and penalize $x$ that are outside the feasible region. However, these methods are now considered relatively inefficient and have been replaced by other methods that are focused on the Karush-Kuhn-Tucker (KKT) conditions.

The KKT conditions are the necessary conditions for optimality for a constrained optimization problem. If the problem is convex, i.e. the cost function and the constraint functions are convex, then the KKT conditions are both necessary and sufficient for global convergence.

Considering the optimization problem

$$
\begin{align*}
\text{minimize} & \quad f(x), \\
\text{subject to} & \quad h_1(x) = 0 \\
& \quad \vdots \\
& \quad h_p(x) = 0 \\
& \quad g_1(x) \leq 0 \\
& \quad \vdots \\
& \quad g_m(x) \leq 0,
\end{align*}
$$

in which $\mathcal{U}$ is the open set where strict inequalities can be considered, since they cannot put directly in the constraints of the optimization problem. The KKT equations of (4.1) are given by

$$
\begin{cases}
\nabla L(x^*) = \nabla f(x^*) + \lambda_1 \nabla h_1(x^*) + \ldots + \lambda_p \nabla h_p(x^*) + \mu_1 \nabla g_1(x^*) + \ldots + \mu_m \nabla g_m(x^*) = 0 \\
h_1(x^*) = 0, \ldots, h_p(x^*) = 0 \\
g_1(x^*) \leq 0, \ldots, g_m(x^*) \leq 0 \\
\mu_1 \geq 0, \ldots, \mu_m \geq 0 \\
\mu_1 g_1(x^*) = 0, \ldots, \mu_m g_m(x^*) = 0 \\
x^* \in \mathcal{U}
\end{cases}
$$

The first equation describes the objective of canceling the gradients of the objective function and the gradients of the active constraints at the solution point.

For the gradients to be canceled, Lagrange multipliers $\lambda_j, j = 1, \ldots, p$ and $\mu_i, i = 1, \ldots, m$ are necessary to balance the deviations of magnitude of the gradient of the objective function and gradient of the constraints.

**Quadratic Programming Subproblem**

SQP is based on Newton’s method with the addition of constraints. The key idea is to generate a quadratic programming (QP) subproblem whose solution is used to form a search direction for a so-
called line search procedure. Given the problem description of (4.1) and the set of equations (4.2), the QP subproblem is formulated as a quadratic approximation of the Lagrangian function

\[ L(x, \lambda) = f(x) + \sum_{i=1}^{m} \lambda_i \cdot g(x). \] (4.3)

This function approximates the constrained problem by an unconstrained problem. The direction to the next iteration \(d_k\) is computed by solving the problem

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2}(d_k + x_k)^T H_k (d_k + x_k) + c^T (d_k + x_k), \\
\text{subject to} & \quad A_{eq,i} (x_k + d_k) - b_{eq,i} = 0, \quad i = 1, \ldots, p, \\
& \quad A_j (x_k + d_k) - b_j \leq 0, \quad j = 1, \ldots, m,
\end{align*}
\] (4.4)

where \(A_{eq,i}\) is the \(i^{th}\) line of matrix \(A_{eq}\) and \(A_j\) is the \(j^{th}\) line of matrix \(A\). Knowing that \(A_{eq,i} x_k = b_{eq,i}\) and \(A_j x_k \leq b_j\), and that lateral translations (translations in the domain of the cost function) do not affect the minimum value of a convex quadratic function, it is possible to rewrite the above as

\[
\begin{align*}
\text{minimize} & \quad q(d_k) = \frac{1}{2} d_k^T H_k d_k + c^T d_k, \\
\text{subject to} & \quad A_{eq,i} d_k = 0, \quad i = 1, \ldots, p, \\
& \quad A_j d_k \leq 0, \quad j = 1, \ldots, m.
\end{align*}
\]

The step length parameter \(\alpha_k\) is determined by an approximate line search procedure so that sufficient decay is obtained in the cost function. Backtracking line search procedure is described in Appendix C.4.

The matrix \(H_k\) is a positive definite approximation of the Hessian matrix of the Lagrangian function (4.3). This matrix can be updated by any of the quasi-Newton methods (Appendix C.3.1), although BFGS (Broyden-Fletcher-Goldforb-Shanno algorithm [37]) appears to be the most popular method.

It is also worth mentioning that constrained problems usually require less iterations than unconstrained problems when using SQP. One of the reasons for this is because, due to the limits of the feasible area imposed by the constraints, the optimizer is able to make informed decisions regarding the directions and step length of each iteration.

The SQP solving phase involves two phases. The first phase involves the calculation of a feasible point if it exists. The second phase involves the generation of an iterative sequence of feasible points that converge to the solution. The working set is defined as the reunion of all the active sets estimated at a certain iteration, denoted by \(A_k\). The search direction \(d_k\) is such that it minimizes the cost function while remaining within every active constraint boundaries. The feasible subspace for the choice of \(\hat{d}_k\) is formed from a basis of \(Z_k\) whose columns are orthogonal to the active set estimation, which means, \(\tilde{A}_k Z_k = 0\). The search direction \(\hat{d}_k\) is then formed from a linear combination of the columns of \(Z_k\), which means \(\hat{d}_k = Z_k p\) for some vector \(p\). Doing this guarantees that the next iteration remains on the active constraints since \(\hat{d}_k\) is in the null space of \(\tilde{A}_k\). For a full description of how to obtain the matrix
The QP subproblem requires a feasible point to start. When a current point of the SQP is not feasible, then the following linear programming (LP) problem is solved

\[
\begin{align*}
\text{minimize} \quad & \epsilon, \\
\text{subject to} \quad & A_{eq,i}x_k = b_i, \quad i = 1, \ldots, p, \\
& A_i x_k - \epsilon \leq 0, \quad j = 1, \ldots, m.
\end{align*}
\]

(4.5)

By solving this problem, it is possible to find some \(x_k\) that satisfies the constraints using a slack variable \(\epsilon\).

It is important to note that the active set algorithms are not suitable for large scale problems since it needs to store full matrices that can take a significant amount of memory. Furthermore, dense linear algebra operations may require a long time to execute which can severely slow down the computation of large-scale optimization problems. For medium-sized problems SQP can perform better than interior point method, presented next, however, for large numbers of optimization variables and constraints, SQP proves to be very inefficient.

### 4.2.2 Interior Point Method

Interior point methods are known for their ability to deal with large NLPs. Similarly to SQP, the interior point approach to solve constrained optimization problems of the form of (4.1) is to reduce it to a sequence of approximate minimization problem by

\[
\begin{align*}
\text{minimize} \quad & f_\mu(x, s) = \min_{x, s} \left( f(x) - \mu \sum_i \ln(s_i) \right), \\
\text{subject to} \quad & h_1(x) = 0, \ldots, h_p(x) = 0 \\
& g_1(x) + s_1 = 0, \ldots, g_m(x) + s_m = 0,
\end{align*}
\]

(4.6)

where \(s_i > 0\) are slack variables with \(i \in \{1, \ldots, m\}\) and the penalty parameter \(\mu > 0\).

Analyzing (4.6), as \(\mu\) decreases to zero, \(f_\mu(x, s)\) approaches \(\min(f(x))\). The logarithmic term is called the barrier function.

Since (4.6) has only equality constraints, it is easier to solve (4.6) than (4.1). To solve the approximate problem, the algorithm uses one of two main types of step at each iteration:

- A direct step in \((x, s)\). This step tries to find a solution for the KKT equations (4.2) for the approximate problem via linear approximation. This is also called the Newton step.

- A conjugate gradient step using a trust region.

By default, the algorithm first attempts to take a direct step. If it cannot, it attempts a conjugate gradient (CG) step. One case where it does not take a direct step is when the approximate problem is not locally convex, near the current iterate.
At each iteration the algorithm decreases a merit function of the form

$$\phi_v = f_\mu(x, s) + \nu \| (h(x), g(x) + s) \|.$$  \hspace{1cm} (4.7)

The penalty parameter $\nu$ increases at each iteration to force the solution towards feasibility and less toward minimizing the cost function. This means that interrupting a certain minimization and starting again from the last point of the previous minimization is different running the minimization without interruptions. In this problem, the cost function is a desire of, for example, minimizing the overall energy spent, whereas the constraints regarding for example collision avoidance or bound constraints, are a much more important request in this context. Thus, when choosing parameters such as maximum number of iterations or minimum step size, one should have in consideration the fact that feasibility becomes more and more imposed in the algorithm, as the number of iterations increases. Moreover, if an attempted step does not decrease the merit function (4.7), the algorithm rejects the attempted step and attempts a new step. Therefore, as new increases $\nu$ the behaviour of the steps will change, and it may, in some cases, lead the algorithm to not converge.

If the objective function returns a value that does not belong to $\mathbb{R}$, such as, NaN (not a number), infinite or an error, the algorithm rejects the $x$ and, similarly to the reasoning for the merit function, the algorithm attempts another shorter step.

**Direct Step**

The direct step $(\Delta x, \Delta s)$ is defined as follows,

$$
\begin{bmatrix}
H & 0 & J_h^T & J_g^T \\
0 & S \Lambda & 0 & -S \\
J_h & 0 & I & 0 \\
J_g & -S & 0 & I \\
\end{bmatrix}
\begin{bmatrix}
\Delta x \\
\Delta s \\
-\Delta y \\
-\Delta \lambda \\
\end{bmatrix}
= -
\begin{bmatrix}
\nabla f - J_h^T y - J_g^T \lambda \\
S \lambda - \mu e \\
h \\
g + s \\
\end{bmatrix}
$$  \hspace{1cm} (4.8)

where

- $H$ denotes the Hessian of the Lagrangian of $f_\mu$ given by

$$H = \nabla^2 f(x) + \sum_i \lambda_i \nabla^2 g_i(x) + \sum_j y_j \nabla^2 h_j(x);$$

- $J_{g,i}$ denotes the Jacobian of the constraint function $g_i$;

- $J_{h,i}$ denotes the Jacobian of the constraint function $h_i$;

- $\lambda$ denotes the Lagrange multiplier vector associates with constraints $g$;

- $y$ denotes the Lagrange multiplier vector associated with constraints $h$;

- $S = \text{diag}(s)$
\[ \Lambda = \text{diag}(\lambda) \]

- \( e \) denotes the vector of ones with the same size as \( g \)

The equation (4.8) is directly taken from the KKT conditions using a linearized Lagrangian. The first equation corresponds to the linearization of the KKT cost function, the second equation corresponds to the slack variable heuristic linearization, and the last two are the linearizations of the equality and inequality constraints respectively.

In order to solve this equation for \((\Delta x, \Delta s)\), the algorithm makes an LDL factorization of the matrix which factorizes the matrix into a lower triangular matrix, followed by a diagonal matrix, followed by the conjugate of the first lower triangular matrix. This is a computationally expensive step, which permits to determine whether the projected Hessian is positive definite or not. If not, the algorithm uses CG step.

**Conjugate Gradient step**

The conjugate gradient approach to solving (4.6) is to adjust \( x \) and \( s \), keeping the slack variables \( s \) positive. See Appendix C.2 for more information regarding the conjugate gradient algorithm.

The general idea is to minimize a quadratic approximation of (4.6), which is also an approximation of the real optimization problem, inside a trust region, and also subject to linearized constraints.

Specifically, recalling the variables defined in Direct Step section the algorithm obtains the Lagrange multipliers by approximately solving the KKT equations (4.2) in the least-squares sense. Then, it takes a step \((\Delta x, \Delta s)\) to approximately solve

\[
\begin{align*}
\text{minimize} & \quad \nabla f^T \Delta x + \frac{1}{2} \Delta x^T \nabla_x^2 L \Delta x + \mu e^T S^{-1} \Lambda \Delta s, \\
\text{subject to} & \quad g(x) + J_g \Delta x + \Delta s = 0, \\
& \quad h(x) + J_h \Delta x = 0.
\end{align*}
\] (4.9)

The reader is referred to [38], [39] and [40] for more details and more information about the derivation of the algorithm.

By default, the Hessian is computed using BFGS. However there two more options for the algorithm that determines the Hessian such as LBFGS, where there is limited-memory large-scale quasi-Newton approximation, suitable for very large algorithms and *finite-difference* which produces an Hessian estimation based on finite differences of the gradient. The last method did not perform well on the tests since it works best when the gradient is defined analytically; however, the calculation of the gradient requires too much time or is impossible to compute analytically in some cases.

Since the problem at hand can easily become a large-scale problem, the best way to solve the problem is by using interior point with CG as the subproblem, as it will be seen in the results (Chapter 6).

By using CG in the subproblem, the algorithm does not spend time at each iteration trying to perform the direct step. Actually, results showed that conjugate gradient is the best approach to solve the final optimization problem. This is because the conjugate gradient takes a reasonably informed direction for
the step, and manages to converge relatively quickly to good local minima, even for high-order cases. The direct step approach aims to take an even more informed step than CG (performs a Newton step), but actually when it was used, the direct step commonly misguided the convergence of the algorithm, and produced worse results.

4.2.3 Additional comments

Since nonlinear approaches perform approximations to a quadratic convex problem, having a nonlinear problem with similar structure will help to get fast results. Bearing this in mind, the constraints and the cost function should be approximately quadratic, or easily linearized. As it will be seen in the results (Chapter 6), the algorithm actually performs well when using energy as the cost function. Also, all constraints are either explicitly linear or just linear operations with the sampled curves, which can only be implemented as nonlinear constraints. Therefore, the problem has a good structure and is easily solvable in most cases. Increasing too much number the of optimization variables will create non-sparse Hessians. This is due to the fact that altering a coefficient will change the whole trajectory for each vehicle. This leads to difficulties in adapting a trajectory when the number of control points is too high, so it is actually possible to get much better results using a low number of control points.

The main difference between SQP and interior-point methods is that SQP takes a more informed step in each iteration. However, this information may be hard to obtain. Firstly, a feasible point has to be found, and this can take time in case of complicated situations. The interior-point using conjugate gradient, however, takes a faster and less accurate approach which requires more iterations but does not fall behind SQP, even for small scale problems, as seen in Chapter 6.

The interior-point is also more agile in a sense where, if it does not find a feasible solution, it realizes it quickly, and that permits to retry the algorithm using different initialization and parameters such as finer sampling or higher tolerance between vehicles. SQP, on the other hand, needs a feasible point before it starts iterating properly, and this search for a feasible \( x \) can require a long time. This makes SQP much less reliable than interior-point methods.

4.3 Alternatives to \textit{fmincon}

Besides \textit{fmincon} there are several general nonlinear programming solvers, which are in fact, more popular than \textit{fmincon}: \textit{Forces Pro}, \textit{BARON IpOpt}, \textit{Knitro}, \textit{SNOPT}, \textit{DIDO}, to name but a few. However, these are all third party software, and most of them either require a license, are difficult to install or are not compatible with all operating systems.

There is however, an optimizer specialized in semi-infinite problems that is also present in MATLAB's Optimization Toolbox: \textit{fseminf}. 

It solves semi-infinite problems (finite number of optimization variables and infinite constraints) which is the case when formulating an optimization problem using Bézier curves, and it is a good alternative to \textit{fmincon} which solves finite optimization problems. For small number of continuous constraints, \textit{fseminf} performs well; however, because of the way the function is constructed, it is not possible to use a vector of infinite constraints in the program when programming. This means that each infinite constraint has to be individually declared and defined. When defining constraints for the 2 vehicle case, there is only one combination of vehicles, so there is only the need to define one constraint, but for 10 vehicles, there are 45 combinations, which have to be individually defined. This small detail has not yet been tackled by MATLAB and prevents \textit{fseminf} to be used for bigger problems. Infinite constraints considered by \textit{fseminf} are defined as

\[ K_j(x, w_j) \leq 0, \quad \forall w_j \in I_j, \]

where \( x \) are the optimization variables, \( w_j \) a continuous time instant and \( I_j \) an infinite set of the \( j \) constraint. The way the function deals with these constraints, is to reformulate them by an equivalent one given by

\[
\max_{w_j \in I_j} K_j(x, w_j) \leq 0 \quad \forall j = 1, \ldots, n_\theta, \tag{4.10}
\]

where \( n_\theta \) is number of semi-infinite constraints and \( K \) the vector of semi-infinite constraints.

After this step is complete, the function further simplifies the problem by performing piecewise quadratic and cubic approximations \( k_j(x, w_j) \) to the functions \( K_j(x, w_j) \) of (4.10) at each iteration of the algorithm. These type of approximations are an interpolation problem which is know to be a least-squares problem, thus having a closed-form solution and being easy to compute. In the final step, \textit{fseminf} takes the maximum value of each \( k_j(x, w_j) \) and reduces the problem to a finite optimization problem. This decreases the chances of collisions to happen between samples, but since the approximations are not exact, it does not prevent that there are no collisions between samples. Thus, there is still need to run the “minimum distance between two Bézier curves” algorithm (Section 2.4.3) to make sure the trajectory is totally safe, but it does reduce the number reruns of the optimization algorithm and increase its overall performance.

Even though this algorithm is explained step-by-step in further detail in [35], little information was found about the implementation of \textit{fseminf} and only 2 examples have been found (provided by MATLAB, for simple cases). For this reason, the code used for the nonlinear and infinite constraint function for the case of 4 vehicles and zero obstacles is shown in Appendix D.2.

Summarizing, \textit{fseminf} discretizes the problem using the maxima of piecewise quadratic and cubic approximations of the infinite constraints, with sampling times which can vary from different iterations, instead of zero-order hold approximations used in \textit{fmincon}, using fixed sampling times. However, it is not yet extendable to higher order problems, so it will not be used in the motion planning algorithm.
Global minimum

Being certain that the minimum found by the optimizer is actually the global minimum of a non-convex problem is usually not an easy task. There is a special MATLAB toolbox called Global Optimization Toolbox which is specifically made for finding global minima. There are two algorithms to choose from: GlobalSearch and MultiStart. The GlobalSearch algorithm is preferentially directed for finding a single global minimum in the most efficient way, using a single processor. It uses only fmincon as the minimizer. On the other hand, MultiStart is more directed for finding local minima, and can also run in parallel. Furthermore, it gives the option to choose several minimizers whereas Global Search only uses fmincon. MultiStart also allows to explore the results of user-defined starting points for the algorithm.

These algorithms are a good choice when there is little information about the global minimum. However, there are ways to get near the global minimum in optimal motion planning with respect to energy or time, provided that the initialization is near enough. In most cases, initializing the algorithm trajectories as straight lines from the origin to its destination ensures, to a certain degree, that the minimizer converges to the global minimum. In this context, the running time of the algorithm is a big factor, using only fmincon is sufficient. Thus, neither GlobalSearch nor MultiStart are used. Nevertheless, the methodology of using different initializations when the algorithm fails is still present in the final algorithm developed in this thesis, but is implemented in a more specific manner to tackle the problem at hand.
Chapter 5

Implementation

5.1 Cost function

In this section, we describe the method used to obtain a cost function that accounts for the energy consumption of a trajectory with regard to a certain vehicle model. The goal is to explore the differential flatness property to fully describe the states and inputs of the system, using solely the position vector and its time derivatives, and then use the Bézier control points as the new optimization variables.

5.1.1 Approximated energy

Calculating the true energy of the vehicle, such as a Medusa vehicle, is an extremely complex problem that can lead to very slow computations of the optimal solution [18]. Therefore, there is the need to formulate an expression that is sufficiently accurate to generate good trajectories but also simple enough to achieve fast computations. The approximation of the energy that was chosen as cost function is given by

\[
E = \int_{0}^{T} \left| F(t) \cdot v(t) \right| + \left| T_\omega(t) \cdot \omega(t) \right| \, dt,
\]

where \( F(t) \) represents the linear force, \( v(t) \) the linear velocity, \( T_\omega(t) \) the torque and \( \omega(t) \) the rotational velocity. This expression accounts for the energy spent in translational and rotational movement, and corresponds to the kinetic energy spent in the trajectory with respect to the vehicle model.

These variables are expressed in the body frame and not in the world frame. The body frame coordinates are obtained using a rotational matrix and a model of the vehicle’s motion. To describe the motion of the vehicle and the time evolution position and orientation of a body fixed frame with respect to the world frame, a dynamic model needs to be defined. In this chapter, we describe the dynamic models for three vehicles: the unicycle, the hovercraft and the Medusa model.

For all these models one needs to compute all state and input variables as functions of the positions in the world frame.

One particularity about these models is that, when the vehicles in question apply negative forces, the propellers also use energy, so this also counts for the cost function. For this reason, the absolute value
5.1.2 Unicycle model

The unicycle model can be written as

\[
\begin{bmatrix}
\dot{x} \\
\dot{y} \\
\dot{\psi}
\end{bmatrix} =
\begin{bmatrix}
\cos \psi & -\sin \psi & 0 \\
\sin \psi & \cos \psi & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u \\
r
\end{bmatrix}.
\]

(5.2)

Assuming that \(x(t)\) and \(y(t)\) are Bézier curves, we would like the optimization variables to be the coefficients of \(x\) and \(y\) which are given by,

\[
x(t) = \sum_{k=0}^{n} c_{x,k} B_{k,n}(t),
\]

\[
y(t) = \sum_{k=0}^{n} c_{y,k} B_{k,n}(t),
\]

with \(t \in [0, T]\).

Using the differentiation rule (2.4) the velocity depends on the coefficients as

\[
\dot{x}(t) = \frac{n}{T} \sum_{k=0}^{n-1} (c_{x,k+1} - c_{x,k}) B_{k,n-1}(t),
\]

\[
\dot{y}(t) = \frac{n}{T} \sum_{k=0}^{n-1} (c_{y,k+1} - c_{y,k}) B_{k,n-1}(t).
\]

Using again the differentiation rule

\[
\ddot{x}(t) = \frac{n(n-1)}{T^2} \sum_{k=0}^{n-2} (c_{x,k+2} - 2c_{x,k+1} + c_{x,k}) B_{k,n-2}(t),
\]

\[
\ddot{y}(t) = \frac{n(n-1)}{T^2} \sum_{k=0}^{n-2} (c_{y,k+2} - 2c_{y,k+1} + c_{y,k}) B_{k,n-2}(t).
\]

The yaw angle \(\psi\) can be written as

\[
\psi = \arctan \left( \frac{\dot{y}}{\dot{x}} \right),
\]

which means the rotational velocity of the body is given by

\[
r = \frac{\dot{x}\ddot{y} - \ddot{x}\dot{y}}{\dot{x}^2 + \dot{y}^2}.
\]

Solving (5.2) for \((u, r)\)

\[
\begin{bmatrix}
u \\
r
\end{bmatrix} =
\begin{bmatrix}
\cos \psi & \sin \psi & 0 \\
-\sin \psi & \cos \psi & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\dot{x} \\
\dot{y} \\
\dot{\psi}
\end{bmatrix}.
\]
The force in $u$, denoted by $\tau_1$, and torque $\tau_3$ can also be recovered from the position and its derivatives using

$$\tau_1 = m(r(-\dot{x}\sin \psi + \dot{y}\cos \psi) + \ddot{x}\cos \psi + \ddot{y}\sin \psi),$$

$$\dot{\tau}_3 = I_z \dddot{y} - x(3)\dot{y} - 2(\dot{x}\ddot{x} + \dot{y}\ddot{y})r.$$

So, it is possible to fully characterize this model using the control points as input and, since the inputs are not differentially related over $\mathbb{R}$, the system is differentially flat.

### 5.1.3 Hovercraft model

The hovercraft model is studied, since it is the closest differentially flat model to the Medusa vehicle. The hovercraft model that will be adopted is described in [41], where its differential flatness is proved, which means the problem can be solved using polynomial methods (see Section 3.4). The Medusa and the hovercraft are supposed to move in an horizontal plane.

Firstly, to explain the hovercraft some definitions are required:

- The vector $v = [u, v, r]$ denotes linear velocities in surge, sway and angular velocity in yaw (around the z-axis in the body frame);

- The vector $\eta = [x, y, \psi]$ denotes the position and orientation of the body frame with respect to the world frame;

- The vector $\tau = [\tau_1, \tau_2, \tau_3]$ denotes the control forces in surge and sway and also the control torque in yaw;

- The diagonal matrix $M = \text{diag}\{m_{11}, m_{22}, m_{33}\}$ represents the constant mass and inertia matrix in each component of the body axis;

- The diagonal matrix $D = \text{diag}\{d_{11}, d_{22}, d_{33}\}$ represents the constant hydrodynamic damping coefficients matrix in each component of the body axis;

- The matrix $C(v)$ is the Coriolis matrix and centripetal forces, given by

\[
C(v) = \begin{bmatrix}
0 & 0 & -m_{22}v \\
0 & 0 & m_{11}u \\
m_{22}v & -m_{11}u & 0
\end{bmatrix};
\]

- The matrix $J(\eta)$ is the rotation matrix over yaw angle $\psi$ about the z-axis of the world frame for the linear velocities in both frames

\[
J(\eta) = \begin{bmatrix}
cos \psi & -\sin \psi & 0 \\
\sin \psi & \cos \psi & 0 \\
0 & 0 & 1
\end{bmatrix}.
\]
In [42], the general model is proposed

\[
\begin{align*}
M \ddot{v} + C(v)v + Dv &= \tau \\
\dot{\eta} &= J(\eta)v
\end{align*}
\]

(5.3)

The hovercraft model used in [43], considers an hovercraft with a symmetric shape as shown in Figure 5.1 taken from [41].

Using this simplification enforces the following assumptions

\[
m_{11} = m_{22}, \quad \tau_1 = m_{11}\tau_u, \quad \tau_2 = 0, \quad \tau_3 = m_{33}\tau_r, \quad d_{11} = d_{33} = 0, \quad \beta = \frac{d_{22}}{m_{22}}
\]

(5.4)

The first simplification \( m_{11} = m_{22} \) comes from the fact that the vessel is symmetric, thus, the mass with respect to \( x \) and \( y \) axis is the same. This simplification is what makes this hovercraft model differentially flat when comparing to the general underactuated surface vessel model (5.3). The fact that \( \tau_2 \) is zero, shows that the hovercraft is underactuated, which is also the case of the Medusa. There is also an assumption that the hydrodynamics \( d_{11} \) and \( d_{33} \) are both zero, which further simplifies the calculations but does not affect the differential flatness property. If these coefficients were indeed present, they could be readily compensated by partial state feedback through the input forces \( \tau_1 \) and \( \tau_3 \).

With these considerations, the following kinematic model is obtained

\[
\begin{bmatrix}
\dot{x} \\
\dot{y} \\
\dot{\psi}
\end{bmatrix} =
\begin{bmatrix}
\cos \psi & -\sin \psi & 0 \\
\sin \psi & \cos \psi & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u \\
v \\
r
\end{bmatrix}
\]

(5.5)

It is possible to eliminate the \( \psi \) from the equations (5.3). This is done by performing the following
coordinate transformation proposed in [44]  

\[
\begin{align*}
  z_1 &= x \cos \psi + y \sin y \\
  z_2 &= -x \sin \psi + y \cos \psi \\
  z_3 &= \psi
\end{align*}
\]

which results in the following dynamic model

\[
\begin{align*}
  \dot{u} &= \frac{m_{22}}{m_{11}} vr - \frac{d_{11}}{m_{11}} u + \tau_1, \\
  \dot{v} &= -\frac{m_{11}}{m_{22}} vr - \frac{d_{22}}{m_{22}} v, \\
  \dot{r} &= \frac{m_{11} - m_{22} u v}{m_{33}} - \frac{d_{33}}{m_{33}} r + \tau_3.
\end{align*}
\]

Applying the simplifications (5.4)

\[
\begin{align*}
  \dot{u} &= vr + \tau_u, \\
  \dot{v} &= -ur - \beta v, \\
  \dot{r} &= \tau_r.
\end{align*}
\]

The following expressions are derived in [41], and prove that the system is differentially flat:

\[
\begin{align*}
  \Upsilon_x &= \ddot{x} + \beta \dot{x}, \quad \Upsilon_y = \ddot{y} + \beta \dot{y}, \\
  \psi &= \arctan \left( \frac{\Upsilon_y}{\Upsilon_x} \right), \\
  u &= \frac{\dot{x} \Upsilon_x + \dot{y} \Upsilon_y}{\sqrt{\Upsilon_x^2 + \Upsilon_y^2}}, \\
  v &= \frac{\dot{y} \ddot{x} - \dot{x} \ddot{y}}{\sqrt{\Upsilon_x^2 + \Upsilon_y^2}}, \\
  r &= \frac{y^{(3)} \Upsilon_x - x^{(3)} \Upsilon_y - \beta^2 (\ddot{x} \dot{y} - \ddot{y} \dot{x})}{\Upsilon_x^2 + \Upsilon_y^2}, \\
  \tau_u &= \frac{\dot{x} \Upsilon_x + \dot{y} \Upsilon_y}{\sqrt{\Upsilon_x^2 + \Upsilon_y^2}}, \\
  \tau_r &= \frac{y^{(4)} \Upsilon_x - x^{(4)} \Upsilon_y + \beta (y^{(3)} \ddot{x} - x^{(3)} \ddot{y}) - \beta^2 (\dot{x} \ddot{y} - \dot{y} \ddot{x})}{\Upsilon_x^2 + \Upsilon_y^2} - 2 \frac{[\Upsilon_x (x^{(3)} + \beta \ddot{y}) + \Upsilon_y (y^{(3)} + \beta \ddot{x})]}{\Upsilon_x^2 + \Upsilon_y^2} \frac{r}{\Upsilon_x^2 + \Upsilon_y^2},
\end{align*}
\]

where the typo in the expression for \( r \) provided in [41] is corrected.

The previous expressions provide the necessary tools to compute the energy of the hovercraft simplified system. Therefore, the cost function (5.1) is given by

\[
E = \int_0^T |\tau_1 u| + |\tau_3 r| \ dt,
\]

since \( \tau_2 = 0 \).
5.1.4 Medusa model

The Medusa model can be seen as the generalized underactuated ship, with no side thrust propulsion. The kinematic model is given by (5.5) and the dynamics by (5.6). In the Medusa vehicle, \( m_{11} = 5 \, \text{kg} \) and \( m_{22} = 33.325 \, \text{kg} \). The fact that these masses are different, cause the proof of differential flatness to be extremely complex. No closed-form solutions were found during calculations to try to isolate the states and input.

This does not mean directly that the system is not differentially flat. In order to prove the system is not differential flat, one would have to prove that static feedback linearization is not possible under any diffeomorphism for the control input, which is beyond the scope of the thesis. For more details on how to prove non-flatness for multi-input systems the reader is referred to [15].

However, several sources indicate that the generalized underactuated ship with no side thrust model is not differentially flat [41], [45]. For that reason, optimization problems for Medusa model cannot be solved using polynomial methods. The main difficulty lies in the calculation of the heading angle. Since differential flatness requires that all states and inputs can be expressed as functions of the flat output and its derivatives, most underactuated systems are indeed not differentially flat since they usually involve more complex relations in the model.

There are approximately related flatness properties, namely, partial differential flatness, [46] and Liouvillian systems [34], that manage to take advantage of almost differentially flat systems, however, this model does not fall into either category.

This problem was found to be infeasible to solve using polynomial methods, therefore, the thesis will be centered in developing the best motion planning algorithm for the drone and hovercraft case.

Since Medusa model falls into the category of a general surface vessel, and the literature seems to indicate, although not in a clear manner, that the above model is not differentially flat, other algorithms have to be considered. One idea would be to use direct collocation which would, theoretically, be the most suitable optimization methods to address this problem in a similar manner, by using splines composed by piecewise Bernstein polynomials and then using the “minimum distance between two Bézier curves” algorithm to prevent collisions.

5.2 Initialization

One very important part of every optimization problem, especially the non-convex ones, is the initialization. We want to guarantee the solver starts iterating as close as possible to the desired solution. Doing the right initialization can dictate, in non-convex optimization, if the solver is able to find the global minimum or converges to local minima.

5.2.1 Hungarian assignment

The Hungarian assignment (or Munkres’ assignment algorithm [47]) is a fast optimization algorithm which finds the best assignment possible for a certain number of identities, provided that these identities
do not have the same assignment.

This is helpful in the case where the desired final positions are predefined but not assigned to specific vehicles. This is called the unlabeled case, whereas in the labeled case the robots have a fixed goal assignment [14].

By choosing the cost function of the Hungarian assignment as the minimum distance between initial and final positions, one can prevent several the occurrence of unnecessary potential collisions between vehicles, especially in the go-to-formation maneuver, where typically the initial set of positions and the final set of positions are both distributed approximately in straight lines, that are parallel to each other. An example of such a problem setting is shown in the results chapter (Chapter 6).

To implement the Hungarian assignment, first one needs to define a cost matrix in which the rows correspond to each vehicle and the columns to each assignment. The numbers in the cost matrix will be the distances of the initial positions of each vehicle to each possible final position.

Filling the elements of the cost matrix in this particular form, the nearest final positions will be readily assigned after running the corresponding linear optimization algorithm. For a clear step-by-step explanation, the reader is referred to [47].

5.2.2 Fixed coefficients: Initial and final constraints

Recall that the vehicles’ positions, i.e. the flat outputs, are defined as Bézier polynomials and consequently completely described by the coefficients or control points. Then, imposing constraints in initial and final position, velocity and acceleration or orientation, this means that the first 4 coefficients and the last 4 coefficients of the Bézier curve that parameterizes the trajectory are defined for the hovercraft model and the first and last 3 coefficients are defined for the unicycle model. In this section, the explicit computations of these coefficients will be detailed.

Position

The initial and final position is simply given by the first and last coefficients of each vehicle (see Property 1). Defining the coefficients for the position as \( c(j, v) \in \mathbb{R}^2 \), where \( j \) is the coefficient number \( j \in 0, ..., n \), and \( n \) is the order of the curve which is predefined. Moreover, \( x \) and \( y \) are the coordinates of the coefficient in the x-axis and y-axis respectively, and \( v \) is the vehicle number in a total of vehicles defined by \( V \). The initial coefficients are, therefore, given by

\[
c(1, v) = (x_i(v), y_i(v)), \quad \forall v \in \{1, ..., V\},
\]

with \((x_i(v), y_i(v))\) being the initial condition for the vehicle \( v \).

Similarly, for the final position constraint

\[
c(n, v) = (x_f(v), y_f(v)), \quad \forall v \in \{1, ..., V\},
\]

with \((x_f(v), y_f(v))\) being the final condition for the vehicle \( v \).
To simplify the notation, the $\forall v \in \{1, ..., V\}$ will be omitted in the following expressions of this section.

**Linear velocity**

To satisfy the initial and final velocity conditions for Bézier curves specified in Property 3 the coefficients must be such that

$$c(2, v) = c(1, v) + (\ddot{x}_i(v), \ddot{y}_i(v)) \frac{T}{n}.$$  

and

$$c(n - 1, v) = c(n, v) - (\ddot{x}_f(v), \ddot{y}_f(v)) \frac{T}{n}.$$  

which also fixes the values of $c(2, k)$ and $c(n - 1, k)$.

**Linear acceleration**

Similarly, if the initial and final accelerations are also constrained, then the values of $c(3, k)$ and $c(n - 2, k)$ are also fixed by direct application of Property 3 to the second derivative of the Bézier curve, yielding

$$c(3, v) = (\dddot{x}_i(v), \dddot{y}_i(v)) \frac{T^2}{n(n - 1)} + 2c(2, v) - c(1, v),$$  

and

$$c(n - 2, v) = (\dddot{x}_f(v), \dddot{y}_f(v)) \frac{T^2}{n(n - 1)} + 2c(n - 1, v) - c(n, v).$$

**Derivative of linear acceleration**

Applying the same derivation rule for the acceleration, the initial derivative of acceleration defines

$$c(4, v) = (x_i^{(3)}(v), y_i^{(3)}(v)) \frac{T^3}{n(n - 1)(n - 2)} + 3c(3, v) - 3c(2, v) + c(1, v),$$  

and the final acceleration

$$c(N - 3, v) = -(x_f^{(3)}(v), y_f^{(3)}(v)) \frac{T^3}{n(n - 1)(n - 2)} + 3c(n - 2, v) - 3c(n - 3, v) + c(n, v)).$$

**Orientation**

Following the hovercraft model in Section 5.1.3, the tangent of the orientation is given by

$$\tan \psi = \frac{\ddot{y} - \beta \ddot{y}}{\ddot{x} - \beta \ddot{x}}.$$  

This means that the orientation is already defined if linear velocity and acceleration initial constraints are defined, i.e., there has to be a choice of which initial constraints have to be defined. In this thesis perspective, it is justifiable that the initial and final orientation constraints are the ones to be followed.

Since finding the set accelerations which lead to a desired orientation is an undetermined problem, and transforming this equation into linear equality constraints proved to lead to very slow computations.
of the optimization problem, an acceleration $\ddot{x}$ was chosen, and then $\ddot{y}$ is calculated accordingly. This $\ddot{x}$ is chosen to be the lowest possible that does not lead to numerical problems.

For the unicycle model (Section 5.1.2) the orientation is given by

$$\tan \psi = \frac{\ddot{y}}{\ddot{x}},$$

so there needs to be a compromise between the initial velocity wish and orientation wish. A similar reasoning can be applied as before, by choosing a very low velocity in $\dot{x}$ and then calculating the correspondent $\dot{y}$ that leads to a desired orientation.

**Angular velocity**

It is also important to have a defined initial and final angular velocity. In the hovercraft model,

$$r = \frac{y^{(3)}Y_{\dot{x}} - x^{(3)}Y_{\dot{y}} - \beta^2(\dot{x}\ddot{y} - \dot{y}\ddot{x})}{Y_{\dot{x}}^2 + Y_{\dot{y}}^2}.$$

To define an initial angular velocity, a similar reasoning is applied. A very low $x^{(3)}$ is defined, and a correspondent $y^{(3)}$ is calculated. This also permits the initial and final parts of the trajectory to be even smoother.

For the unicycle model,

$$r = \frac{x\ddot{y} - \dot{x}\ddot{y}}{x^2 + y^2}.$$

The unicycle acceleration is chosen similarly to the previous cases.

**Feasibility**

There are cases where the chosen $x$ derivative leads to unfeasible fixed coefficients. After every coefficient is completed, in case of non feasibility, the algorithm tries to find the closest points, by iteratively making small changes to the coefficients, until they fulfill every condition. There are also several checks made on the inputs that detect beforehand if the problem is physically possible, solely using the input data.

**5.2.3 Bounds**

It is possible to restrict the values of the coefficients of position, linear velocity and linear acceleration by using the convex hull property. Another great advantage of this method is that the constraints are linear, which permits to add these bounds without losing computational speed. Moreover, the constraints are guaranteed to always be satisfied between samples. However, the feasibility area is restricted which can prevent the program from finding the optimal solution or not being able to find any solution at all. Besides this fact, in most practical cases under consideration, the vehicle is prevented to reach near its maximum capabilities.
Since in the unicycle model and in the hovercraft model, the angular velocity and acceleration are not Bézier curves or cannot be put in rational Bézier form, the constraints have to be done using sampling and are nonlinear. Several numerical issues had to be dealt with, especially numerical issues regarding these expressions in case of very small denominators. This can prevent the problem from ever finding a solution, or even converge to a poorly planned trajectory, that when using finer sampling times is discovered to be unfeasible. In the program, a finer sampling time was used for the nonlinear constraints of the angular-related bounds.

5.2.4 Additional comments

The inputs needed to run the program are: the time horizon \( T \), the sampling period \( T_s \), the order of the Bernstein polynomial \( n \), the radius of the circumference that surrounds the vehicle, the maximum and minimum values that the vehicle can endure for position, linear velocity, angular velocity, linear acceleration and angular acceleration, as well as initial and final desired values for position, heading angle linear velocity and angular velocity. Moreover, physical constants are needed such as the vehicle’s mass, moment of inertia about the z-axis and drag term and finally, optimization related constants such as tolerances of the radius, maximum number of iterations and minimum step size, and also a well defined, differentially flat, vehicle model.

In optimization problems, if one is able to specify the gradient and Hessian of the cost function and nonlinear constraints, chances are that the problem will be solved in a significantly reduced amount of time. This is due to the fact that the solver spends a great deal of time calculating the gradient and approximating the problem by Hessians that are as sparse as possible. In this model, however, the Gradient and the Hessian are related with the coefficients. Even though the system is differentially flat, explicitly finding how the energy depends on each coefficient is a very complex problem to solve, and depends on the number of coefficients. Since the implementation needs at least 8 fixed coefficients for the hovercraft and 4 fixed coefficients for the unicycle it is actually much faster to let the optimizer calculate the gradient by finite differences and then using BFGS to calculate the Hessian (see Section C.3.1).

5.3 Flow chart

In Figure 5.2, a flow chart of the program’s general layout is presented. Although it is a very simplified scheme it allows to understand the general structure of the algorithm.
Figure 5.2: Program layout
Chapter 6

Results

In this chapter, the results for the developed motion planning implementation are shown. This algorithm optimizes Bézier control points for trajectories of $N_v$ vehicles and $N_o$ obstacles, using MATLAB’s built-in function `fmincon`. The experiments will consider randomly generated initial and final conditions, using a constant seed, for comparison when utilizing Hungarian assignment. The results were obtained using a MacBook Air, early 2014, Intel Core i7, dual-core, with Turbo Boost until 3.3 GHz.

6.1 Hovercraft

For the hovercraft results, the feasible area is similar to the one next to Oceanário de Lisboa, where ISR marine vehicles are usually tested. This area will be changed, in case there are too many vehicles to fit laterally in the simulation. The random initial and final conditions are generated such that in the beginning the $y$ coordinate is around $20 \pm 15$ [m] and in the end it is exactly at $200$ [m]. For the $x$ coordinate, the values are randomly assigned inside the position bounds such that the vehicles are close to each other, and the probability of collision increases. The initial orientation is randomly set using normal distribution around $90^\circ$ and the final orientation is set to be $90^\circ$. All these parameters are very easily changeable in the program.

All the constant inputs considered for the hovercraft are shown in Table 6.1 and are close to the Medusa vehicle specifications.

| $x_{\text{min}}$ | $x_{\text{max}}$ | $y_{\text{min}}$ | $y_{\text{max}}$ | $|v_{\text{max}}|$ | $|\tau_{u,\text{max}}|$ | $|r_{\text{max}}|$ | $|\tau_{r,\text{max}}|$ | $r_v$ | mass |
|---------------|---------------|---------------|---------------|---------------|----------------|---------------|---------------|-------|-------|
| 0 m           | 200 m         | 0 m           | 340 m         | 1.5 m/s       | 30 N           | 0.375 rad/s   | 22.522 Nm     | 0.5175 m | 30 kg |

In Table 6.1, the four rows are the bounds on position, $|v_{\text{max}}|$, $|\tau_{u,\text{max}}|$, $|r_{\text{max}}|$ and $|\tau_{r,\text{max}}|$ represent the maximum absolute values for linear velocity, force, angular velocity and torque, respectively and $r_v$ is the radius of the hovercraft.

This set of results will test the optimization algorithm with the best optimization related settings, in
terms of computational time and minimum cost in function of the number of vehicles.

A significant number of experiments can be performed to test an optimization algorithm, by showing how it behaves when changing the parameters. To simplify the discussion of these results, only the most important and relevant experiments will be performed.

The algorithm will be tested first using the Hungarian assignment algorithm and then without, for a case where the bounds and constraints are easily satisfied, even though it corresponds to a scenario which is less likely to be used in real situations. This followed by another more practical case that also tests how the algorithm reacts when the constraints are difficult to satisfy. The time horizon for the first experiment will be $T = 4000$ s such that the optimal solution is well inside the feasible region, even for a high number of vehicles $N_v$, where the map is enlarged and the trajectories become bigger. These conditions also allow the program to run faster than when using more conservative conditions.

In this section, the algorithm used for finding the global minimum is interior-point, with conjugate gradient as sub-algorithm and BFGS as the algorithm to compute the approximate Hessian.

**Labeled case**

The results for the hovercraft model, in the labeled case (no Hungarian assignment allowed) are shown in Table 6.2. In these simulations, no obstacles were considered.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.430049</td>
<td>0.17106</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.847811</td>
<td>0.37467</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>3</td>
<td>1.526329</td>
<td>0.57844</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>4</td>
<td>1.776419</td>
<td>0.93086</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>5</td>
<td>3.689143</td>
<td>0.71891</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>6</td>
<td>11.321944</td>
<td>0.72025</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>7</td>
<td>4.030018</td>
<td>1.3323</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>8</td>
<td>6.005675</td>
<td>1.3178</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>9</td>
<td>3.561763</td>
<td>1.8528</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>10</td>
<td>6.997030</td>
<td>2.5544</td>
<td>0</td>
<td>11</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>11</td>
<td>12.769750</td>
<td>3.7844</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>12</td>
<td>30.619549</td>
<td>1.5696</td>
<td>0</td>
<td>13</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>13</td>
<td>16.803622</td>
<td>2.1231</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>14</td>
<td>32.580818</td>
<td>1.8578</td>
<td>0</td>
<td>11</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>15</td>
<td>45.536394</td>
<td>2.2932</td>
<td>0</td>
<td>11</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>16</td>
<td>31.943784</td>
<td>4.0387</td>
<td>0</td>
<td>11</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>17</td>
<td>43.002383</td>
<td>5.7108</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>18</td>
<td>32.166956</td>
<td>5.4084</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>19</td>
<td>91.492071</td>
<td>4.1154</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>20</td>
<td>136.617451</td>
<td>3.7683</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>21</td>
<td>91.619547</td>
<td>3.5572</td>
<td>0</td>
<td>14</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>25</td>
<td>96.624743</td>
<td>11.4829</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>30</td>
<td>565.375054</td>
<td>7.1749</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
</tbody>
</table>
These results show little amounts of energy consumed in each trajectory. This is explained by the large time horizon $T$ and also because in the hovercraft model, only the lateral drag $d_{22}$ is considered, and the drag in the other coordinates is supposedly readily compensated using a controller, which simplifies the model. Moreover, this energy is related to the motion, and not to the motor or the batteries of the each vehicle, which further justifies the low values of the simulation.

- **Rerun**: the number of failures of the algorithm. If the final result is not feasible, the algorithm is ran again using finer sampling and smaller step, as well as a higher tolerance for the distance between samples. After a certain number of reruns, chosen by the user, optimization variables are also reinitialized using random values increasingly far from the straight line trajectory, chosen as initial guess.

- **N**: increasing the number of control points increases the feasibility region and the ability for the algorithm to perform more precise changes in the trajectory. However, this increases complexity and can lead the algorithm to converge to poor local minima. The minimum N used in the program is 10. Nevertheless, using higher N is more convenient for more complicated problems and usually leads to faster and better results.

- **Minimum step**: in the presented results, the algorithm stopped when a step made in the cost function was smaller than the minimum step. For a small step, the algorithm takes longer but achieves better results. This parameter represents the trade-off between fast and better results and it corresponds to the step tolerance parameter in fmincon. It is important to note that the minimizer can also stop due to other reasons such as a maximum number of iterations, but that number was set to be significantly large, and was never the cause for the algorithm to stop, in any presented result.

- **$T_s$**: The sampling time represents the trade-off between fast iterations and certainty that the result is feasible. It is particularly important to make sure that the values of $r$ and $r_\tau$ stay inside the bounds since these variables can have very high peaks, thus requiring small sampling times.

Results for the trajectory with 3 vehicles are shown in Figures 6.1 and 6.2.

Figure 6.1 shows that the optimization algorithm successfully prevented all collisions without going to the unfeasible area depicted in grey, and also managed to align the vehicles in the final positions with the desired orientation. The vehicles’ initial and final positions are represented by the colored dots in the edges of the trajectories. The vehicle starts in the bottom and goes up. The time frame being depicted is at $t = 2400$ s. The grey area is the fixed unfeasible region.

In Figure 6.2 it is clear that the physical variables respect their bounds and that initial and final constraints are respected as well. Properties of the Bézier curves are also evident, such as the initial and final points, convex hull and the differentiation rule. Moreover, the heading angle evolution towards $90^\circ$ is also evident. The evolution of the angular velocity and torque follow the orientation's behaviour. The power is divided in two halves: the first half corresponds to the power that the vehicle uses during its acceleration whereas the second half corresponds to the vehicles’ braking power applied to stop the
vehicle in the final position. It can also be seen that the values of the physical variables are very far from
the maximum specifications, which is excepted since $T = 4000 \text{ s}$ is a very long time to travel such a
small distance.

To test how the algorithm reacts when having a small feasibility region, in a more realistic case,
consider $T = 400 \text{ s}$. In this case, control points of the velocity have to be near the maximum value for
the vehicles to reach the end destination in time. Using only few coefficients will not allow the vehicle to
reach the speed it needs. In fact, the only feasible results found were using $N \geq 14$. The simulation using
$N=14$ is shown in Figures 6.3 and 6.4. The algorithm took 15.09 seconds and the energy of the optimal
trajectory is 40.6546 Joule, using 2 seconds of sampling time and minimum step of 0.05 Joule.

In Figure 6.3, it is possible to see the vehicles’ trajectories adapt to the presence of the obstacle and
the position bounds. It is important to note that these position bounds were already being considered in
Figure 6.1, although there were not shown in the figure.

In Figure 6.4, it can be observed that the control points for $x$ are on the frontier of the minimum
values since the region where the vehicle in the final left position is very tight and requires the trajectory
to approach the unfeasible region, to escape the moving obstacle. Moreover, as mentioned above,
since the time horizon is short, the velocity control points of $v_y$ are pushed to a maximum in some cases.
Although it is not completely clear in the figure, it is possible to see several control points reaching
maximum absolute values. Another important remark is the spike of the power of 2 vehicles. Although it
is not clear in the figure, the blue and yellow lines in the power plot correspond to the vehicles that have
blue and yellow initial and final position dots in Figure 6.3. In order for these vehicles to reach the end
point in time, they have to perform a trajectory which is not oriented with the final heading angle. This
causes the algorithm to only align the vehicles in the end of their trajectory. Since the time horizon is
small, this has to be done rapidly, causing the power spike.

**Unlabeled case**

In the unlabeled case, the set of final positions does not require to be specifically occupied by a certain
vehicle. Thus, it can be optimally chosen using Hungarian assignment described in Section 5.2.1. Going
Figure 6.2: Time evolution of all the physical variables for each hovercraft in the $T = 4000$ s case.

back to the case with $T = 4000$ s time horizon, the results for run time and energy are presented in Table 6.3.

Using the Hungarian assignment, the problem becomes very simple. The initial guess is very close to the optimal solution, which shows how relevant it is to successfully solve an optimization problem with a good initialization. Since collisions are difficult to happen and the time horizon is large, the algorithm only required very few iterations until it stopped. For a very high number of vehicles, when the step
becomes 0.5, the algorithm is practically testing the feasibility and exiting after 2 to 5 iterations. The sampling time had to be enlarged due to memory issues, which also prevented to take results for a higher number of vehicles.

### 6.2 Unicycle

To present the results for the unicycle model, and depict the effects of different models and specifications, a simulation is performed in a similar setting as the $T = 400 \text{ s}$ hovercraft experiment, with two obstacles, shown in Figure 6.3 and Figure 6.4. The unicycle model follows approximately the specifications of the DJI M100 quadrotor. The constant inputs used for the simulations are presented in Table 6.4.

The simulation using $N=10$ is shown in Figures 6.5, and 6.6. The algorithm took 10.8 seconds and the energy of the optimal trajectory is 13.3599 Joule, using 2 seconds of sampling time and minimum step of 0.05 Joule.

The resulting trajectory presented in Figure 6.5, is substantially different from the one in Figure 6.3. In this case, the trajectories first adapt to an orientation towards which they can accelerate to avoid the obstacle. In particular, the vehicle with the yellow initial and end points has to accelerate such that it does not crash with the dark blue dot vehicle, and afterwards accelerate even further to escape the moving obstacle. This causes the vehicle, in a environment without drag, to have to make an effort to stop. The final orientations are also achieved right in the end of the trajectory.

In Figure 6.6, there are two spikes in the power and angular variables in the beginning and end of the trajectory. This is because the vehicle is first oriented in an optimal manner, to then accelerate in the optimal direction. This also shows another feature of using Bézier curves to force smooth trajectories.
Since the acceleration is also a Bézier curve, the acceleration curve will be smooth as well. However, the angular variables are not Bézier curves, thus the optimizer is able to produce relatively aggressive behaviour in those variables.

Figure 6.4: Time evolution of all the physical variables for each hovercraft in the $T = 400$ s case.
Table 6.3: Run time and energy of the obtained trajectories with Hungarian assignment with $T = 4000$ s.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.026360</td>
<td>0.17561</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>3</td>
<td>1.802638</td>
<td>0.39227</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>4</td>
<td>1.144785</td>
<td>0.76124</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>5</td>
<td>1.456218</td>
<td>0.94336</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>10</td>
<td>1.987222</td>
<td>1.8252</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>20</td>
<td>12.198509</td>
<td>4.0352</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>30</td>
<td>4.172632</td>
<td>5.7047</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>40</td>
<td>6.892830</td>
<td>7.5583</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.5</td>
</tr>
<tr>
<td>50</td>
<td>8.870660</td>
<td>9.2109</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.5</td>
</tr>
<tr>
<td>60</td>
<td>13.548060</td>
<td>11.0736</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.5</td>
</tr>
<tr>
<td>70</td>
<td>20.353896</td>
<td>12.9385</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.5</td>
</tr>
<tr>
<td>80</td>
<td>33.471470</td>
<td>14.8263</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.5</td>
</tr>
<tr>
<td>90</td>
<td>32.741575</td>
<td>16.7778</td>
<td>0</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>100</td>
<td>36.610692</td>
<td>18.572</td>
<td>0</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>110</td>
<td>47.840710</td>
<td>20.3926</td>
<td>0</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>120</td>
<td>59.570166</td>
<td>22.2787</td>
<td>0</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>130</td>
<td>74.105587</td>
<td>24.1101</td>
<td>0</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>140</td>
<td>96.197148</td>
<td>25.8842</td>
<td>0</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>150</td>
<td>119.37465</td>
<td>27.639</td>
<td>0</td>
<td>10</td>
<td>300</td>
<td>0.5</td>
</tr>
<tr>
<td>160</td>
<td>127.750465</td>
<td>29.4919</td>
<td>0</td>
<td>10</td>
<td>300</td>
<td>0.5</td>
</tr>
<tr>
<td>170</td>
<td>159.107556</td>
<td>31.3994</td>
<td>0</td>
<td>10</td>
<td>300</td>
<td>0.5</td>
</tr>
<tr>
<td>180</td>
<td>187.347948</td>
<td>33.1223</td>
<td>0</td>
<td>10</td>
<td>300</td>
<td>0.5</td>
</tr>
<tr>
<td>190</td>
<td>205.232390</td>
<td>35.0099</td>
<td>0</td>
<td>10</td>
<td>400</td>
<td>0.5</td>
</tr>
<tr>
<td>200</td>
<td>230.095500</td>
<td>36.8654</td>
<td>0</td>
<td>10</td>
<td>400</td>
<td>0.5</td>
</tr>
<tr>
<td>300</td>
<td>826.611416</td>
<td>55.3213</td>
<td>0</td>
<td>10</td>
<td>400</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 6.4: Constant inputs used for unicycle simulations.

| $x_{min}$ | $x_{max}$ | $y_{min}$ | $y_{max}$ | $|v_{max}|$ | $|\tau_{u,max}|$ | $|r_{max}|$ | $|\tau_{r,max}|$ | $r_v$ | Mass |
|-----------|-----------|-----------|-----------|------------|--------------|----------|--------------|------|------|
| 0 m       | 200 m     | 0 m       | 340 m     | 22 m/s     | 5 N          | 2.618 rad/s| 5 Nm         | 0.325 m | 2.355 kg |

6.3 **Comparing parameters of optimization in MATLAB**

The parameter that most impacts the results is the minimizer algorithm. The previous results were obtained using interior-point method with conjugate gradient. In this section, results are shown for SQP and interior-point using factorization as the sub-algorithm. The results using the same conditions as the ones used to obtain Table 6.2 are shown in Table 6.5.

It can be seen that SQP is similar, if not slightly better than interior point for small problems but, for problems with higher complexity, it starts to take too long. This is not the case with interior point, where the algorithm rapidly converges to an unfeasible result and tries again using another initialization.

Now trying the same experiment but using interior-point algorithm with factorization as the sub-algorithm (Section 4.2.2) instead of conjugate gradient. The results are shown in Table 6.6.
Figure 6.5: Optimal trajectory for 3 unicycles and 2 obstacles. The vehicles’ initial and final positions are represented by the colored dots in the edges of the trajectories. The vehicle starts in the bottom and goes up. The large dark blue circles represent two obstacles. The one in the bottom is static and the one in the top is moving. The time frame being depicted is at $t = 208$ s. The grey area is the fixed unfeasible region.

Table 6.5: Run time and energy of the obtained trajectories using SQP without Hungarian assignment with $T = 4000$ s.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.495865</td>
<td>0.17106</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>2</td>
<td>0.788077</td>
<td>0.37467</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>3</td>
<td>0.949975</td>
<td>0.57186</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>4</td>
<td>1.036382</td>
<td>0.78313</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>5</td>
<td>1.224354</td>
<td>0.98696</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>6</td>
<td>1.471091</td>
<td>1.155</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>7</td>
<td>1.362540</td>
<td>1.4298</td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>8</td>
<td>1.301563</td>
<td>1.8726</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>9</td>
<td>3.946344</td>
<td>2.1698</td>
<td>0</td>
<td>11</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>10</td>
<td>2.081851</td>
<td>2.7392</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>11</td>
<td>7.154099</td>
<td>2.7929</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>12</td>
<td>29.351421</td>
<td>2.0016</td>
<td>0</td>
<td>13</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>13</td>
<td>3.926688</td>
<td>2.6612</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>14</td>
<td>12.844787</td>
<td>3.1679</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>15</td>
<td>137.834869</td>
<td>3.7135</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>16</td>
<td>426.041991</td>
<td>2.4115</td>
<td>2</td>
<td>15</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>17</td>
<td>214.135999</td>
<td>6.1396</td>
<td>2</td>
<td>15</td>
<td>100</td>
<td>0.005</td>
</tr>
</tbody>
</table>

As it can be seen, using factorization does not seem to be a good sub-algorithm. Although it might lead to more informed steps, the algorithm becomes slower and it actually achieves worse results when compared to the SQP and the interior-point method with conjugate gradient as the sub-algorithm.

Figure 6.7 facilitates the comparison of the three minimizers and displays the results in tables 6.2, 6.5 and 6.6. This indicates that the best method to use for this problem is interior-point with conjugate gradient sub-algorithm.
Figure 6.6: Time evolution of all the physical variables for each unicycle for the $T = 400$ $s$ case.
Table 6.6: Run time and energy of the obtained trajectories using interior-point with factorization subalgorithm without Hungarian assignment with $T = 4000$ s.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.435164</td>
<td>0.17106</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.686028</td>
<td>0.37467</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>3</td>
<td>2.019625</td>
<td>0.5973</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>4</td>
<td>2.594779</td>
<td>0.81182</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>5</td>
<td>1.643485</td>
<td>1.0284</td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>6</td>
<td>5.387450</td>
<td>1.0771</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>7</td>
<td>6.231528</td>
<td>1.4535</td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>8</td>
<td>170.084744</td>
<td>8.4661</td>
<td>10</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>9</td>
<td>24.422987</td>
<td>8.4946</td>
<td>4</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>10</td>
<td>16.196132</td>
<td>4.8544</td>
<td>2</td>
<td>12</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>11</td>
<td>33.303572</td>
<td>11.6341</td>
<td>3</td>
<td>14</td>
<td>100</td>
<td>0.005</td>
</tr>
<tr>
<td>12</td>
<td>19.375570</td>
<td>10.9453</td>
<td>3</td>
<td>14</td>
<td>100</td>
<td>0.005</td>
</tr>
</tbody>
</table>

Figure 6.7: Run time and minimum cost function obtained by interior-point method using conjugate gradient sub-algorithm, sequential quadratic programming, and interior-point method with direct step (factorization) sub-algorithm, for the simulations using $T = 4000$ s.
Chapter 7

Conclusions

7.1 Achievements

In this work, a fast optimal motion planning algorithm was designed for two AUV models: the unicycle and the hovercraft. The requirements to describe the optimal control problem, and formulate it into an equivalent optimization problem by means of differentially flat systems were presented. It was also presented an energy-related cost function for two differentially flat vehicle models which leads to fast computations of optimal trajectories, even satisfying a significant amount of nonlinear constraints that arise from the inter-vehicle constraints and bounds for the angular velocity and torque. Results also show that the algorithm can handle optimization problems with a significant number of vehicles. Moreover, an extensive study that gathers and compares the main methods used in motion planning optimization is presented. First, an innovative way to parameterize a trajectory is shown, by using Bézier curves to approximate the optimal trajectory. This method has very good properties for motion planning that, together with the differential flatness property greatly simplify the optimization problem. Furthermore, using Bézier curves ensures that no two vehicle collide, which is not the case when using the usual discretized parameterization methods for which inter-sample collision avoidance is not guaranteed. In the future, the algorithm can be easily adapted to a receding horizon implementation, widening the range of application to systems that are not necessarily differentially flat. For example, considering the Medusa vehicle, the hovercraft model could be used as an approximation and then tracking controllers could be used to track the generated trajectories and compensate for the differences in the model dynamics. This algorithm, not only solves the problem for the go-to-formation maneuver, to start cooperative missions, but also allows to add other objectives such as active navigation localization. The tool developed to plan optimal trajectories can now be explored to conduct further research.

7.2 Future Work

To optimize trajectories for vehicles modelled by non-differentially flat systems such as the Medusa, an interesting approach would be to use BB-form splines (see Section 2.3) together with direct collocation method (see Section 3.3.3). Then, one could validate the result by applying a similar algorithm to
the one described in Section 2.4.3, by finding the closest spline segment, and applying the “minimum distance between two Bézier curves algorithm”. Furthermore, splines could reduce the conservatism in the bounds of the physical variables, allowing more feasible space for the algorithm, using the same number of optimization variables.

Additionally, the tests performed with the implementation of the proposed algorithm provide good indication that it is sufficiently fast to be used in real-time applications. This algorithm could be also adapted to run in receding horizon form and directly applied to control vehicles with a differentially flat model, such as drones and most ground robots. Accounting for spiral currents or winds would also be interesting features to consider and explore.
Bibliography


Appendix A

Motivating optimization problem

A.1 Problem Description

The problem in Section 2.1 considers a double integrator setting. This problem is given by

\[ \ddot{x} = u, \]

which can be translated into

\[
\begin{cases}
\dot{x}_1 = x_2 \\
\dot{x}_2 = u
\end{cases}
\]

from where one gets the dynamic system

\[
\begin{bmatrix}
\dot{x}_1 \\
\dot{x}_2
\end{bmatrix} = \begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u.
\]

The general form of the dynamics of a linear system is given by

\[ \dot{x} = Ax + Bu, \]

and the dynamic matrices referred to the state \( x \) are

\[
A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad D = [0].
\]

The goal is to stop a vehicle, as soon as possible, in a given position, starting from a certain initial state, while penalizing the input force needed for the considered trajectory, which indirectly accounts for the energy spent by the vehicle. Firstly, this was achieved by simply considering the unconstrained optimization problem.
minimize \( u(t) \int_0^T x_1(t)^2 + x_2(t)^2 + \rho u(t)^2 \, dt \),

where \( T \) is a given time horizon, \( x_1(t) \) and \( x_2(t) \) are the states (speed and acceleration, respectively), \( u(t) \) is the control input signal (force) and \( \rho \) is the weighting factor that represents the balance of the wish of using less energy, and the wish to stop the vehicle in minimal time.

This translates into the following cost function

\[ J(u) = \int_0^T x_1(t)^2 + x_2(t)^2 + \rho u(t)^2 \, dt, \]

which, in discrete form is

\[ J_d(u) = \sum_{k=0}^N x_1^2_k + x_2^2_k + \rho u_k^2. \]

There are several ways to approach this problem, such as direct methods, indirect methods (analytic approach) and polynomial methods.

### A.2 Analytic Approach

Cost function (A.1) is in the form of a linear-quadratic problem with infinite horizon but only considered until time \( T \)

\[ J = \int_0^\infty x^T Q x + u^T R u + 2x^T N u \, dt, \quad (A.1) \]

where

\[ x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad R = \rho, \quad N = 0. \quad (A.2) \]

The problem is then solved inputting (A.2) in MATLAB, considering the constants needed for the problem \( \rho = 0.1 \), \( x_0 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}^T \), \( T_s = 0.01 \text{ s} \), \( T = 10 \text{ s} \) (final time of simulation) and checking if the following conditions are fulfilled:

- \((A, B)\) is controllable,
- \( R > 0 \) and \( Q \geq 0 \),
- \((C, A)\) is observable,

where \( C \) represents a virtual output that decides the importance of each state in the cost function, given by \( Q = C^T C \), thus, \( C = I_2 \). Ensuring these properties, it is possible to obtain the optimal input and control states using linear-quadratic programming and Riccati’s equation, that has a very fast computation time. The results are depicted in Figure A.1 Despite being an infinite time horizon problem, the vehicle stops in the origin approximately after 5 seconds. For this case, to solve this simplified problem,
the best approach would be to use an analytic approach; however, most problems cannot be put in such a simple form as a linear-quadratic problem, so other methods have to be explored.
Appendix B

Calculation of barycentric coordinates for point-line case

Having a line segment with extremes $A$ and $B$ and a query point $Q$, the closest point to $Q$ in that line segment, denoted by $P$ is given by

\[
\begin{bmatrix}
P_x \\
P_y
\end{bmatrix} = \lambda_1 \begin{bmatrix}
A_x \\
A_y
\end{bmatrix} + \lambda_2 \begin{bmatrix}
B_x \\
B_y
\end{bmatrix}
\]

with $\lambda_1$ and $\lambda_2$ being the two barycentric coordinates. Since $\lambda_1 + \lambda_2 = 1$

\[
\begin{bmatrix}
P_x \\
P_y
\end{bmatrix} = -\lambda_1 \begin{bmatrix}
C_x \\
C_y
\end{bmatrix} + \begin{bmatrix}
B_x \\
B_y
\end{bmatrix}, \quad \begin{bmatrix}
C_x \\
C_y
\end{bmatrix} = \begin{bmatrix}
B_x - A_x \\
B_y - A_y
\end{bmatrix}
\]  
(B.1)

So $C$ is the vector that starts on $A$ and ends on $B$. Since the vector $\overrightarrow{QP}$ is perpendicular to $\overrightarrow{AB}$

\[
\overrightarrow{PQ} \cdot \overrightarrow{AB} = 0, \\
(Q_x - P_x)C_x + (Q_y - P_y)C_y = 0.
\]

Replacing (B.1) in the previous expression

\[
(Q_x + C_x\lambda_1 - B_x)C_x + (Q_y + C_y\lambda_1 - B_y)C_y = 0, \\
\lambda_1 = \frac{C_x(B_x - Q_x) + C_y(B_y - Q_y)}{C_x^2 + C_y^2}, \\
\lambda_1 = \frac{\overrightarrow{AB} \cdot \overrightarrow{QB}}{||\overrightarrow{AB}||^2}.
\]

Doing the same calculations for $\lambda_1 = 1 - \lambda_2$ results in

\[
\lambda_2 = \frac{\overrightarrow{AB} \cdot \overrightarrow{AQ}}{||\overrightarrow{AB}||^2}.
\]
Appendix C

Convex unconstrained optimization

Convex unconstrained optimization problems are written as

\[
\text{minimize}_{x \in \mathbb{R}^n} \quad f(x) \quad (C.1)
\]

where \(x\) is the optimization vector with \(n\) optimization variables, and \(f(x)\) a convex cost function.

Convex unconstrained optimization problems are the fastest category of problems to solve. There are several ways to solve general unconstrained problems that will be presented in the next section as a background for constrained nonlinear optimization.

C.1 Gradient Descent Method

The gradient descent method reduces the optimization problem (C.1) into finding the minimum of the gradient of the convex cost function \(f(x)\). The algorithm is given by

1. choose \(x_0 \in \mathbb{R}^n\) and tolerance \(\epsilon > 0\)
2. \(k = 0\)
3. loop
   4. \(g_k = \nabla f(x_k)\)
   5. if \(||g_k|| < \epsilon\) stop
   6. \(d_k = -g_k\)
   7. find \(\alpha_k > 0\) with the backtracking subroutine
   8. update \(x_{k+1} = x_k + \alpha_k d_k\)
   9. \(k \leftarrow k + 1\)
4. end loop

First, the gradient is computed, then the algorithm makes an educated guess on how further in that gradient direction should the new \(x\) be, such that this new \(x\) is as close to the function's minimum as possible. This is done by computing \(\alpha\) using the backtracking subroutine, described in Appendix C.4. This means that, geometrically, if the step was perfectly weighted, the gradient for the new \(x\) would be
orthogonal to the old one, because if that is not the case, it means that the iteration could have moved the new $x$ closer to the minimum. However, iterating in orthogonal directions does not lead to the fastest way of going from the initial guess to the minimum (Figure C.1). The conjugate gradient solves this issue.

![Figure C.1: Example of iterations in Gradient Descent with optimal step size (green) and Conjugate Gradient Method (red) for minimizing a quadratic function.](image)

**C.2 Conjugate Gradient Method**

The Conjugate Gradient Method [48] improves the chosen direction at each iteration by searching in $n$ conjugate directions, such that the progress made in one direction does not affect the other progress made in the remaining directions. This method is known in mathematics as a way to solve systems of linear equations, particularly those whose matrix is symmetric and semi-positive, as the Hessian matrix $H$ in convex problems.

The generation of $n$ conjugate directions is made by generating $n$ $H$-orthogonal vectors $d = \{d_1, d_2, \ldots, d_n\}$ that have the following property:

$$d_i H d_k = 0, \quad i \neq k, \quad i, k \in \{1, \ldots, n\}. \quad (C.2)$$

Defining $x^*$ as the optimal vector, we want to find a linear combination of $n$ $H$-orthogonal vectors such that

$$x^* = \sum_{i=1}^{n} \alpha_i d_i,$$
where $\alpha_i$ is a weight vector. It is possible to calculate $\alpha_i$ by doing

$$d_k^T H x^* = \sum_{i=1}^{n} \alpha_i d_k^T H d_i,$$

and then apply property (C.2) to get

$$d_i^T H x^* = \alpha_i d_i^T H d_i,$$

$$\alpha_i = \frac{d_i^T H x^*}{d_i^T H d_i}.$$

So, we first need to calculate the $n$ H-orthogonal vectors $d$ and afterwards compute $\alpha_k$.

This is called the Direct Method, which needs the computation of all the $n$ H-orthogonal vectors to directly calculate $\alpha_i$.

One way to compute the $n$ H-orthogonal vectors using the direct method is to find the eigenvectors of $H$

$$Hv = \lambda v$$

which can be computationally inefficient. Actually, most times there is no need to calculate all the $n$ H-orthogonal vectors.

Another way to compute these vectors is by applying the iterative method. Firstly, an initial guess $x_0$ is defined. It can be assumed without loss of generality that $x_0 = 0$, if not, the change of variable $Az = b - Hx_0$ can be considered.

The way to know if the some $x$ is close to the optimal $x^*$ is to assume that $x^*$ is also the minimizer for the quadratic function

$$f(x) = \frac{1}{2} x^T H x - x^T b, \quad x \in \mathbb{R}^n.$$ 

which approximates the actual cost function. After that, $x^*$ is computed by

$$\nabla f(x^*) = Hx^* - b = 0,$$

which is solvable since the problem is convex. This also means that the first guess for the first H-orthogonal vector should be

$$d_1 = -\nabla f(x_1)^T = b - Hx_0.$$ 

The vector $d_1$ can be seen as the residual provided by the initial step of the algorithm. Let $r_k$ be the residual of the $k^{th}$ step, the residual is

$$r_k = b - Hx_k.$$ 

In the gradient descent, $r_k$ would be the direction to follow. But in Conjugate Gradient, the goal is to try and make $d_k$ the conjugate to the other vectors.

One way to do this is to require that the next direction is built on the current residual and all the
previous search directions
\[ d_k = r_k - \sum_{i < k} \frac{d_i^T H r_k}{d_i^T H d_i} d_i \]
so the next iteration will be given by
\[ x_{k+1} = x_k + \alpha_k d_k \]
with
\[ \alpha_k = \frac{b_k^T (b - H x_k)}{b_k^T H b_k} = \frac{d_k^T r_k}{d_k^T H d_k}. \]

This method allows the system to be approximately solved in cases where \( n \) is so large that the direct method would take too long, since in the iterative method there is no need to calculate all the H-orthogonal vectors.

### C.3 Newton method

Similarly to the previous methods, Newton method attempts to find the point \( x \) where the gradient of the cost function \( f(x) \) is zero, converging to that point from a certain initial guess \( x_0 \).

It was first published in 1685 by John Wallis [49] but it was only in 1740 that the researcher Thomas Simpson described Newton’s method as an iterative method and inferred its use for optimization [50].

The method is used to solve the following nonlinear system of equations:

\[
\begin{align*}
\frac{\partial f}{\partial x_1}(x^*) &= 0 \\
\frac{\partial f}{\partial x_2}(x^*) &= 0 \\
&\vdots \\
\frac{\partial f}{\partial x_n}(x^*) &= 0
\end{align*}
\]

where \( x^* \) is the optimal point. Using the Newton method, the iterations come as

\[ x_{k+1} = x_k - H^{-1} \nabla f(x_k). \]

where \( H = \nabla^2 f(x_k) \) is the Hessian of the cost function.

The Newton’s method algorithm for convex optimization is:
1. choose \( x_0 \in \mathbb{R}^n \) and tolerance \( \epsilon > 0 \)
2. \( k = 0 \)
3. loop
4. \( g_k = \nabla f(x_k) \)
5. if \( ||g_k|| < \epsilon \) stop
6. \( d_k = -H^{-1}g_k \)
7. find \( \alpha_k > 0 \) with the backtracking subroutine
8. update \( x_{k+1} = x_k + \alpha_k d_k \)
9. \( k \leftarrow k + 1 \)
10. end loop

Newton method has quadratic convergence [51]. It converges in less iterations than gradient descent methods since it uses more information regarding the cost function, however, to calculate the inverse of the Hessian it takes \( \mathcal{O}(n^3) \) floating point operations per second (FLOPS). Thus, for a high number of optimization variables \( n \), the problem becomes unfeasible.

### C.3.1 Quasi-Newton method

Quasi-Newton methods are used when \( H \) is too heavy to compute or unavailable for the problem. Instead of computing explicitly the Hessian, other methods are used to find an approximate Hessian. These methods vary, with the most popular being: Broyden–Fletcher–Goldfarb–Shanno (BFGS) [37], LBFGS, which is a memory limited version of BFGS, Davidon–Fletcher–Powell (DFP) [52] Symmetric Rank 1 (SR1) [53] or using finite differences of the gradient.

However, since the Hessian is now approximated, information regarding the function’s behaviour becomes less accurate, and the guess for the step is not as good as in the Newton’s method. This leads to an increase of the number of iterations and less precise convergences.

### C.4 Backtracking subroutine

To complete the description of this algorithm, we still need to define a backtracking subroutine to control the step size, which is achieved by computing \( \alpha_k \). Many procedures have been suggested, such as the exact line search where

\[
\alpha_k = \arg\min_{\alpha_k > 0} f(x_k + \alpha_k d_k),
\]

which can take too long to compute. Although it is possible to calculate the step \( \alpha_k \) analytically in some special cases (like the quadratic case), in most cases the methods consist of inexact procedures since the goal is usually to minimize \( f \) as much as possible at each iteration. Several inexact line search methods have been proposed, such as Goldstein [54], Armijo [55], Wolfe [56], Powell [57], Dennis and Schnabel [58], Fletcher [59], Potra and Shi [60], Lamaréchal [61], Moré and Thuente [62], Andrei [63] and many others. In particular, Armijo proposed a simple and efficient algorithm:
1. choose backtracking parameters $\hat{\alpha} > 0$, $0 < \gamma < 0.5$ and $0 < \beta < 1$

2. $\alpha_k = \hat{\alpha}$

3. loop

4. if $f(x_k + \alpha_k d_k) < f(x_k) + \gamma \nabla f(x_k)^T (a_k d_k)$ stop

5. $\alpha_k = \beta \alpha_k$

6. end loop

This backtracking subroutine represents an heuristic for how big the step will be in the negative direction of the gradient. The logic behind the calculation of $\alpha_k$ can be seen as an optimistic approach. It starts with a high step, which means a high $\alpha_k$, given by $\hat{\alpha}$, and then check if taking this wide step towards $d_k$ decreases the cost function. If the leap of faith is too large, which means, the cost function increases, then $\alpha_k$ is reduced by a factor of $\beta$, iterating until a solution is found. However, there is still an added factor in the inequality between old and new cost functions in line 4: $c = \gamma \nabla f(x_k)^T (a_k d_k)$. Since $\gamma > 0$, $\alpha > 0$ and $d_k \nabla f(x_k) = -\nabla^2 f(x_k) < 0$, $c$ is always negative. Thus, $c$ can be seen as a relaxation component, which is proved in [55] to lead the algorithm to convergence. So, it is actually possible that $f(x_k + \alpha_k d_k) \geq f(x_k)$. This lowers the restriction of the algorithm.

C.4.1 Comments

Convex problems are only a small segment in the optimization world of problems, however, a common way to deal with nonconvex problems or just any general problem without checking convexity is to approximate the problem locally by a convex quadratic problem, as seen in Section 4.2.

These methods serve as an introduction for a more complicated type of optimization problems: nonlinear constrained problems.
Appendix D

MATLAB code

D.1 Bézier operations

In this section, the code that implements the properties described in Section 2.2 is presented.

Bernstein basis to monomial basis

```matlab
function m = b2m(b)
% b2m: Converts Bernstein into monomial basis

% b = [0, 1, -2, 3, 4, 5; 1, 20, 3, 4, -5, 6]';

n = size(b,1)-1;
for j = 0:n
    for i = 0:j
        m(j+1,:) = m(j+1,:) + nchoosek(n,j)*nchoosek(j,i)*(-1)^(j-i)*b(i+1,:);
    end
end
% to confirm the result:
% m = flipud(m); % for polyval
% figure, hold on
% tau = 0:0.01:1; bezier = bernsteinMatrix(n,tau)*b;
% if size(b,2) == 2
%     plot(bezier(:,1),bezier(:,2))
%     plot(b(:,1),b(:,2),'o')
%     monomialx = polyval(m(:,1),tau);
%     monomialy = polyval(m(:,2),tau);
%     plot(monomialx,monomialy,'LineWidth',0.5);
% elseif size(b,2) == 1
%     plot(linspace(0,1,length(bezier)),bezier)
%     plot(linspace(0,1,length(b)),b,'o')
%     monomial = polyval(m,tau);
```
Monomial basis to Bernstein basis

```matlab
function b = m2b(m)
% m2b: Converts monomial into Bernstein basis

% m = fliplr([40, -120, 120, -40, 5; 140, -410, 540, -360, 95, 1]);
% m = flipud(m);

n = size(m,1)-1;
b = zeros(n+1,size(m,2));

for j = 0:n
    for i = 0:j
        b(j+1,:) = b(j+1,:)+nchoosek(n-i,j-i)*m(i+1,:);
    end
    b(j+1,:) = b(j+1,:)/nchoosek(n,j);
end

% to confirm the result:
% m = flipud(m); % for polyval
% figure, hold on
% tau = 0:0.01:1; bezier = bernsteinMatrix(size(b,1)-1,tau)*b;
% if size(m,2) == 2
%     plot(bezier(:,1),bezier(:,2))
%     plot(b(1,:),b(2,:),'o')
%     monomialx = polyval(m(1,:),tau);
%     monomialy = polyval(m(2,:),tau);
%     plot(monomialx,monomialy,'LineWidth',0.5);
% elseif size(m,2) == 1
%     plot(linspace(0,1,length(bezier)),bezier)
%     plot(linspace(0,1,length(b)),b,'o')
%     monomial = polyval(m,tau);
%     plot(linspace(0,1,length(monomial)),monomial,'LineWidth',0.5)
% end
end
```

Sum of two Bézier curves

```matlab
function [fg,m] = sumb(f,g)
% sumb: sums two bezier coefs with different order
% returns the coefficients and the order

m = size(f,1)-1;
if size(m,2) == 2
    plot(linspace(0,1,length(bezier)),bezier)
    plot(linspace(0,1,length(b)),b,'o')
    monomial = polyval(m,tau);
else if size(m,2) == 1
    plot(linspace(0,1,length(monomial)),monomial,'LineWidth',0.5)
end
end
```
if m < n, aux = f; f = g; g = aux; m = size(f,1)−1; n = size(g,1)−1; end

fg = zeros(m+1,size(f,2));
for i = 0:m
    sum = zeros(1,size(f,2));
    for j = max([0,i−m+n]):min([n, i])
        sum = sum + nchoosek(n,j)*nchoosek(m−n,i−j)/nchoosek(m,i) * g(j+1,:);
    end
    fg(i+1,:) = f(i+1,:)+sum;
end

% % to confirm the results
% tau = 0:0.01:1;
% bf = bernsteinMatrix(m,tau)*f;
% bg = bernsteinMatrix(n,tau)*g;
% bfg = bernsteinMatrix(m,tau)*fg;
% set(groot,'defaultLineLineWidth',2)
% figure,
% subplot(3,1,1), hold on
% plot(bf(:,1),bf(:,2)), plot(f(:,1),f(:,2),'o')
% subplot(3,1,2), hold on
% plot(bg(:,1),bg(:,2)), plot(g(:,1),g(:,2),'o')
% subplot(3,1,3), hold on
% plot(bfg(:,1),bfg(:,2)), plot(fg(:,1),fg(:,2),'o')
% bfg_num = bf*bg; plot(bfg_num(:,1),bfg_num(:,2))

Product of two Bézier curves

function [fg, order] = pb(f,g)
% pb: computes coefs of the product between two Bézier curves

m = size(f,1)−1;
n = size(g,1)−1;

fg = zeros(m+n+1,size(f,2));
for i = 0:(m+n)
    sum = zeros(1,size(f,2));
    for j = max([0, i−n]):min([m, i])
        sum = sum + nchoosek(m,j)*nchoosek(n,i−j)/nchoosek(m+n,i) * f(j+1,:).* g(i−j+1,:); + sum;
    end
    fg(i+1,:) = sum;
end
order = m+n;

% % to confirm the results
% tau = 0:0.01:1;
% bf = bernsteinMatrix(m,tau)*f;
% bg = bernsteinMatrix(n,tau)*g;
% bfg = bernsteinMatrix(n+m,tau)*fg;
% set(groot,'defaultLineWidth',2)
figure
if size(f,2) == 2
    subplot(3,1,1), hold on
    plot(bf(:,1),bf(:,2)), plot(f(:,1),f(:,2),'o')
    subplot(3,1,2), hold on
    plot(bg(:,1),bg(:,2)), plot(g(:,1),g(:,2),'o')
    subplot(3,1,3), hold on
    plot(bfg(:,1),bfg(:,2)), plot(fg(:,1),fg(:,2),'o')
    bfg_num = bf.*bg;
elseif size(f,2) == 1
    subplot(3,1,1), hold on
    plot(linspace(0,1,length(bf)),bf), plot(linspace(0,1,length(f)),f,'o')
    subplot(3,1,2), hold on
    plot(linspace(0,1,length(bg)),bg), plot(linspace(0,1,length(g)),g,'o')
    subplot(3,1,3), hold on
    plot(linspace(0,1,length(bfg)),bfg), plot(linspace(0,1,length(fg)),fg,'o')
    bfg_num = bf.*bg;
end

D.2 fseminf nonlinear and infinite constraints function

The code used to define the infinite constraints using fseminf described in Section 4.3 is presented below. The inputs xbar, Ts and SEMINF are the optimization variables for the main optimization problem (column with the 2D control points for each vehicle), the optimization variable for the sampling time, and structure containing all the remaining data needed, respectively. The structure SEMINF contains a class named veh, which includes all the functions regarding trajectory calculation and respective coefficients, for each vehicle. The case presented corresponds to the case with 4 vehicles and zero obstacles. It can be seen that the constraints have to be individually defined.

function [c,ceq,K1,K2,K3,K4,K5,K6,Ts] = seminfcon4v0o (xbar,Ts,SEMINF)
% Computes non linear constraints regarding collision detection.

% Initial sampling interval
if isnan(Ts(1,1))
    Ts = zeros(SEMINF.Ncomb,2); Ts(1,: ) = SEMINF.Ts;
end

% sampling set
tau = 0:Ts(1,1):1;
% xbar decomposition

cpos = zeros(SEMINF.N,2,SEMINF.Nv);

for v = 1:SEMINF.Nv
    cpos(:,:,v) = [SEMINF.cfixed(1:3,:,v);...
        xbar((v-1)*SEMINF.Nvar*2+1:2:v*SEMINF.Nvar*2-1),...
        xbar((v-1)*SEMINF.Nvar*2+2:2:v*SEMINF.Nvar*2);...
        SEMINF.cfixed(end-2:end,:,v)];
    SEMINF.veh(v) = SEMINF.veh(v).updateCoefs(cpos(:,:,v)');
    SEMINF.veh(v).updateTraj(bernsteinMatrix(SEMINF.N-1,tau));
end

% taking care of obstacle-vehicle constraints

iv1 = SEMINF.ilvv(:,1);
iv2 = SEMINF.ilvv(:,2);

K1 = SEMINF.eps+SEMINF.safety+...
    SEMINF.veh(iv1(1)).Radius+SEMINF.veh(iv2(1)).Radius−...
    vecnorm(SEMINF.veh(iv1(1)).Trajectory'−SEMINF.veh(iv2(1)).Trajectory');
K2 = SEMINF.eps+SEMINF.safety+...
    SEMINF.veh(iv1(2)).Radius+SEMINF.veh(iv2(2)).Radius−...
    vecnorm(SEMINF.veh(iv1(2)).Trajectory'−SEMINF.veh(iv2(2)).Trajectory');
K3 = SEMINF.eps+SEMINF.safety+...
    SEMINF.veh(iv1(3)).Radius+SEMINF.veh(iv2(3)).Radius−...
    vecnorm(SEMINF.veh(iv1(3)).Trajectory'−SEMINF.veh(iv2(3)).Trajectory');
K4 = SEMINF.eps+SEMINF.safety+...
    SEMINF.veh(iv1(4)).Radius+SEMINF.veh(iv2(4)).Radius−...
    vecnorm(SEMINF.veh(iv1(4)).Trajectory'−SEMINF.veh(iv2(4)).Trajectory');
K5 = SEMINF.eps+SEMINF.safety+...
    SEMINF.veh(iv1(5)).Radius+SEMINF.veh(iv2(5)).Radius−...
    vecnorm(SEMINF.veh(iv1(5)).Trajectory'−SEMINF.veh(iv2(5)).Trajectory');
K6 = SEMINF.eps+SEMINF.safety+...
    SEMINF.veh(iv1(6)).Radius+SEMINF.veh(iv2(6)).Radius−...
    vecnorm(SEMINF.veh(iv1(6)).Trajectory'−SEMINF.veh(iv2(6)).Trajectory');

end

% taking care of obstacle-vehicle constraints

c = []; ceq = [];
end