Motion Planning for Cooperative Autonomous Robots using Optimisation Tools

Thomas David Pamplona Berry

Thesis to obtain the Master of Science Degree in

Electrical And Computer Engineering

Supervisor: Prof. António Manuel dos Santos Pascoal

December 2020

Acknowledgments

I would like to thank my parents and sister for their encouragement, caring and support over all these years, for always being there for me through thick and thin and without whom this project would not be possible.

I would also like to acknowledge my dissertation supervisor, Prof. Pascoal for his insight, support and sharing of knowledge that has made this thesis possible.

Abstract

This thesis presents methods for cooperative motion control for control of fleets of underwater autonomous robots. Good, robust, and fast-to-calculate methods for multiple underwater vehicle motion control is important to take into account inter-vehicle constraints, environmental constraints, energy consumption and mission duration. A suite of programs have been developed using Matlab that generate approximations of optimal trajectories for vehicles in a 2D space.

Keywords

Motion Planning, autonomous vehicles, optimisation, Bezier Curves, Flat systems.

Contents

List of Figures

List of Tables

Listings

Acronyms

ODE	Ordinary Differential Equation
DOF	Degress of Freedom
AUV	Autonomous Underwater Vehicle
TT	Trajectory-Tracking
PF	Path Following
NLP	Non-linear Problem
GJK	Gilbert–Johnson–Keerthi
EPA	Extended Polytope Algorithm
DEPA	Directed Extended Polytope Algorithm
IVP	Initial Value Problem
LQR	Linear Quadratic Regulator

1

Introduction

Contents

1.1 Motivation

Worldwide, there has been growing interest in the use of autonomous vehicles to execute missions of increasing complexity without constant supervision of human operators.

The WiMUST project [1] is an example of a project that demanded the use of such autonomous vehicles. The goal of the WiMUST project was to design a system of cooperating Autonomous Underwater Vehicles (AUVs) able to perform innovative geotechnical surveying operations. Specifically, the WiMUST system consisted of an array of physically disconnected AUVs, acting as intelligent sensing and communicating nodes of a moving acoustic network. Together, the vehicles form a geometry formation which is actively controllable according to the needs of a specific application.

Applications for the AUV formation handled by the WiMUST system include seabed mapping, seafloor characterization and seismic exploration. The overall system behaves as a distributed acoustic array capable of acquiring acoustic data obtained by illuminating the seabed and the ocean sub-bottom with strong acoustic waves sent by one (or more) acoustic sources installed onboard a support ship/boat (see figure 1.1). Advantages of multiple AUVs acting cooperatively as opposed to a single one include robustness against failure of a single node and improving the seabed and sub-bottom resolution. They can also adapt better to unforeseen circumstances in the terrain by making better use of the larger environments that they can observe as the spatial distance between each AUV can be varied.

Another, unrelated, example of application that justifies the use of multiple autonomous vehicles was the Intel show at the 2018 Winter Olympics, where 1200 drones were used to put on a light show in the night sky. Each drone was mounted with a light bulb and, together, formed different shapes in the sky.

Complex systems like the two previously presented have in common the ability to properly plan motion for each vehicle that satisfies certain criteria such as simultaneous arrival of each vehicle in a formation while minimizing spent energy or time and avoiding collisions between vehicles and the environment.

Over the past decades, many approaches to solve motion planning problems have been proposed. Examples include bug algorithms, randomised algorithms such as PRM, RRT, RRT*, cell decomposition methods, graph-based approaches, planners based on learning, and methods based on optimal control formulations. Each technique has different advantages and disadvantages, and is best-suited for certain types of problems. Trajectory generation based on optimal control formulations stands out as particularly suitable for applications that require the trajectories to minimize (or maximize) some cost function while satisfying a complex set of vehicle and problem constraints. Finding closed-form solutions for Optimal Control problems can be difficult or even impossible to solve, and therefore numerical solutions must be sought. A numerical alternative to solve such complex Optimal Control problems consists in optimising trajectories given by Bernstein Polynomials. Recent pioneering work on the use of Bernstein Polynomials [2] for the numerical approximation of these Optimal Control problems show interesting results [3],

see this sentence

namely, they show how can which will be further explored in the work of this thesis.

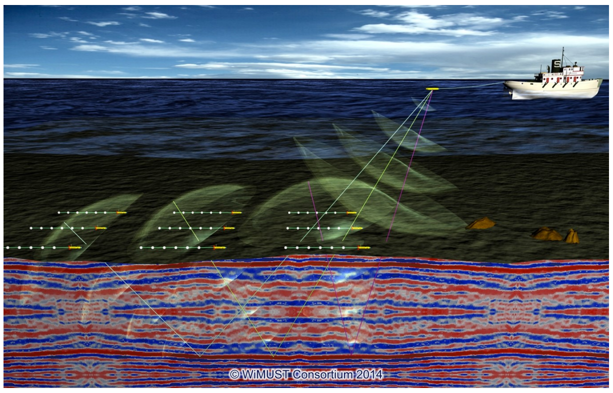


Figure 1.1: Artist's rendition of the WiMUST system for subbottom profiling with source-receiver decoupling

1.2 Background

The work discussed in this thesis focuses on motion planning for multiple cooperative vehicles. Motion Planning, as the name suggests, consists in planning motion for robots, such as mobile vehicles or robotic arms. Trajectory generation based on optimal control formulations will be used, specifically, to solve the motion planning problems.

A trajectory is a time parameterized set of states of a dynamical system. These states can be position, their derivatives, heading, among others. The states and inputs are related to each other by a set of dynamic equations. When dealing with multiple vehicles, the trajectory optimisation problem takes into account the union of states and inputs of each vehicles such that the solution describes trajectories for all of the vehicles simultaneously.

As discussed in the previous section, numerical solutions to the trajectory generation problems must be sought. The numerical methods can be grouped into either direct methods or indirect methods.

Indirect methods "use the necessary conditions of optimality of an infinite problem to derive a boundary value problem in ordinary differential equations", the solutions of which must be found using analytic or numerical methods.

Direct methods, on the other hand, are based on transcribing infinite optimal control problems into finite-dimensional Non-linear Problems (NLPs) using some kind of paramerisation (e.g., polynomial approximation or piecewise constant parameterization). They can be solved using ready-to-use NLP solvers (e.g. MATLAB) and do not require the computation of co-state and adjoint variables as indirect methods do.

The focus on the work of this thesis will be on the usage of Direct methods. Several parameterisation methods will be explored, such as peace-wise constants inputs, polynomials, and the usage of Bernstein polynomials.

When solving an optimisation problem, a model must be chosen for each vehicle.

The states and inputs that describe a vehicle will depend on the choice of model for them. The choice of model to represent the AUVs will affect the dynamic equations that rule them. which rule The Direct methods that will be tested will operate trajectories for be operated on different vehicle models, such as the Medusa Model [4] or a simpler Dubin's car [5].

1.3 Objectives

The work presented in this thesis focuses on the usage of Direct Methods.

There are several direct methods for trajectory optimisation, for example, single and multiple shooting, collocation and quadratic programming. However, polynomial methods based on Bezier curves are particularly advantageous because they have favourable geometric properties which allow the efficient computation of the minimum distance between trajectories. As the complexity of the polynomials increases, the solutions converge to the optimal.

The cost can be constructed based on several criteria such as time and consumed energy. For *cooperative* motion planning, the cost will have to be constructed differently because it will have to take into account the motion of the multiple vehicles at once, in particular, possible inter-vehicle collision.

In practice, some of the objectives of this work include

- test some methods for obstacle avoidance
- compare some different parameterization methodoligies
- analize the complexity of increasing order and number of vehicles
- test viability for non differentially flat systems
- test the usage of log barrier functions may help with speedinng up the optimisation process because they reduce the number of constraints by placing them in the cost function.

1.4 Thesis Outline

In chapter 4.5, an overview of the different available numerical methods for the motion planning for a single vehicle will be presented. exapnd

In chapter 3, an overview of different vehicle models is presented. expand

In chapter 4, a discussion of the code structure is discussed, along with the choice of optimisation algorithms. exapnd

In chapter 5, some results are presented.

exapnd

exapnd

In chapter 6, the conclusion is made.

This will be followed, in chapter , by application examples for a double integrator in 1 and 2 dimensions that capture the dynamics of a single vehicle. In chapter , some considerations for the control of multiple cooperative vehicles will be presented. The report will be concluded with a final overview of the different methods considered and a plan for the project's work will be defined.

2

Existing Trajectory Optimisation Methods for Motion Planning

Contents

In this chapter, direct methods for trajectory optimisation will be reviewed. These methods are generic, i.e., they can be applied to a wide range of dynamic systems. Here, these methods are applied to the control of a vehicle in both one and two dimensions. A good understanding of motion planning for single vehicles will be necessary before extending to multiple vehicles.

2.1 The Optimisation Problem

Motion planning for a single vehicle can be cast in the form of an optimal control problem of the form (see [6])

$$\begin{aligned}
\underset{x(.),u(.)}{\text{minimise}} \quad & J = \int_0^T L(x(t),u(t))dt + \Psi(x(T))\,dt \\
\text{subject to} \quad & x(0) = x_0, \\
& \dot{x} = f(x(t),u(t)), && t \in [0,T] \\
& h(x(t),u(t)) \geq 0, && t \in [0,T] \\
& r(x(T)) = 0
\end{aligned} \tag{2.1}$$

where $x(t)$ are the states, defined as $x : [0,T] \to \mathbb{R}^{n_x}$, $u(t)$ are the inputs, defined as $u : [0,T] \to \mathbb{R}^{n_u}$, $f(x,t)$ is the system of equations for the dynamics, defined as $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$, x_0 is the initial state, $h(x(t),\, u(t))$ represents the path constraints, $r(x(T)) = 0$ the terminal constraints, $L(x(t),u(t))$ is the running cost, defined as $L : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}$, $\Psi(x(t))$ is the terminal cost defined as $\Psi : \mathbb{R}^{n_x} \to \mathbb{R}$ and finally T is known as the time ”horizon”, i. e., no matter what motion planning algorithm is used, the produced control law will only be valid within time $t \in [0;T]$.

An example of a path constraint is to keep the minimum distance to a known obstacle bounded below by a desired safety distance. If state variables 1 and 2 of x represent the position in a 2 dimensional plane, then $h(x)$ could be

$$h(x(t)) = \min_{t\in[0,T]} \|x - p_0\| - d_{\text{min}} \tag{2.2}$$

where p_0 is the location of an obstacle's centre of mass and d_{min} is the minimum accepted distance to that centre.

All of the available Direct Methods used in this Thesis have as objective producing a control law that optimises some form of the above cost function. These Methods will be demonstrated with the control of a double integrator in 1 and 2 dimensions. A double integrator is a linear system where the second derivative of the position variable y, is the acceleration a, that is,

$$\ddot{y} = a \tag{2.3}$$

A general linear system is represented, in state-space form, by

$$\dot{x} = Ax + Bu \tag{2.4}$$

where $x = [x_1, \ldots, x_{n_x}]^T$ is the state of the system and $u = [u_1, \ldots, u_{n_u}]^T$ is the input. For the double integrator, the states and inputs are given by

$$\begin{cases} x_1 = y \\ x_2 = v \\ u = a \end{cases} \tag{2.5}$$

where x_1 and x_2 represent position and velocity, respectively, and u is the acceleration. The system's dynamics then become

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = u \end{cases} \tag{2.6}$$

or

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{2.7}$$

in matrix form, which means matrices A and B of equation 2.4 are given by

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.8}$$

.

The cost function 2.1 will be simplified to

$$J = \int_0^\infty x_1^2 + x_2^2 + \rho u^2 \; dt \tag{2.9}$$

where ρ is a scalar that penalizes "energy" consumption.

This dynamic system is diferentially flat (see section 2.8) with x_1 as the flat output, because it is possible to express x_2 as a derivative of x_1 and u as a double derivative of x_1.

Although this 1-D model is very simple, it can provide insight into real life applications. One easy example is a train what follows a track. It cannot leave the track and so, environmental constraints, such as obstacles, cannot be fixed in time if they are between the train's initial and target position. A non-fixed constraint could be another train that is joining the same track, therefore, the first train cannot be at certain points at certain times.

The optimisation problem (2.1), is infinite dimensional because the solution will be a set of functions of time. Functions over time have an infinite number of points. The following direct methods will replace the infinite number of points of the function by approximated functions which are parameterised by a finite number of variables.

Before presenting the direct methods, results of the application of the Linear Quadratic Regulator are presented. This is important as the results serve as baselines to compare with the direct methods. The Linear Quadratic Regulator (LQR) is a feedback law which guarantees minimal cost for certain kinds of cost functionals.which is not a direct method, it is a feedback law which guarantees minimal cost for certain kinds of cost functionals.

2.2 Linear Quadratic Regulator

A Linear Quadratic Regulator is a technique applicable to linear dynamic systems of the form

$$\dot{x} = Ax + B \tag{2.10}$$

using the same definitions as in section 2.1.

The Linear Quadratic Regulator algorithm yields, under certain conditions, an appropriate state-feedback LQR controller that minimises a cost function of the form

$$J = \int_0^\infty x^T Q x + u^T R u + 2x^T N u dt \tag{2.11}$$

The control law is of the form

$$u = -Kx \tag{2.12}$$

where K is given by

$$K = R^{-1}(B^T P(t) + N^T) \tag{2.13}$$

where $P(t)$ is the solution of the differential Riccati equation [7]

$$A^T P(t) + P(t)A - (P(t)B + N)R^{-1}(B^T P(t) + N^T) + Q = -\overline{P}(t) \tag{2.14}$$

with appropriate boundary conditions.

For the situation where the time horizon T is ∞, $P(t)$ will tend to a constant solution, P, and, as a result, P is found by solving the continuous time algebraic Riccati equation

$$A^T P + PA - (PB + N)R^{-1}(B^T P + N^T) + Q = 0 \tag{2.15}$$

The solution provided by this algorithm will be optimal and unique if the following conditions are fulfilled:

- (A, B) is controllable;

- for a system output $y = Cx$, the pair (A, C) is observable;
- $R > 0$ and $Q > 0$.

Control via LQR is in closed-loop form, which has the advantage of being robust against parameter uncertainty and reducing the effect of external disturbances.

In order to visualise the resulting controlled double integrator evolves over time, the feedback law (2.12) is fed into the system (2.10), non zero initial conditions are provided, and the resulting Initial Value Problem (IVP) is solved. In other words, the objective is to find the soltion to

$$\begin{aligned} &\dot{x} = (A - BK)x \\ &\text{subject to} \quad x(0) = x_0 \end{aligned} \tag{2.16}$$

where x_0 is the initial conditions.

Here an example of control of the double integrator is presented in the context of linear quadratic regulator theory. Here, the initial conditions, which will be the same for all the subsequent 1-dimensional examples throughout this chapter, are given by

$$x_0 = \begin{bmatrix} x_{1_0} \\ x_{2_0} \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \tag{2.17}$$

Figure 2.1 shows a solution of an LQR controlled double integrator. The regulator used for this example was obtained with $Q = I$ and $R = \rho = 0.1$ as weights to obtain K in equation 2.13 so that the resulting cost matches 2.9.

It can be seen that all variables tend asymptotically to zero. It should be noted that after around 6 seconds the system can already be considered stationary.

2.3 Single Shooting

Single Shooting is a direct method for motion planning. It consists of parameterising, for a given system, the control input signal $u(t)$ as a piece-wise constant function of time, as defined by

$$u(t) = q_i, \qquad t \in [t_i, t_{i+1}] \tag{2.18}$$

where q_i has the dimensions of the input, and the intervals $[t_i, t_{i+1}]$ denote the time-intervals between instants indexed by indices i and $i + 1$.

The following step consists of solving the following Ordinary Differential Equation (ODE):

$$\begin{aligned} &x(0) = x_0, \\ &\dot{x}(t) = f(x(t), u(t; q)), \quad t \in [0, T]. \end{aligned} \tag{2.19}$$

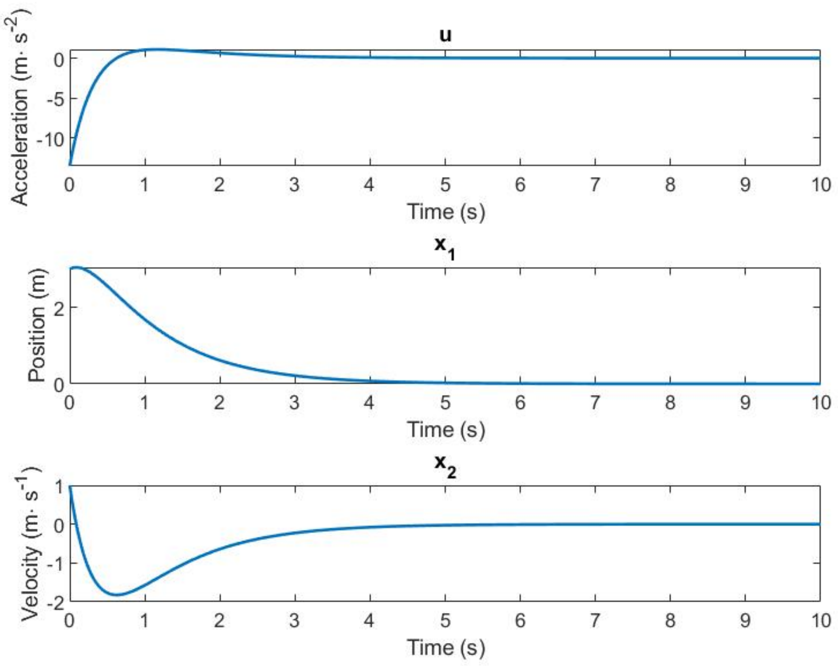


Figure 2.1: Result for a Linear Quadratic Regulator

to obtain the time evolution of x.

With the obtained trajectories, the optimisation problem (2.1) can be extended and expressed as a function of the parameters $q = (q_0, q_1, \dots, q_{N-1})$ as optimisation variables. The new optimisation problem is written in the form

$$\begin{aligned} \underset{q}{\text{minimise}} \quad & \int_0^T L(x(t;q), u(t;q))dt + \Psi(x(T;q))\ dt \\ \text{subject to} \quad & x(0) = x_0, \\ & \dot{x} = f(x(t), u(t)), && t \in [0, T] \\ & h(x(t), u(t)) \geq 0, && t \in [0, T] \\ & r(x(T)) = 0 \end{aligned} \tag{2.20}$$

This method can produce good results when a sufficiently low step duration is used. However, an important concern when compared to other methods is the computation time. Convergence for a good solution takes a long time because the number of parameters that are used is relatively large. With an increasing number of search parameters comes a increasingly bigger complexity and, because the cost can only be calculated for the completed trajectory, calculating the gradients with respect to each parameter can be very difficult.

Trajectory planning for the double integrator with single shooting will be unconstrained. This is because the solution produced by the optimisation problem will shape u, which will have no influence on the problem's initial conditions.

The solution for this unconstrained problem was found with the Matlab function `fminsearch`.

During the optimisation process, the variables x_1 and x_2 will be necessary for calculation of the cost. The integral 2.9 squared can be easily calculated without ever having to obtain an explicit expression for signals x_1 and x_2. This is because x_2 will be continuous piece-wise straight lines (by integration of constants) and x_1 will be continuous piece-wise parabolas (by integration of straight lines).

Figure 2.2 shows the solution of this shooting problem. The input u has a total of 30 coefficients, each having a duration of 20 ms.

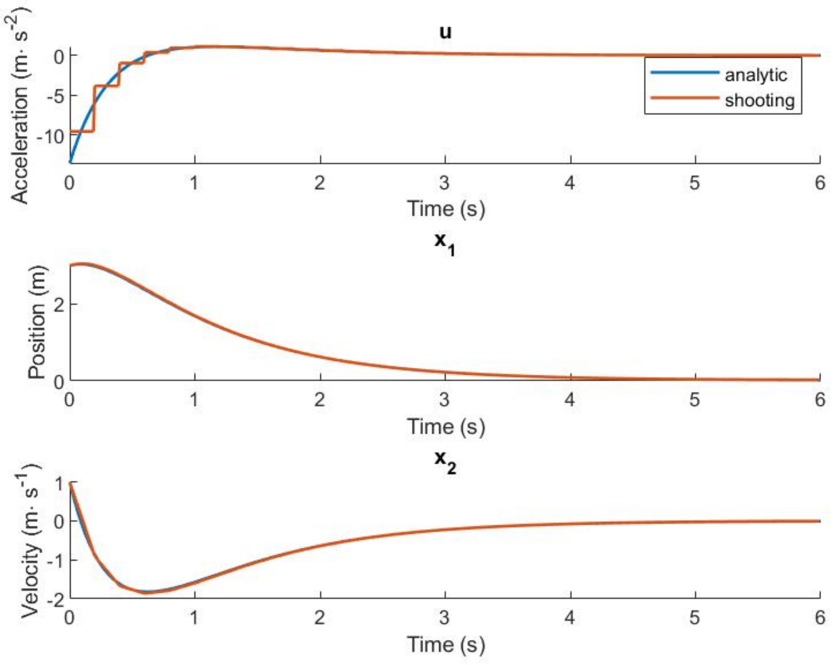


Figure 2.2: Direct Optimisation via a piecewise constant input
Computation time: 2.44 s

2.4 Multiple Shooting

With multiple shooting, u is parameterised in the same way as single shooting (see 2.18). The main difference between multiple shooting and single shooting is that the ODE is calculated for each time interval separately. This ODE uses as initial value s_i and is expressed as

$$\dot{x}_i(t; s_i; q_i) = f(x_i(t; s_i, q_i), q_i), \qquad t \in [t_i; t_{i+1}] \tag{2.21}$$

Continuity of state x for each time-step has to be assured. As a result, an equality constraint has to be added to the problem. This constraint is given by

$$s_{i+1} = x_i(t_{i+1}, s_i, q_i), \qquad i = 0, 1, \dots, N-1 \tag{2.22}$$

where the path cost l_i can now be calculated for each time interval $t \in [t_i; t_{i+1}]$ and is given by

$$l_i(s_i; q_i) = \int_{t_i}^{t_{i+1}} L(x(t; s_i; q_i), q_i)dt \tag{2.23}$$

The optimisation problem can now be redefined as

$$\begin{aligned} \underset{q,s}{\text{minimise}} \quad & \sum_{i=0}^{N-1} l_i(s_i, q_i) + \psi(s_N) \\ \text{subject to} \quad & a(0) = x_0, \\ & s_{i+1} = x_i(t_{i+1}, s_i, q_i), \qquad i = 0, 1, \dots, N-1, \\ & h(s_i, q_i) \geq 0, \qquad\qquad\qquad 1, \dots, N, \\ & r(x(T)) = 0 \end{aligned} \tag{2.24}$$

For a solution u to be obtained with the same duration and order as in single shooting, the optimising algorithm will have to search through twice the amount of variables (an s_i for every q_i). However, the larger search space will be sparse due to the presence of constraints, and, as a result, easier to solve, compared to the small and dense optimisation problem produced by single shooting [8].

Solutions for multiple shooting will be identical to single shooting and will differ only in computation time. Therefore, examples with this method are not be given.

2.5 Quadratic Programming

Quadratic programming problems have the form

$$\begin{aligned} \underset{\overline{x}(.)}{\text{minimise}} \quad & \frac{1}{2}\overline{x}^T H \overline{x} + f^T \quad \text{subject to} \quad A \cdot \overline{x} \leq b, \\ & Aeq \cdot \overline{x} = beq, \\ & lb \leq \overline{x} \leq ub \end{aligned} \tag{2.25}$$

and are not specifically designed to tackle motion planning problems. The first step to apply this technique is to discretise the system's dynamics. A discrete linear system can be represented as

$$\phi_{x_k} + \Gamma_{u_k} - x_{k+1} = 0 \tag{2.26}$$

where x_k and u_k are the state and input at discrete time k, and Φ and Γ are state an input matrices of appropriate dimensions.

The cost function to optimise that is equivalent to 2.11 is given by

$$J_d = x_N^T Q x_N + \sum_{k=0}^{N-1} x_k^T Q x_k + u_k^T R u_k \tag{2.27}$$

where the matrices Q and R will be identical to the continuous time problem.

The goal of the quadratic programming optimiser is to obtain values x_k and u_k for all discrete-time instants that minimise 2.27. In order to formulate the optimisation problem in the form of 2.25, all of the variables to optimise have to be flattened to a the column vector

$$\overline{x} = \begin{bmatrix} x_1^0 \dots x_n^0 & u_1^0 \dots u_m^0 & (\dots) & x_1^{N-1} \dots x_n^{N-1} & u_1^{N-1} \dots u_m^{N-1} & x_1^N \dots x_n^N \end{bmatrix}^T \tag{2.28}$$

where the superscript is the iteration number that goes from 0 to N, n is the dimension of x and m is the dimension of u. There will be a total of $(N-1)(n+m)+n$ variables to optimise.

To optimise 2.27, matrix H will be constructed by repetition of matrices Q and R as shown in

$$H = \begin{bmatrix} [Q] & & & & & \\ & R & & & & \\ & & \ddots & & & \\ & & & [Q] & & \\ & & & & R & \\ & & & & & [Q] \end{bmatrix} \tag{2.29}$$

The system dynamics 2.26 will impose ”dynamic” linear constraints given by

$$\underbrace{\begin{bmatrix} [\phi] & [\Gamma] & [-I] & & \dots & \\ & & [\Phi] & [\Gamma] & [-I] & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\text{Aeq (dynamics)}} \overline{x} = \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\text{beq dynamics}} \tag{2.30}$$

where I is the identity matrix with size equals to the dimension of x.

The first and last samples of x will be restricted to the initial and final conditions, x_i and x_f

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \end{bmatrix}}_{\text{Aeq (initial state)}} \overline{x} = \underbrace{\mathbf{x}^i}_{\text{beq (initial state)}} \tag{2.31}$$

$$\underbrace{\begin{bmatrix} 0 & \dots & 0 & 1 & 0 \\ 0 & \dots & 0 & 0 & 1 \end{bmatrix}}_{\text{Aeq (final state)}} \overline{x} = \underbrace{\mathbf{x}^f}_{\text{beq (final state)}} \tag{2.32}$$

The final matrices Aeq and beq that will be plugged in the optimisation software will be given by

$$\underbrace{\begin{bmatrix} \text{Aeq (dynamics)} \\ \text{Aeq (initial state)} \\ \text{Aeq (final state)} \end{bmatrix}}_{\text{Aeq}} \overline{x} = \underbrace{\begin{bmatrix} \vec{0} \\ x^i \\ x^f \end{bmatrix}}_{\text{beq}} \tag{2.33}$$

An additional inequality constraint may be used to bound values of $\overline{x}$. These are coded in the lb and ub vectors which correspond to the lower bound and upper bounds for $\overline{x}$. Later, in the application examples, this will be used to bound the values of u.

In order to control the double integrator using quadratic programming techniques, the system's discrete-time equivalent will have to be obtained first. Matrices Φ and Γ can be obtained via the Matlab function `c2d`. H will be as described with Q and R identical to the analytical solutions; f will be zero because there are no non-squared components of the cost function. Figure 2.3 shows the solution of the quadratic problem for an unconstrained input and identical initial conditions to the analytical example were used.

Figure 2.4 shows a solution to a problem where u is bounded by

$$\begin{aligned} ub &= U_{\text{max}} \begin{bmatrix} \infty & \infty & 1 & \dots & \infty & \infty & 1 & \infty & \infty \end{bmatrix} \\ lb &= U_{\text{min}} \begin{bmatrix} \infty & \infty & 1 & \dots & \infty & \infty & 1 & \infty & \infty \end{bmatrix} \end{aligned} \tag{2.34}$$

where U_{max} and U_{min} are given are given by ± 1. Here we can see a clear saturation on the input up to time $2.5\,\text{s}$ after which the system evolves like in the unconstrained example.

2.6 Monomial Polynomials

Another way to describe trajectories is by using polynomials. If an optimal trajectory can be modelled by a polynomial, then it can be parameterised in terms of the polynomial's coefficients. As a result, the optimisation problem tends to have a much smaller search space compared to, for example, shooting methods because with a relatively low order of polynomial, a whole trajectory over time $t \in [0, T]$ can be obtained will a relatively good cost.

Polynomial methods are especially suited to deal with *differentially flat systems*[9]. In, differentially flat systems, the state and input variables can be directly expressed, without integrating any differential equation, in terms of the flat output and a finite number of its derivatives[10]. This means that with coefficients that describe just the flat output it is possible to describe all of the other variables of the system, thus resulting in a reduction of the number of parameters to be optimised.

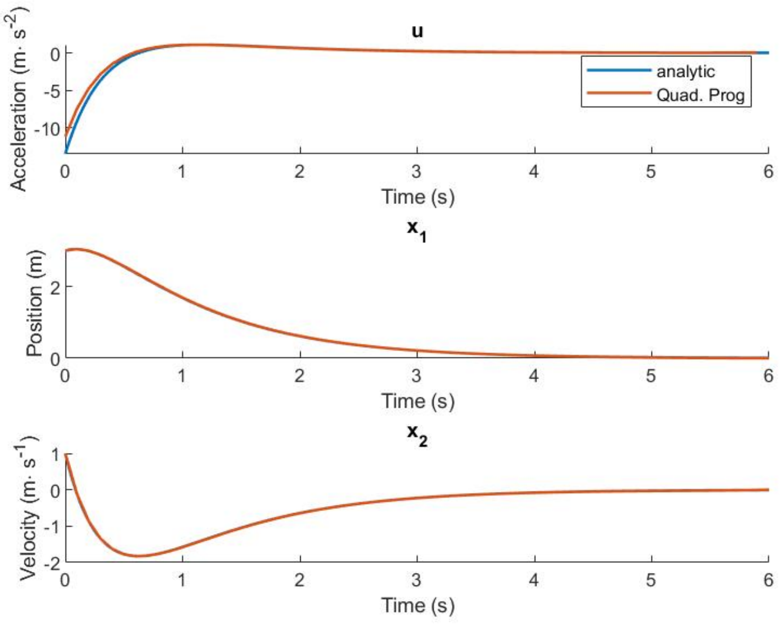


Figure 2.3: Quadratic Programming solution

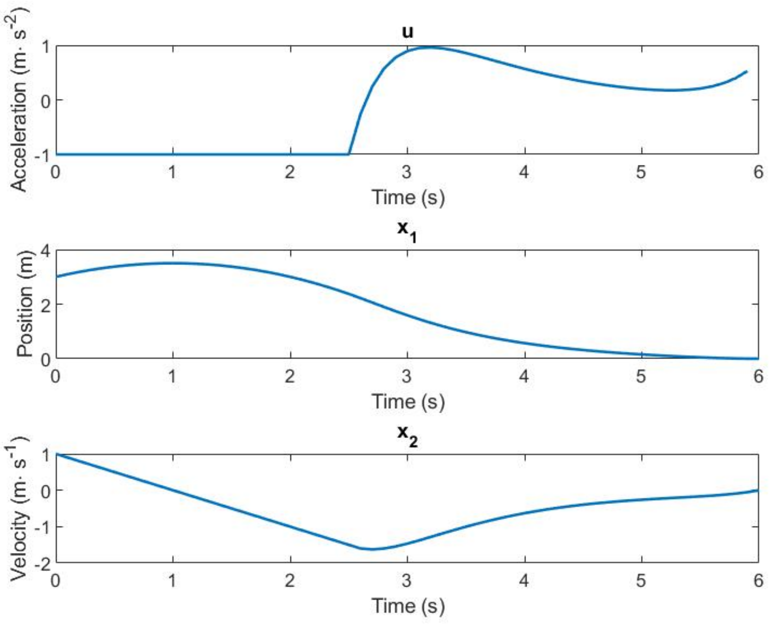


Figure 2.4: Quadratic Programming solution with bounded input

The solution provided by the Matlab function `fmincon` for this method will be a polynomial $\overline{p}$ for x_1. Once the optimal coefficients have been obtained the trajectories for x_1, x_2 and u can be obtained by polynomial derivation and evaluation. This is possible because the system is differentially flat. These polynomials are related in the system of equations

$$\begin{cases} u(t) = a_0 + a_1 t + a_2 t^2 + \ (\dots) \\ x_2(t) = v_0 + a_0 t + \frac{a_1}{2} t^2 + \frac{a_2}{3} t^3 + \ (\dots) \\ x_1(t) = p_0 + v_0 t + \frac{a_0}{2} t^2 + \frac{a_1}{6} t^3 + \frac{a_2}{12} t^4 + \ (\dots) \end{cases} \tag{2.35}$$

The cost function will be the same as the one used for the analytic method, and with the same initial conditions (see 2.9 and 2.17).

Figure 2.5 shows the solution of a polynomial optimisation problem for a polynomial of order 7. Higher orders where attempted but they took too long to converge.

In order to guarantee the initial and final conditions of the state variable x, the linear constraint that has to be introduced is given by

$$\underbrace{\begin{bmatrix} 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 1 & 0 \\ T^N & T^{N-1} & \dots & T & 0 \end{bmatrix}}_{\text{Aeq}} \overline{p} = \underbrace{\begin{bmatrix} x_1^i \\ x_2^i \\ 0 \end{bmatrix}}_{\text{beq}} \tag{2.36}$$

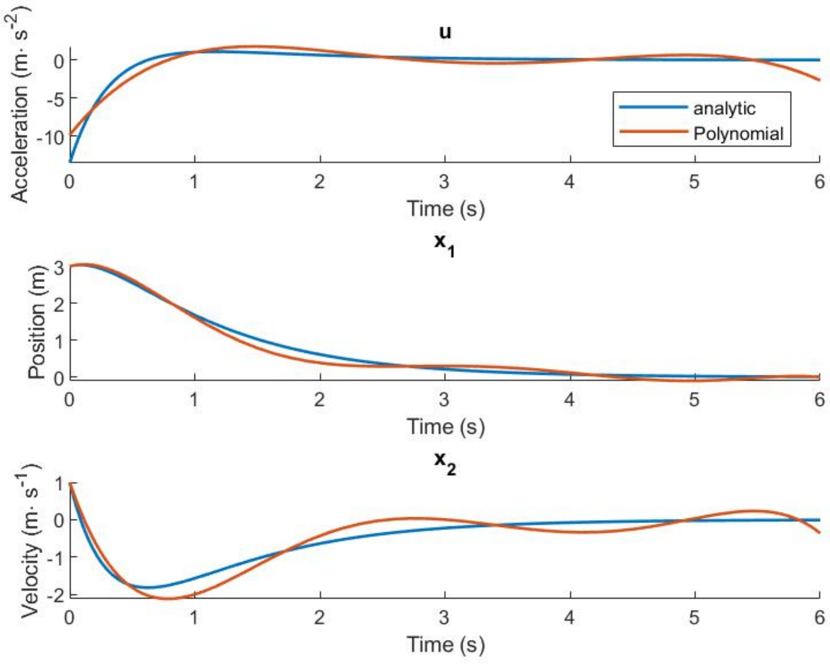


Figure 2.5: Direct Optimisation with Polynomials
Polynomial order: 7
Computation time: 1.33 s

2.7 Bezier Curves

What follows is a brief summary of the type of polynomials that will be exploited in the thesis.

A Bernstein polynomial is given by

$$P(\tau) = \sum_{k=0}^{n} p_k B_{k,n}(\tau), \qquad \tau \in [0,1] \tag{2.37}$$

where n is the order of the polynomial and τ is the time contraction of t, given by

$$\tau = \frac{t}{T}, \qquad t \in [0,T] \tag{2.38}$$

p_k are the polynomial coefficients or *control points*, and b_k^n are the *Bernstein Basis*, given by

$$B_k^n(\tau) = \binom{n}{k} (1-\tau)^{n-k} \tau^k \tag{2.39}$$

As a result, a bernstein polynomial of order n is a linear combination of $n+1$ bernstein basis with weights given by p_k.

The Bernstein Basis, as a function of τ, for 6 different orders, n, are plotted, on in each subplot of figure 2.6.

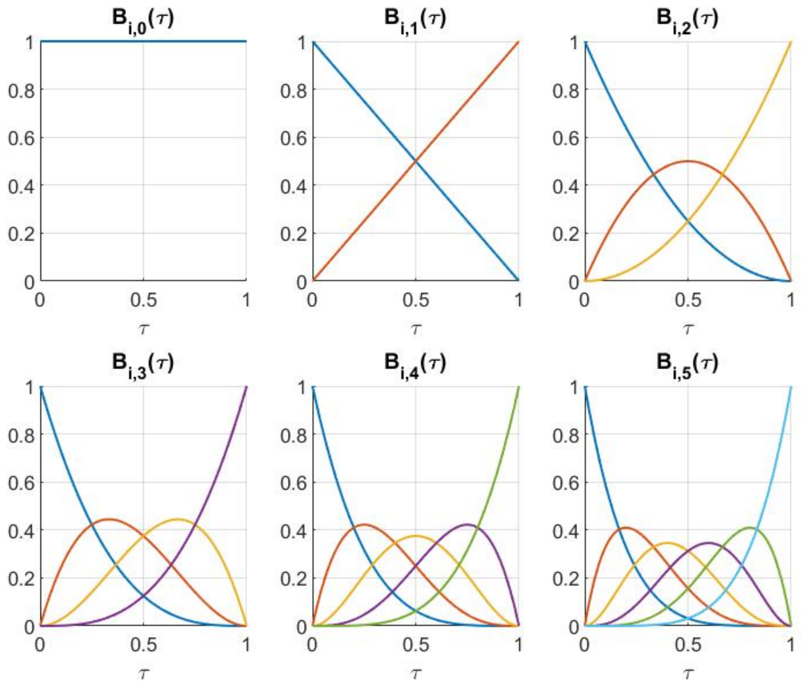


Figure 2.6: Representation of the Bernstein Basis for $\tau \in [0,1]$ for orders 0 to 5

For a polynomial of order n, the $i \in 0..n$ control points can be represented on a vector p, for example,

$$p = \begin{bmatrix} 0 \\ 0.5 \\ 1 \\ 0.7 \\ 0.3 \\ -0.7 \\ -1 \\ -0.5 \\ -0.1 \end{bmatrix} \tag{2.40}$$

The plot of the polynomials respective to vector p is in figure 2.7. The control points on vector p are plotted along time time axis uniformly. What can be seen is that the control points ”attract” the curve towards themselves.

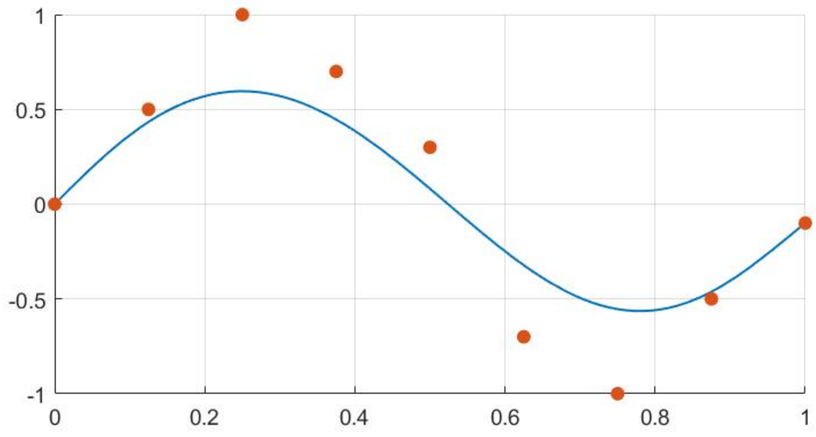


Figure 2.7: A Bernstein Polynomial

Two Bernstein Polynomials of the same order, i.e., same number of control points, can be plotted against each other within the time domain $\tau \in [0, 1]$. The control points of the curves together form $n + 1$ points in 2 dimensions. These are plotted together with the example 2-D plot of 2.8. Once again, like in the 1-D case, the control points ”shape” the curve by attracting the curve to themselves.

One thing that is important to note is how the control points’ will play a role on the shape of the curve. Figure 2.9 shows how the same control points can form a completely different curve.

Figure 2.8 shows a two dimensional Bernstein polynomial.

One of the first properties than can be observed in figure 2.7 is that, within the domain $t \in [0, T]$, the polynomial is limited by a convex hull formed by the control points.

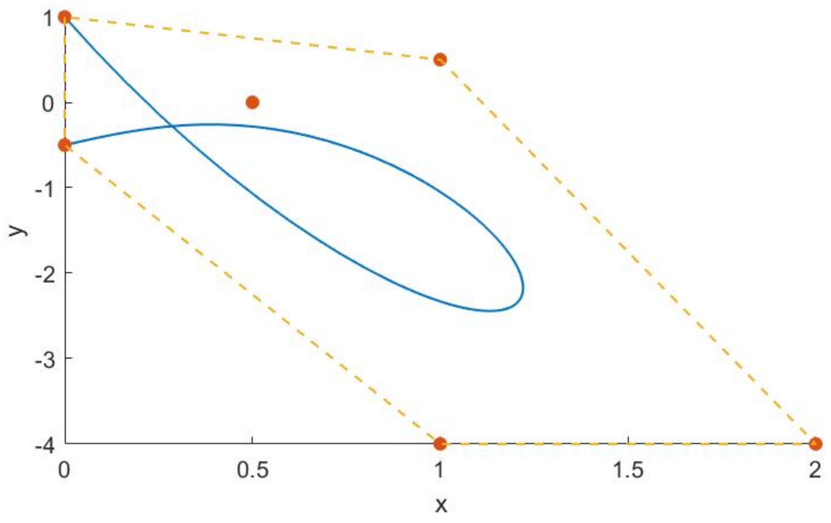


Figure 2.8: A 2D Bernstein Polynomial

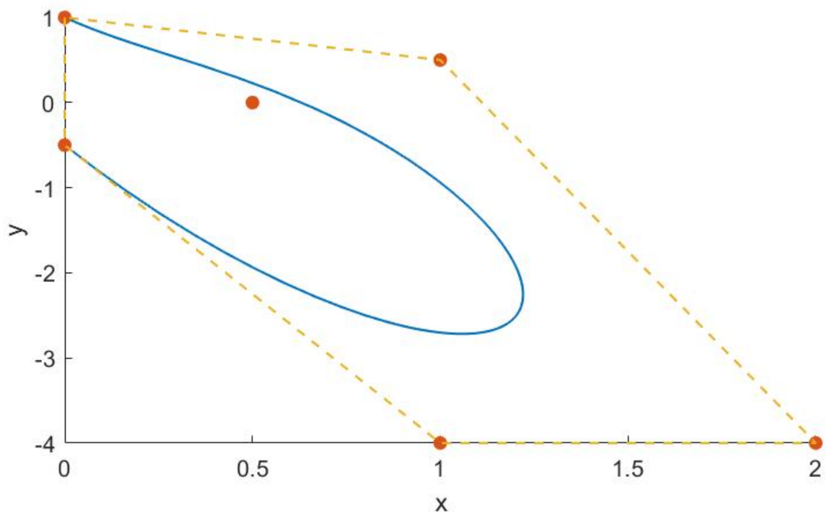


Figure 2.9: A 2D Bernstein Polynomial

The initial and final values of $P(t)$ in the interval $[0, T]$ are given by

$$\begin{aligned} P(0) &= p_{n,0} \\ P(T) &= p_{n,n} \end{aligned} \tag{2.41}$$

and the derivative and integral are given by

$$\dot{P}(t) = \frac{n}{T} \sum_{k=0}^{n-1} (p_{k+1,n} - p_{k,n}) B_{k,n-1}(t) \tag{2.42}$$

$$\int_0^T P(t)dt = \frac{T}{n+1}\sum_{k=0}^{n} p_{k,n} \tag{2.43}$$

The control points for the derivative of a bernstein polynomial can be obtained my a multiplication of a derivation matrix by the control points of the original curve stored in the column vector, with the matrix given by

$$\boldsymbol{D}_{N-1} = \begin{bmatrix} -\frac{N}{T} & \frac{N}{T} & 0 & \dots & & 0 \\ 0 & -\frac{N}{T} & \frac{N}{T} & \dots & & 0 \\ \vdots & \ddots & \ddots & \ddots & & \\ 0 & & \dots & & -\frac{N}{T} & \frac{N}{T} \end{bmatrix} \in \mathbb{R}^{(N+1)\times N} \tag{2.44}$$

The control points of the Anti-Derivative/Primitive can be obtained similarly to the derivation by multiplying the control points to a primitive matrix (my discovery) and adding a vector:

$$p = A \cdot \dot{p} + p_0 \tag{2.45}$$

where p_0 is the initial values of P and the matrix A is given by

$$\boldsymbol{A}_{N+1} = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ \frac{T}{N+1} & 0 & \dots & 0 & 0 \\ \frac{T}{N+1} & \frac{T}{N+1} & \dots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \\ \frac{T}{N+1} & \frac{T}{N+1} & \dots & \frac{T}{N+1} & \frac{T}{N+1} \end{bmatrix} \tag{2.46}$$

A Bernstein polynomial of degree N and coefficients $p_{k,N} \in \mathbb{R}, k = 0, \dots, N$ can be expressed as a Bernstein polynomial of degree M with $M > N$, with coefficients $p_{k,M} \in \mathbb{R}, k = 0, \dots, M$ given by

$$p_{k,M} = \sum_{j=\max(0,k+M-N)}^{\min(N,k)} \frac{\binom{M-N}{k-j}\binom{N}{j}}{\binom{M}{k}} p_{j,N} \in \mathbb{R}^n \tag{2.47}$$

The control points for the degree elevation can also be obtained by multiplying the control points vector by a matrix $\boldsymbol{E}$ whose indexes are given by

$$\boldsymbol{E}_{ij} = \frac{\binom{N}{j}\binom{M-N}{i-j}}{\binom{M}{i}} \tag{2.48}$$

Multiplication and addition is given by

$$f(t)g(t) = \sum_{i=0}^{m+n} \left(\sum_{i=\max(0,i-n)}^{\min(m,i)} \frac{\binom{m}{j}\binom{n}{i-j}}{\binom{m+n}{i}} f_{j,m} g_{i-j,n} \right) B_{i,m+n}(t) \tag{2.49}$$

$$f(t) \pm g(t) = \sum_{i=0}^{m} \left(f_{i,n} \pm \sum_{j=\max(0,i-m+n)}^{\min(n,i)} \frac{\binom{n}{j}\binom{m-n}{i-j}}{\binom{m}{i}} g_{j,n} \right) B_{i,m+n}(t) \tag{2.50}$$

The De Casteljau's algorithm is a recursive method to evaluate polynomials in Bernstein form. A geometric interpretation of this algorithm presented as follows:

1. Connect the consecutive control points in order to create the control polygon of the curve.
2. Subdivide each line segment of this polygon with the ratio $t : (1-t)$ and connect the obtained points. This way a new polygon is obtained having one fewer segment.
3. Repeat the process until a single point is achieved – this is the point of the curve corresponding to the parameter t.

Figure 2.10 illustrates the breakdown of the control points into polygons and sub polygons. The use of this algorithm is popular algorithm in Computer Aided Graphic Design.

The evaluation of the curve can be done analytically. For a Bernstein Polynomial $\boldsymbol{x}_N : [0, t_f] \to \mathbb{R}^n$, and a scaler $t_{div} \in [0, t_f]$, the Bernstein polynomail at t_{div} can be computer using the following recursive relation:

$$\begin{aligned} \overline{\boldsymbol{x}}_{i,N}^{[0]} &= \overline{\boldsymbol{x}}_{i,N}, \quad i = 0, \dots, N \\ \overline{\boldsymbol{x}}_{i,N}^{[j]} &= \overline{\boldsymbol{x}}_{i,N}^{[j-1]} \frac{t_f - t_{div}}{t_f} + \overline{\boldsymbol{x}}_{i+1,N}^{[j-1]} \frac{t_{div}}{t_f} \end{aligned} \tag{2.51}$$

with $i = 0, \dots, N-j$, and $j = 1, \dots, N$. Then, the Bernstein polynomial evaluated at t_{div} is given by

$$\overline{\boldsymbol{x}}_N(t_{div}) = \overline{\boldsymbol{x}}_{0,N}^{[N]}. \tag{2.52}$$

Moreover, the Bernstein polynomial can be subdivided at t_{div} into two Nth order Bernstein polynomial with Bernstein coefficients

$$\overline{\boldsymbol{x}}_{0,N}^{[0]}, \overline{\boldsymbol{x}}_{0,N}^{[1]}, \dots, \overline{\boldsymbol{x}}_{0,N}^{[N]}, \quad \text{and} \quad \overline{\boldsymbol{x}}_{0,N}^{[N]}, \overline{\boldsymbol{x}}_{0,N}^{[N-1]}, \dots, \overline{\boldsymbol{x}}_{0,N}^{[0]}. \tag{2.53}$$

The use of Bernstein polynomials as an alternative to monomial polynomials, as presented in the previous section, is preferable (see [3]).

The convex hull property of these polynomials will prove useful in an algorithm that calculates distance between curves, which will be used in deconfliction of trajectories in multiple vehicle control. By just knowing the position of the control points in the space, it is possible to guarantee constraint satisfaction in the whole trajectory and not just in the control points.

Just as with monomial polynomials, when these Bernstein polynomials are applied to differentially flat systems, a reduction of the number of parameters to optimise is possible because with just a flat output, all state variables can be derived. If the system is not differentially flat, then parameters for all

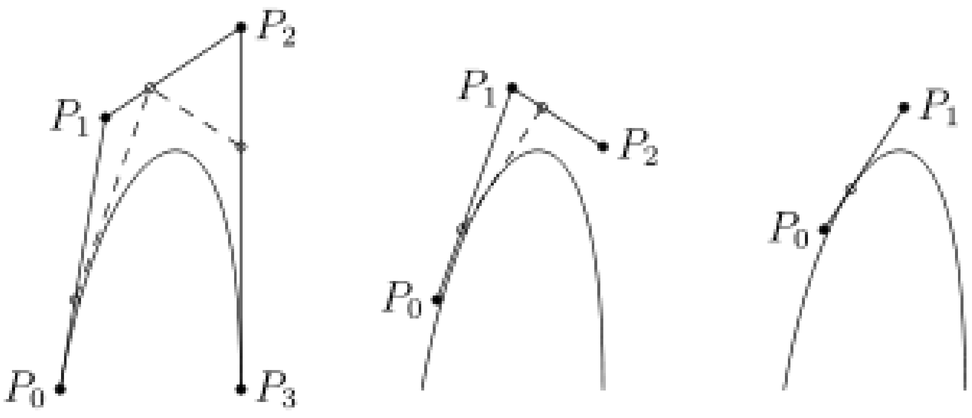


Figure 2.10: Visual representation of the De Casteljau algorithm

state variables need to be defined and optimised but will be subject to the constrained imposed by the system's dynamics. The optimisation problems developed for motion planning in this work center around the usage of what is known as Bernstein polynomials.

The solution obtained by the Matlab function `fmincon` will provide the coefficients $\overline{p}$ for the Bernstein polynomial of state variable x_1.

Figure 2.11 shows the solution for the same quadratic optimisation problem and with the same initial conditions as the previous sections. The red circles in the figure show the distribution of the obtained coefficients. The resulting polynomial has order 14.

In order to guarantee that the initial and final conditions are respected, the linear constraint that has to be introduced is given by

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \frac{n}{T} & -\frac{n}{T} & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}}_{\text{Aeq}_{\text{1D}}} \overline{p} = \underbrace{\begin{bmatrix} x_0^i \\ x_1^i \\ 0 \end{bmatrix}}_{\text{beq}_{\text{1D}}} \tag{2.54}$$

2.8 Differentially Flat Systems

A nonlinear system of the form

$$\dot{x} = f(x(t), u(t)), \quad x(t) \in \mathbb{R}^n \tag{2.55}$$

is said to be *differentially flat* [10] or simply *flat* if there exists a set of variables $y = (y_1, \dots, y_m)$ called the *flat output*, such that

- the components of y are not differentially related over $\mathbb{R}$;
- 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, and

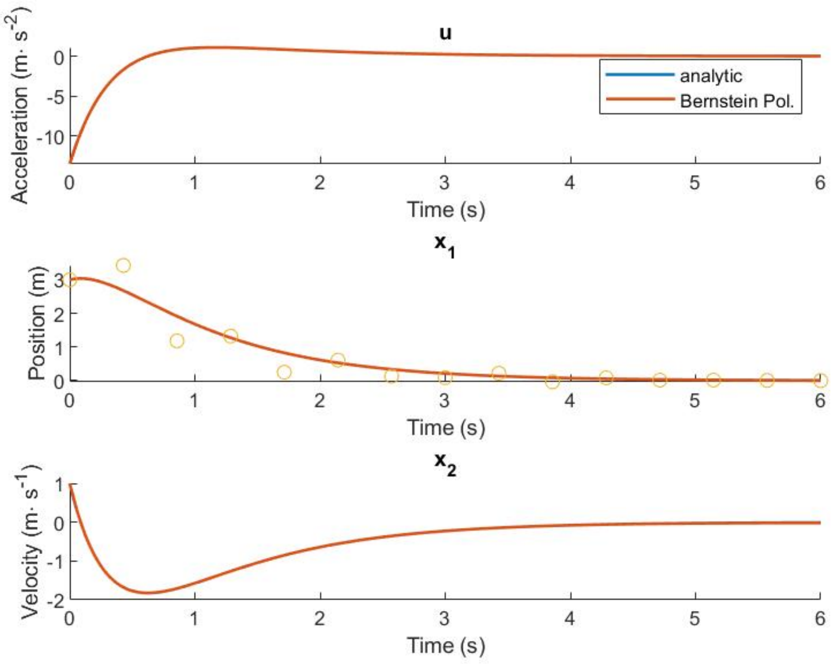


Figure 2.11: Direct Bernstein Solution

- every component of y may be expressed as a function of the system variables and of a finite number of their time-derivatives.

The flat output has, in general, a clear physical interpretation and captures the fundamental properties of a given system and its determination allows to simplify considerably 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.

2.9 Remarks

Out of all of the presented parameterisation methods, the preferred is Bernstein polynomials because they provide fast solutions and, as will be seen in later chapters, will work well for both prove advantageous for differentially and non differentially flat systems.

3

Autonomous Vehicle Models

Contents

In this chapter equations that rule the motion of an autonomous marine vehicle are derived. But first, the coordinate frames will be defined (3.1). Then the general vehicle motion equations for the Dubin's car (3.3). Finally, the motion equations for the MEDUSA model are presented (the characterization of the vehicle MEDUSA (3.2).

This chapter will focus primarily on dynamics in a two dimensional space.

3.1 Reference Frames and Notation

To derive the equations of motion for a marine vehicle it is standard practice to define two coordinate frames; Earth-fixed inertial frame $\{U\}$ composed by the orthonormal axes $\{x_U, y_U, z_U\}$ and the body-fixed frame $\{B\}$ composed by the axes $\{x_B, y_B, z_B\}$, as indicated in Figure 3.1.

- x_B is the longitudinal axis (directed from the stern to fore)
- y_B is the transversal axis (directed from to starboard)
- z_B is the normal axis (directed from top to bottom)

To simplify the model equations, the origin of the body-fixed frame is normally chosen to coincide with the centre of mass of the vehicle. The motion control of $\{B\}$ (that corresponds to the motion of the vehicle) is described relative to the inertial frame $\{U\}$.

In general, six independent coordinate are necessary to determine the evolution of the position and orientation (six Degress of Freedom (DOF)), three position coordinates (x, y, z) and using the Euler orientation angles (ϕ, θ, ψ). The six motion components are defined as *surge*, *sway*, *heave*, *roll*, *pitch*, and *yaw*, and adopting the SNAME [1] notation it can be written as in the Table 3.1 or in a vectorial form:

- $\eta_1 = [x, y, z]^T$ - position of the origin of $\{B\}$ with respect to $\{U\}$
- $\eta_2 = [\phi, \theta, \psi]^T$ - orientation of $\{B\}$ with respect to $\{U\}$.
- $\nu_1 = [u, v, w]^T$ - linear velocity of the origin of $\{B\}$ relative to $\{U\}$, expressed in $\{B\}$.
- $\nu_2 = [p, q, r]^T$ - angular velocity of $\{B\}$ relative to $\{U\}$, expressed in $\{B\}$.
- $\tau_1 = [X, Y, Z]^T$ - actuating forces expressed in $\{B\}$.
- $\tau_2 = [K, M, N]^T$ - actuating moments expressed in $\{B\}$

In compact form yields

$$\begin{cases} \eta = [\eta_1^T, \eta_2^T]^T \\ \nu = [\nu_1^T, \nu_2^T]^T \\ \tau_{RB} = [\tau_1^T, \tau_2^T]^T \end{cases} \tag{3.1}$$

[1] The Society of Naval Architects & Marine Engineers - http://www.sname.org/

Degree of Freedom	Forces and moments	Linear and angular velocities	Position and Euler angles
1. Motion in the x-direction (surge)	X	u	x
2. Motion in the y-direction (sway)	Y	v	y
3. Motion in the z-direction (heave)	Z	w	z
4. Rotation about the x-axis (roll)	K	p	ϕ
5. rotation about the y-axis (pitch)	M	q	θ
6. Rotation about the z-axis (yaw)	N	r	ψ

Table 3.1: Notation used for marine vehicles

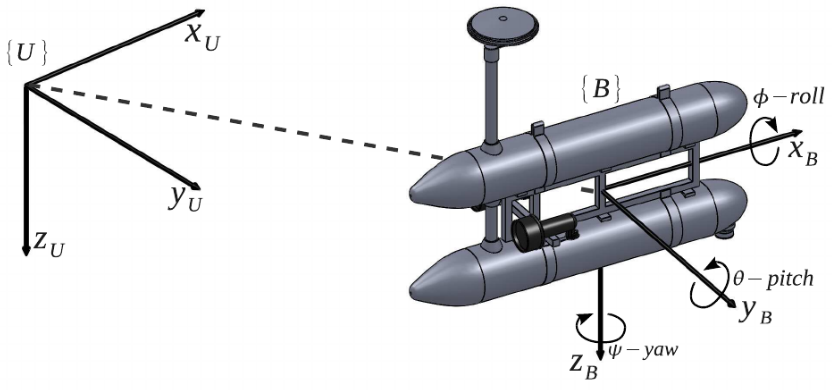


Figure 3.1: Coordinate frames

3.2 Medusa

The kinematics involves only the geometrical aspects of motion, and relates the velocities with position. Using the coordinate frames notation in Section 3.1, the kinematic equation can be expressed as

$$\dot{\eta} = J(\eta)\nu \tag{3.2}$$

with

$$J(\eta) = \left[{}^U_B R\right] \tag{3.3}$$

where

$${}^U_B R(\Theta) = \begin{bmatrix} c\psi c\theta & c\psi s\theta s\phi - s\psi c\phi & c\psi s\theta c\phi + s\psi s\phi \\ s\psi c\theta & s\psi s\theta s\phi + c\psi c\phi & s\psi s\theta c\phi - c\psi s\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix}, s\cdot = \sin(\cdot), c\cdot = \cos(\cdot) \tag{3.4}$$

is the rotation matrix from $\{B\}$ to $\{U\}$, [11], defined my means of three successive rotations (zyx-convention):

$${}^U_B R(\Theta) = R_{z,\psi} R_{y,\theta} R_{x,\phi} \tag{3.5}$$

and

$$T_\Theta(\Theta) = \begin{bmatrix} 1 & s\phi\, t\theta & c\phi t\theta \\ 0 & c\phi & -s\phi \\ 0 & c\phi/c\theta & c\phi/c\theta \end{bmatrix}, \theta \neq \pm 90 \tag{3.6}$$

is the Euler attitude transformation matrix that relates the body-fixed angular velocities (p, q, r) with the roll($\dot{\phi}$), pitch($\dot{\theta}$) and yaw($\dot{\psi}$) rates. Notice that $T_\Theta(\Theta)$ is not defined for the pitch angle $\theta = \pm 90$, resulting from using Euler angles to describe the vehicle's motion. To avoid this singularity, one possibility is to use a quaternion representation [11]. However, due to physical restrictions, the vehicle will operate far from this singularity ($\theta \approx 0$ and $\psi \approx 0$) which implies that we can use the Euler representation.

3.2.1 Dynamic equations

Since the hydrodynamic forces and moments have simpler expressions if written in the body frame because they are generated by the relative motion between the body and the fluid, it is convenient to formulate Newton's law in $\{B\}$ frame. In that case, the rigid-body equation can be expressed as

$$M_{RB}\dot{\nu} + C_{RB}(\nu)\nu = \tau_{RB} \tag{3.7}$$

where M_{RB}s the rigid body inertia matrix, C_{RB} represents the Coriolis and centrifugal terms and τ_{RB} is a generalized vector of external forces and moments and can be decomposed as

$$\tau_{RB} = \tau + \tau_A + \tau_D + \tau_R + \tau_{dist} \tag{3.8}$$

where

1. τ - Vector of forces and torques due to thrusters/surfaces which usually can be viewed as the control input
2. τ_A + The force and moment vector due to the hydrodynamic added mass,

   $$\tau_A = -M_A\dot{\nu} - C_A(\nu)\nu \tag{3.9}$$

3. τ_D - Hydrodynamics terms due to lift, drag, skin friction, etc.

   $$\tau_D = -D(\nu)\nu \tag{3.10}$$

   where $D(\nu)$ denotes the hydrodynamic damping matrix (positive definite).
4. τ_R - Restoring forces and torques due to gravity and fluid density,

   $$\tau_R = -g(\eta) \tag{3.11}$$

5. τ_{dist} - External disturbances: waves, wind, etc.

Replacing (3.8) on (3.7), taking into account (3.9), (3.10), the dynamic equations can be written as

$$\underbrace{M_{RB}\dot{\nu} + C_{RB}(\nu)\nu}_{\text{rigid-body terms}} + \underbrace{M_A(\dot{\nu}) + C_A(\nu)\nu + D(\nu)\nu}_{\text{hydrodynamic terms}} + \underbrace{g(\nu)}_{\text{restoring terms}} = \tau + \tau_{dist} \tag{3.12}$$

which can be simplified to

$$M\dot{\nu} + C(\nu)\nu + D(\nu)\nu + g(\nu) = \tau + \tau_{dist} \tag{3.13}$$

where $M = M_{RB} + M_a$, $C(\nu) = C_{RB}(\nu) + C_A(\nu)$.

3.2.2 Simplified Equations of Motion

This thesis will focus on movement along a 2-D plane, therefore, the dynamics and kinematics can be simplified such that there are only three degrees of freedom $[x, y, \psi]$.

The kinematics with take the simpler form

$$\begin{aligned} \dot{x} &= u\cos\psi - v\sin\psi, \\ \dot{y} &= u\sin\psi + v\cos\psi, \\ \dot{\psi} &= r. \end{aligned} \tag{3.14}$$

τ_u and τ_r are the external force in $surge$ (common mode) and the external torque about the z-axis (differential mode between thrusters), respectively, which can by obtained by

$$\begin{aligned} \tau_u &= F_{sb} + F_{ps}, \\ \tau_r &= l(F_{ps} - F_{sb}) \end{aligned} \tag{3.15}$$

where F_{sb} and F_{ps} are the starboard and port-side thruster forces, respectively, and l is the length of the arm with respect to the centre of mass.

By neglecting roll, pitch and heave motion, the equations for (u, v, r) without disturbances are

$$\begin{aligned} m_u\dot{u} - m_v vr + d_u u &= \tau_u, \\ m_v\dot{v} + m_u ur + d_v v &= 0, \\ m_r\dot{r} - m_{ub} uv + d_r r &= \tau_r, \end{aligned} \tag{3.16}$$

where

$$\begin{aligned} m_u &= m - X_{\dot{u}} \quad d_u = -X_u - X_{|u|u}|u| \\ m_v &= m - Y_{\dot{v}} \quad d_v = -Y_v - Y_{|v|v}|v| \\ m_r &= I_z - N_{\dot{r}} \quad d_r = -N_r - N_{|r|r}|r| \\ m_{uv} &= m_u - m_v \end{aligned} \tag{3.17}$$

All the previous equations were expressed without considering the influence of external factors like ocean currents. If a constant irrotational ocean current, v_c, is introduced, forming an angle $v = v_r + v_c$, where u_r and v_r are the components of the AUV velocity with respect to the current and u_c and v_c are the components of the ocean current velocity in the body frame. The previous dynamic equations (3.16) become

$$\begin{aligned} m_u\dot{u} - m_v(v_r + v_c)r + d_u u &= \tau_u, \\ m_v\dot{v} + m_u(u_r + u_c)r + d_v v &= 0, \\ m_r\dot{r} - m_{ub}(u_r + u_c)(v_r + v_c) + d_r r &= \tau_r, \end{aligned} \tag{3.18}$$

where the expressions for masses and drags remain the same as in (3.17).

3.2.3 ***Differentially flatify***

Actually, the medusa model is differentially flat if x, y and ψ are to be considered the flat outputs.

The remaining variables can be obtained by rearranging the kinematics:

$$\begin{aligned} \begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} &= \begin{bmatrix}\cos(\psi) & -\sin(\psi)\\ \sin(\psi) & \cos(\psi)\end{bmatrix}\cdot\begin{bmatrix}u\\ v\end{bmatrix} \Leftrightarrow \\ \begin{bmatrix}u\\ v\end{bmatrix} &= \begin{bmatrix}\cos(\psi) & -\sin(\psi)\\ \sin(\psi) & \cos(\psi)\end{bmatrix}^{-1}\cdot\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} \Leftrightarrow \\ \begin{bmatrix}u\\ v\end{bmatrix} &= \begin{bmatrix}\cos(\psi) & \sin(\psi)\\ -\sin(\psi) & \cos(\psi)\end{bmatrix}\cdot\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} \end{aligned} \tag{3.19}$$

and the inputs from the dynamics:

$$\begin{aligned} \tau_u &= m_u\dot{u} - m_v v r + d_u u, \\ \tau_r &= m_r\dot{r} - m_{ub} u v + d_r r, \end{aligned} \tag{3.20}$$

These equations will always be valid, however, only for x, y and ψ that respect

$$m_v\dot{v} + m_u u r + d_v v = 0 \tag{3.21}$$

because this equation is what guarantees $\tau_v = 0$. If $\tau_v \neq 0$, the vehicle will have to be fully actuated for the vehicle to follow it.

3.3 Dubin Car

The Medusa model can be even further simplified by a Dubin's Car.

In geometry, the term Dubin's path typically refers to the shortest curve that connects two points in the two-dimensional Euclidean plane (i.e. x-y plane) with a constraint on the curvature of the path and with prescribed initial and terminal tangents to the path, and an assumption that the vehicle travelling the path can only travel forward [5].

A Dubin's car is a simple model for a vehicle that transverses a Dubin's path.

The main difference to the Medusa Model is that this will does not possess the possibility of side slip, therefore, less variables and dynamics equations will be necessary which will translate to a smaller computation time.

A simple kinematic car model for the systems is:

$$\begin{aligned} \dot{x} &= u\cos\psi \\ \dot{y} &= u\sin\psi \\ \dot{\psi} &= \omega \end{aligned} \tag{3.22}$$

where (x, y) is the car's position, ψ is the car's heading, u is the car's speed, and ω is the car's turn rate.

The dynamics will be the simplest possible: basic accelerations.

$$\begin{aligned} \dot{\omega} &= r \\ \dot{u} &= a \end{aligned} \tag{3.23}$$

This model is differentially flat because for any continuous x and y, all of the other variables can be derived.

The tangent and rotational speeds can be obtained by

$$\begin{aligned} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix} \cdot \begin{bmatrix} u \\ 0 \end{bmatrix} \Leftrightarrow \\ \begin{bmatrix} u \\ 0 \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix}^{-1} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \Leftrightarrow \\ \begin{bmatrix} u \\ 0 \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & \sin(\psi) \\ -\sin(\psi) & \cos(\psi) \end{bmatrix} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \end{aligned} \tag{3.24}$$

ψ can be obtained by integrating ω, however, because ω is not a flat output, ψ is derived as

$$\psi = \arctan\frac{\dot{y}}{\dot{x}} \tag{3.25}$$

Finally, tangent acceleration, τ_1 and torque, τ_2 can be derived from u and ω:

$$\tau_1 = \dot{u} \tag{3.26}$$

$$\tau_2 = \dot{\omega} \tag{3.27}$$

which can be used as inputs to the system.

4 Implementation

Contents

In this chapter, implementation of a Direct Methods based on the usage of Bernstein polynomails in both Matlab and Python will is described. This chapter focuses on formalising the optimisation problem for multiple vehicles. Once the optimisation problem is formalised, it can be solved by a whole range of non linear optimisation algorithms.

4.1 The Optimisation Problem

The Motion Planning problem for multiple vehicles that will be focused for the project consists in a go-to formation manoeuvre [9]. The go-to formation manoeuvre consists in the simultaneous arrival of a formation of vehicles to desired locations whilst simultaneously avoiding collisions between each other and the environment.

Just as for the case of optimisation for a single vehicle, motion planning for multiple vehicles will also be based on the minimisation of an appropriate cost function. This time, however, the cost will be the result of the sum of the costs of each individual vehicle and an added constraint will be necessary that will take into account inter vehicle collisions.

The optimal control formulation for this motion planning problem is defined as

$$\begin{aligned} \underset{x^{[i]}(.),u^{[i]}(.),i=1,\dots N_v}{\text{minimize}} \quad & \int_0^T \sum_{i=1}^{N_v} L_i(x^{[i]}(t),u^{[i]}(t))dt + \Psi(x^{[i]}(T)) \\ \text{subject to} \quad & x^{[i]}(0) = x_0^{[i]}, \\ & x^{[i]}(T) = x_f^{[i]}, \\ & \dot{x}^{[i]} = f^{[i]}(x^{[i]}(t),u^{[i]}(t)), \qquad t \in [0,T] \\ & h(x(t),u(t)) \geq 0, \end{aligned} \tag{4.1}$$

where $i = 1 \dots N_v$ refers to a vehicle out of N_v vehicles.

This problem will also produce a solution within time horizon $t \in [0,T]$. Initial and terminal constraints will exist for every vehicle, as well as inter-vehicle constraints.

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 optimises 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*. An example where only the Mayer form would be necessary could be a situation where the desired destination of the vehicles does not have enough room for them all to be arranged in their desired positions. Therefore, the goal of the optimiser is to find the closest to the desired positions.

A direct method based on Bernstein polynomials is used, therefore, some or all of the state variables/inputs are approximated by polynomials, each with the same order N.

By using Bernstein approximation, all of the states and inputs for each vehicle combined will produce

$N+1$ control points per number of state variables plus number of inputs per total number of vehicles.

Let $0 = t_0 < t_1 < \cdots < t_N = T$ be a set of equidistant *time nodes*, i.e., $t_j = j\frac{t_N}{N}$, with $T > 0$. By following the notation of Bernstein polynomials:

$$\begin{aligned} x^{[i]}(t) \approx x_N(t) &= \sum_{j=0}^{N} \overline{x}_{j,N} b_{j,N}(t), \quad t \in [0,T] \\ u^{[i]}(t) \approx u_N(t) &= \sum_{j=0}^{N} \overline{u}_{j,N} b_{j,N}(t), \quad t \in [0,T] \end{aligned} \tag{4.2}$$

with $x_N : [0,T] \to \mathbb{R}^{n_x}$ and $u_N : [0,T] \to \mathbb{R}^{n_u}$. In the equation above, $\overline{x}_{j,N} \in \mathbb{R}^{n_x}$ and $\overline{u}_{j,N} \in \mathbb{R}^{n_u}$ are Bernstein coefficients. Let $\overline{x}_N \in \mathbb{R}^{n_x \times (N+1)}$ and $\overline{u}_N \in \mathbb{R}^{n_u} \times (N+1)$ be defined as $\overline{x}_N = [\overline{x}_{0,N}, \ldots, \overline{x}_{N,N}]$, and $\overline{x}_N = [\overline{u}_{0,N}, \ldots, \overline{u}_{N,N}]$

The optimisation problem now becomes

$$\begin{aligned} \underset{\overline{x}_N^{[i]}, \overline{u}_N^{[i]}, i=1,\ldots N_v}{\text{minimize}} \quad & \int_0^T \sum_{i=1}^{N_v} L^{[i]}(\overline{x}_N^{[i]}, \overline{u}_N^{[i]}) dt + \Psi(x_{N,N}) \\ \text{subject to} \quad & \overline{x}_{0,N}^{[i]} = x_0^{[i]}, \\ & \overline{x}_{N,N}^{[i]} = x_f^{[i]}, \\ & \sum_{k=0}^{N} \boldsymbol{D}_{j,k} \overline{x}_{k,N}^{[i]} = f^{[i]}(\overline{x}_{j,N}, \overline{u}_{j,N}), \qquad \forall j = 0, \ldots, N \quad h(x(t), u(t)) \geq 0, \end{aligned} \tag{4.3}$$

where $D_{j,k}$ is the (j,k)-th entry of the differentiation matrix $D_N = D_{N-1} E_{N-1}^N \in \mathbb{R}^{(N+1)\times(N+1)}$ that is obtained by combining the Bernstein Differentiation matrix (2.44) with the Bernstein degree elevation matrix whose indexes are given by (2.48).

What this new optimisation problem fundamentally means is that cost, and constraints dealt by $h(.)$ can treat the control points as actual parameters for a polynomial function. On the other hand, the dynamics assume that the control points are good enough approximations for the polynomial functions themselves, which means that to force respecting dynamics means to force the dynamics to be respected on every time node respective to a control point.

The optimisation problem (4.3) means that the cost can be obtained by applying the running and terminal costs to the control points.

4.2 The Log Barrier Function

A log barrier functional can be added to the cost functional such that the motion planning problem becomes unconstrained. Specifically, the inequality constraints are moved into the cost functional by applying a log barrier functional to them.

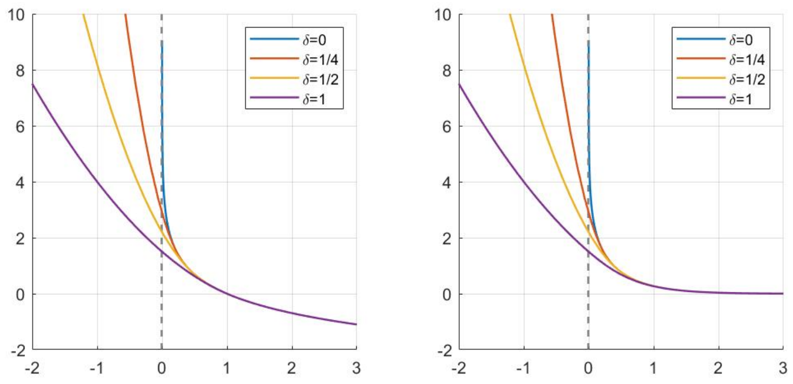


Figure 4.1: log barrier functional without (left) and with (right) the hockey stick function

An optimisation problem of the form

$$\begin{aligned} \text{minimize} \quad & \int_0^T l(x(\tau), u(\tau), \tau)d\tau + m(x(T)) \\ \text{subject to} \quad & \dot{x}(t) = f(x(t)), u(t), t), \quad x(0) = x_0 \\ & c_j(x(t)), u(t), t) \leq 0, \qquad t \in [0, T], j \in \{1, \ldots, k\} \end{aligned} \tag{4.4}$$

becomes

$$\begin{aligned} \text{minimize} \quad & \int_0^T l(x(\tau), u(\tau), \tau) + \sum_j \beta_\delta(-c_j(x(t), u(t), t))d\tau + m(x(T)) \\ \text{subject to} \quad & \dot{x}(t) = f(x(t)), u(t), t), \quad x(0) = x_0 \end{aligned} \tag{4.5}$$

where $\beta_\delta(\cdot)$ is a function that is steep in infeasible solutions and nearly constant in feasible solutions. Given that the inequality constraint functionals must be negative for the solution to be feasible, $\beta_\delta(-c_j(\cdot))$ must be nearly constant for positive values. One possible log barrier functional can be

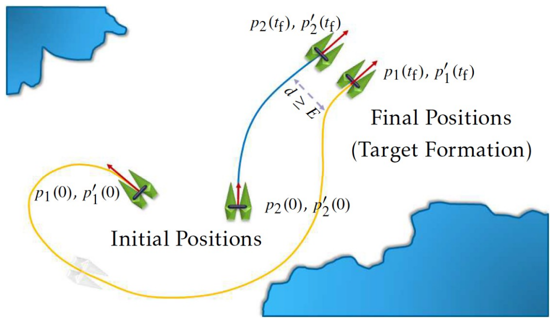
$$\beta_\delta(z) = \begin{cases} -\log z & z > \delta \\ \frac{k-1}{k}\left[\left(\frac{z-k\delta}{(k-1)\delta}\right)^k - 1\right] - \log \delta & z \leq \delta \end{cases} \tag{4.6}$$

which is continuous and whose derivative around $z = \delta$ is continuous too. For full reference see [12].

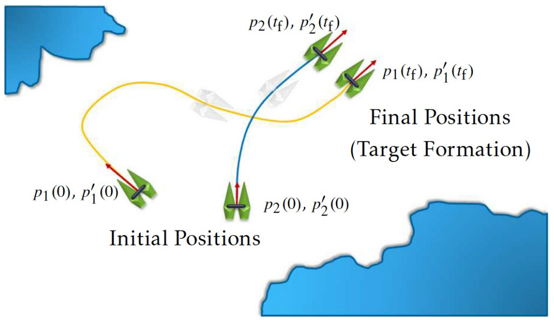
The ”hockey stick” function, with form

$$\delta(z) = \begin{cases} \tanh(z), & z \leq 0 \\ z, & z < 0 \end{cases} \tag{4.7}$$

can be applied to the log barrier functional such that its derivative is smaller for feasible solutions. Figure 4.1 shows the log barrier function with and without the hockey function applied.



(a) Spacial Deconfliction



(b) Temporal Deconfliction

Figure 4.2: Inter Vehicle deconfliction Solutions

4.3 Inter-Vehicular Constraints

There are 2 ways of preventing inter-vehicle collision; *spatial deconfliction* and *temporal deconfliction* [13]. Spatial deconfliction is appropriate for Path Following (PF) and imposes the constraint that the spatial paths of the vehicles under consideration will never intersect and keep a desired safe distance from each other. Temporal deconfliction is appropriate for Trajectory-Tracking (TT) and requires that two vehicles will never be "at the same place at the same time". However, their spatial paths are allowed to intersect. Figure 4.2 illustrates the two types of deconfliction strategies. Temporal deconfliction allows an extra degree of freedom and will intuitively lead to cheaper dynamic costs.

Two method for temporal deconfliction were implemented, one based on sampling the curves and the other based on directly exploiting the properties of Bezier Curves.

For N_v vehicles, any of $\binom{N_v}{2}$ pairs could lead to a collision, therefore, all pairs must be tested, which means a quadratic complexity, therefore, finding fast algorithms to calculate the distance between each pair of trajectories becomes essential.

The easiest method of guaranteeing that the minimum distance between a pair of vehicles is always above a certain value is by calculating the minimum possible distance between each pair of N_v vehicles, which yields a total of $\binom{N_v}{2}$ calculations. If PF is used, then the minimum distance between the paths must be created than a certain value. As a result, these paths cannot intersect. If TT is used, then the minimum distance between vehicles will depend on time $t \in [0T]$, which means that the trajectories must intersect in space but not in time.

4.3.1 Sampling Trajectories

Given that this thesis will focus on the usage of TT and not PF, methods to calculate minimum distances throughout time will be discussed, i.e., minimum distance between trajectories.

The most straightforward way to calculate the minimum distance between two trajectories is to sample each trajectory and calculate the euclidean distance between time equivalent samples. The smallest yields the shortest distance. This method, however, is not perfect because the finer the samples, the more accurate is the result.

One thing to note, however, is that super accurate results of minimum distance between trajectories are not necessary. If, besides knowing the position of the vehicle at a sample, we also know the maximum tangent speed between that sample and the next, then the distance to the other vehicle cannot deviate more than a certain determinable value between those 2 points. It will be smaller than the calculated distance if the vehicles are moving towards each other. The choice of number of samples will determine how big this deviation can be. The optimisation algorithm will stop once it finds a minimum distance that is greater than a certain value, therefore, the number of samples must be big enough such that the deviation is relatively small when compared to the desired minimum distance.

For example, an optimisation problem specifies a minimum distance between 2 vehicles of $1\,\mathrm{m}$. These vehicles are limited to $1\,\mathrm{m\,s^{-1}}$. At a certain point in time between 2 samples, they can be closer to each other than they are at the samples, so lets assume the worst case scenario: half of the time between samples is spent moving towards each other at maximum speed, which implies moving away from each other during the other half of time at maximum speed, therefore, if the time sample is $10\,\mathrm{ms}$, during $5\,\mathrm{ms}$ the vehicles can travel $1\,\mathrm{cm}$ towards each other, at the relative speed of $2\,\mathrm{m\,s^{-1}}$, which is a $1\,\%$ deviation from the established minimum distance of $1\,\mathrm{m}$. If the optimisation problem is reformulated to guarantee $1.01\,\mathrm{m}$, then, with samples spaced by $10\,\mathrm{ms}$, a successful optimisation solution guarantees a minimum distance between vehicles of $1\,\mathrm{m}$. which means a total of 100 samples per second of runtime.

4.3.2 Bezier Curve Distance to a Point

The next method is based on calculating the minimum distance between Bezier curves [14]. This algorithm is adapted to calculate the minimum distance between a curve and a polygon. By subtracting one trajectory to another, which for Bezier curves is explained in section 2.7, finding the minimum distance between trajectories gets translated to finding the closest point of this subtraction curve to the origin. The origin can be interpreted as a 1 point convex shape, therefore, what follows is a brief explanation of an iterative algorithm to calculate the minimum distance of a Bezier curve to a polygon, which will also be used to do collision avoidance with arbitrary convex shapes.

This algorithm consists in recursively breaking down the trajectory into halves by obtaining a new set of control point for each half via de De Casteljau algorithm. For each half, two values are calculated: an upper bound of the minimum distance and a lower bound. The upper bound for the minimum distance will be the closest endpoint of the segment to the polygon, a lower bound is the closest point of the convex hull of the new control points to the polygon. The exit condition of the recursion is when the lower bound is relatively small when compared to the upper bound. The lower bound may be zero if the shapes intersect, which has no influence on the execution of the algorithm. If the exit condition isn't met, the recursion is repeated and the returned value is the smallest of the upper bound along with the time at which the smallest value was found.

4.3.3 GJK Algorithm

The Gilbert–Johnson–Keerthi (GJK) algorithm is a necessary tool for calculating the minimum distance between a Bezier Cuve and a point or polygon, therefore, it is explained here.

The GJK algorithm is an efficient algorithm to calculate minimum distance between arbitrary convex shapes in any dimension.

The GJK algorithm relies heavily on a concept called the Minkowski Sum, but, because the difference operator is used for this algorithm, instead of the sum, the term Minkowski difference will be used. For two shapes A and B, their Minkowski Difference is given by

$$D = A - B = \{a - b | a \in A, b \in B\} \tag{4.8}$$

where D is a new convex shape given by the subtraction of every point in A by every point in B. Figure 4.3 shows 2 shapes on the left hand-side which intersect and the resulting Minkowski Difference on the right hand side. Figure 4.4 shows two shapes that do not intersect and their resulting Minkowski Difference. Notice how the second Minkowski Difference has an identical shape to the first, only differing on its position.

If the Minkowski Difference contains the origin, then the two shapes have common points, i.e., intersect, because the resulting subtraction is zero. As a result, the GJK algorithm is a two part problem: first detect intersection, by testing if the Minkowski Difference contains the origin, then, if it does not, the closest points between A and B result in a Minkowski Point that is closest to the origin, therefore, the second part part of the problem is to look for that closest point to the origin.

For a N-D Minkowski Difference, if a convex shape of up to N+1 vertices that surrounds the origin is found, then the shapes A and B intersect. This shape with up to N+1 vertices is known as a simplex. For 2-D Minkowski Differences, the simplexes can be a point (1 vertex), a line segment (2 vertices) and a triangle (3 vertices). For 3-D spaces, the simplexes can be the same as in 2-D spaces with the addition of a tetrahedron (4 vertices).

The key to GJK's efficiency is to find points in the Minkowski difference that are the best candidates to be in a simplex that can contain the origin.

A support function returns the farthest point in some direction. The resulting point is known as the support point. Finding a support point in the Minkowski Difference along direction d is the same as subtracting the support point of A along d and B along d.

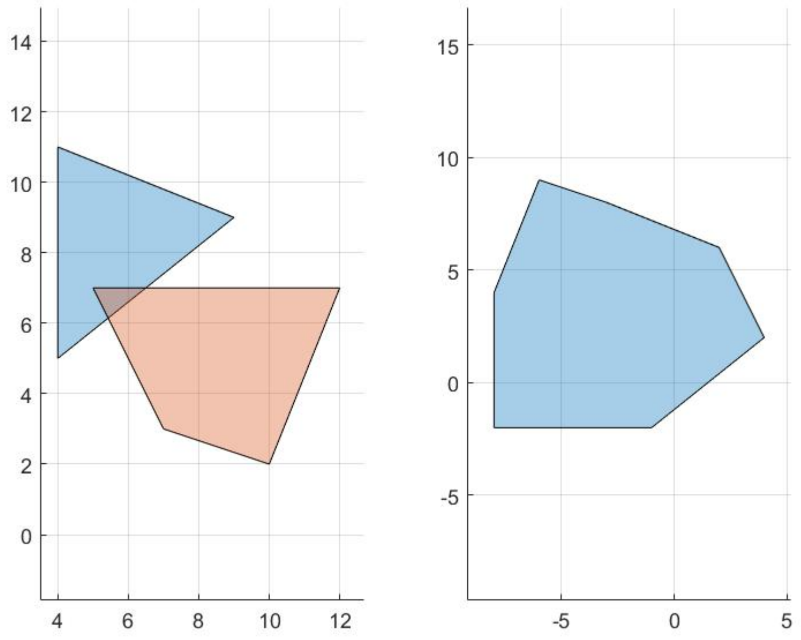


Figure 4.3: Intersecting convex shapes (left) and resulting Minkowsky Difference (right)

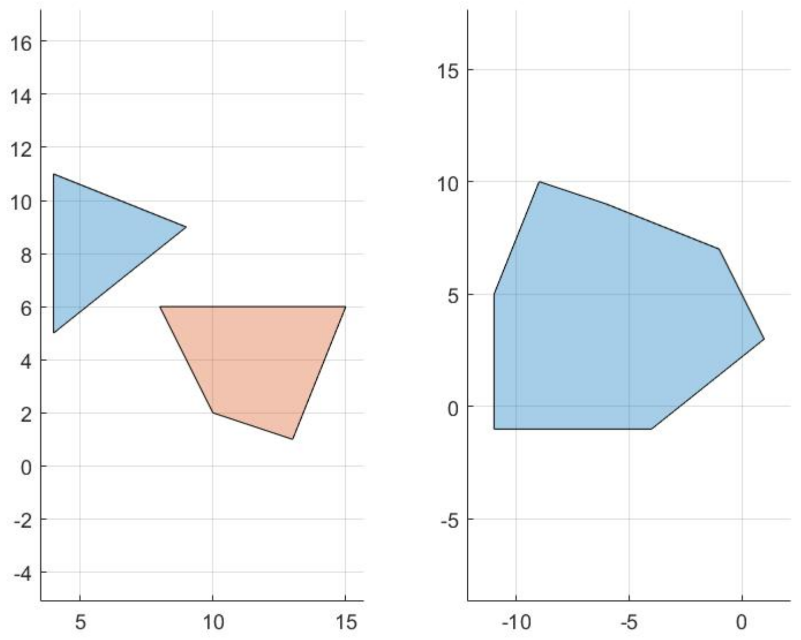


Figure 4.4: Non-intersecting convex shapes (left) and resulting Minkowsky Difference (right)

Choosing the farthest point in a direction has significance because it creates a simplex who contains a maximum area therefore increasing the chance that the algorithm exits quickly. In addition, all the points returned this way are on the edge of the Minkowski Difference and therefore if a point past the origin along some direction cannot be added, the Minkowski Difference cannot contain the origin. This increases the chances of the algorithm exiting quickly in non-intersection cases.

Let W_k be the set of vertices of the simplex constructed in the k^{th} iteration, and v_k as the point in the simplex closest to the origin. Initially, $W_0 = \varnothing$, and v_0 is an arbitrary point of the Minkowski Difference. Since each v_k is contained in the Minkowski Difference, the length of v_k must be an upper bound for the distance.

GJK generates a sequence of simplices in the following way. In each iteration step, a vertex $w_k = s_{A-B}(-v_k)$ is added to the simplex, with the objective of surrounding the origin. If the simplex contains the origin, then the program interrupts because the shapes intersect. If it's proven that the Minkowski Difference cannot contain the origin because the last added vertex did not move "beyond" the origin, then program interrupts and moves on to finding the minimum distance to the origin. If intersection is not proven yet, the new v_{k+1} is perpendicular to the vector given by the last vertex with the one before that, or with the last with the third from the last, if available, depending on which has a dot product greater than 0, and the not used vertex gets removed from the simplex. Alternatively, if no intersection is proven, the new v_{k+1} is the point in the convex hull of $W_k \cup \{w_k\}$ closest to the origin and W_{k+1} becomes the smallest sub-simplex of $W_k \cup \{w_k\}$ that contains v_{k+1}.

4.4 Minimum Distance to Convex Shapes

Earlier, in section 4.3.2, an algorithm to calculate the distance of a trajectory to a polygon was presented. This algorithm, however, is limited to polygons that do not intersect with the trajectory. If the curve intersects with the convex shape, the algorithm returns zero as minimum distance. Optimisation algorithms, like Sequential Quadratic Programming, require the derivative of the constraints to be non zero, even when the current guess for solution is not feasible because this derivate, in other words, will "inform" how far the control points must move so that the solution becomes feasible.

A modification to the algorithm that calculates the minimum distance to a convex shape is presented to calculate the intersection points between the curve and the shape. Afterwards, these intersection points are used to calculate a "penetration" of the curve in the shape.

First thing to note is, during the recursion of the minimum distance algorithm, some endpoints of the cut segments will land inside the shape. If a segment has an endpoint inside and an endpoint outside the shape and the distance between these two points is approximately zero, then the point inside is added to a stack of intersecting points, otherwise, the recursion continues. If the control points of the recursive segments are partially in the shape while others are not, then the recursion continues, otherwise, if the control points are all outside or all inside the shape, then there is no point in continuing because the segment cannot contain any more intersection points.

Once the intersection points are found, the ”penetration” must be calculated. This consists first in finding a convex hull that intersects the shapes. If the number of intersection points is two, then the De Casteljau algorithm is performed twice to find a set of control points for the segment that starts and ends with these two points and the convex hull of these control points is taken, otherwise, the convex hull of all of the intersecting points is used. Once the convex hull is determined, the Directed Extended Polytope Algorithm (DEPA) algorithm explained in section 4.4.1 is performed with the obstacle shape along a predefined direction d. The bigger the penetration depth, the more the trajectory is ”deeper” in the curve.

4.4.1 Directed EPA

If two convex shapes intersect, the GJK algorithm cannot provide collision information like the penetration depth and vector. One algorithm that provides this information is the Extended Polytope Algorithm (EPA). A slight modification for the EPA algorithm is proposed here. It will be referred as the DEPA, whose objective is to find the penetration of one convex shape relative to another along a specific direction d, while the EPA algorithm finds the shortest vector such that the shapes no longer collide. The penetration along a direction is the length of dislocation that the second shape would have to move so that the two shapes no longer collide.

The shapes intersect when the Minkowski contains the origin, therefore, the EPA or the slight variant shown here have as objective dislocating the Minkowski Difference such that it no longer contains the origin. Penetration along a specific direction can be found by calculating the length of the vector that starts in the origin on the Minkowski Difference, has the same direction as d, and stops once it finds the edge of the Minkowski Difference. In other words, this is the norm of the intersection point between a ray starting at the origin with direction d and the edge of the Minkowski Difference. Once this length is found, shape B can move by that length along the direction of d such that it is no longer in collision with shape A.

The process of looking for the penetration along a direction d starts with a polygon which contains the origin, constructed with points along the edge of the Difference. The first step is to find the only edge of this polygon with will intersect with the ray that starts in the origin with direction d. Once this edge is found, the other edges of the polygon are ignored and an iterative process starts. The first step on the iterative process is to calculate a vector which normal to the current intersecting segment and points ”outwards” with respect to the origin. Next, the support function is performed with this vector. The resulting point will be closer to the desired final point. There are now three points in play: the two segment ends and the new point that resulted from the support operation. The next step is to define two segments, one from the one of the segment ends to the new support point, the other from the other segment end to the support point. The next step is to find which of these two new segments intersects with the same ray with direction d and then repeat the iteration.

4.5 Description of the implemented code

The variables for the motion planning problem are each vehicle's state variables and inputs, as explained in section . Each of these variables will be referred to as curves. Optimisation algorithms cannot take continuous functions as variables, therefore, some form of parameterisation of each curve is necessary, as exemplified in some of the algorithms of chapter . Here, each curve will be represented as a Bernstein Polynomial with order N, which will require $N + 1$ control points. A distinction is made between state variables and inputs: state variables must have established initial and final conditions, inputs do not. This implies that for state variables, the initial and final control points must be fixed, therefore, they do not need to participate on the optimisation problem. The following matrix represents how the control points are stored so that they can be accessed by all functions that perform operations on the curves.

$$\begin{bmatrix} x_0^0 & y_0^0 & \psi_0^0 & u_0^0 & v_0^0 & r_0^0 & \tau_{u_0}^0 & \tau_{r_0}^0 \\ x_1^0 & y_1^0 & \psi_1^0 & u_1^0 & v_1^0 & r_1^0 & \tau_{u_1}^0 & \tau_{r_1}^0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_N^0 & y_N^0 & \psi_N^0 & u_N^0 & v_N^0 & r_N^0 & \tau_{u_N}^0 & \tau_{r_N}^0 \end{bmatrix} \tag{4.9}$$

can this be said? because I'm not referencing anything (I'm taking Venanzio's word for it)

It is believed that `fmincon()` performs better when all variables are stored in a line vector, therefore, a function called `matrify()` is necessary in order to transform the flattened optimisation variable to the matrix of equation (4.9).

Elements marked in yellow do not participate in the optimisation algorithm. They are concatenated to this matrix in `matrify()`.

High orders are preferable for each curve [3] because the higher the curve, the closer the control points are to the "matching" point in time of the curve, which is achieved once an optimisation problem finishes. Chapter 5 exemplifies show the control points approximate to the curve and how it is advantageous to produce the dynamics.

The optimisation problem is formulated by constructing a data structure with the fields of table 4.1. Some notes for each of the fields:

- `xi` has as many lines as state variables (not input variables) and as many lines as number of vehicles. Because these functions are designed for vehicles, x and y must be in the first 2 columns
- `xf` works just as `xi`

field	**description**	**mandatory**	**example**
`T`	Time horizon	yes	`10`
`xi`	initial conditions	yes	`[0 0 0 1 0]`
`xf`	final conditions	yes	`[5 5 pi/2 1 0]`
`N`	order of the curves	yes	`15`
`obstacles`	polygons	no default: `[]`	
`obstacles_circles`	circles	no default: `[]`	
`min_dist_int_veh`	minimum distance between vehicles for every point in time	no default: `0`	`.8`
`numinputs`	number of input variables (don't have initial conditions)	no default: `0`	
`uselogbar`	make the problem completely unconstrained and use log barrier functionals	no default: `false`	
`usesigma`	a boolean for the usage of the sigma function if log barrier functionals are to be used	no default: `false`	
`costfun_single`	a function used to calculate the running cost for each singular vehicle	yes	`@costfun`
`dynamics`	a function that describes how the non linear dynamics of the state variables and inputs are linked	yes	`@dynamics`
`init_guess`	a function that provides an initial guess for the optimisation problem which may speed up the process of optimisation	no default: `@rand_init_guess`	`@init_guess`
`recoverxy`	a function that returns the x and y variables by solving just the initial value problem of the inputs	yes	`@recoverxy`

Table 4.1: Description of the constants for optimisation

- obstacles_circles Ncircles by 3, where columns are x, y and radius, respectively
- recoverxy takes an aribtrary X matrix and and a constants structure and returns a Npoints by 2 matrix
- dynamics takes in an arbitrary X matrix and constants structure (to provide pre computer information like a derivation matrix) and must return a column vector which is zeros when all of the dynamic constraints are respected

The data structure for the nonlinear optimisation problem is then passed to a function called run_problem which returns the control points for each of the defined variable along with computation time.

4.5.1 Dynamics

The dynamics are guaranteed with the formulation of 4.3. This means that the code version for the dynamics plus kinematics for the Medusa vehicle, for example, which are 3.15 and 3.16, become

Listing 4.1: Matlab Function

```
ceq = [
    DiffMat*x - u.*cos(yaw) + v.*sin(yaw) - Vcx;
    DiffMat*y - u.*sin(yaw) - v.*cos(yaw) - Vcy;
    DiffMat*yaw - r;
    DiffMat*u - 1/m_u*(tau_u + m_v*v.*r - d_u.*u+fu);
    DiffMat*v - 1/m_v*(-m_u*u.*r - d_v.*v+fv);
    DiffMat*r - 1/m_r*(tau_r + m_uv*u.*v - d_r.*r+fr);
];
```

As explained in section 4.5, Diffmat preserves the order and the equality is maintained in the control points, not the values of the curve itself so some approximation error is expected, which can be minimized with higher orders.

Given that each variable is defined by a set of control points, and, using the convex hull property of Bernstein polynomials, upper and lower bounds for each variable, state or input, become the biggest or smallest control point, respectively.

4.6 Check Soundness with IVP Solution

Once the optimisation process gets completed, the trajectory is defined along with, depending on the used model, the control points for the inputs which can then be re-plugged into the ODE and check what resulting trajectory is obtained, this process is known as solving an IVP. If the approximation is good enough, the resulting trajectory should be nearly identical to the the function for the trajectory.

5 Results

Contents

The following results are based on solving the optimisation problem (4.3) with the implementation described in section 4.5.

Sequential Quadratic Programming [15] will be the nonlinear programming solver of choice. Simulations were run on a 4 × Intel© Core™i5-7200U CPU @ 2.50GHz processor.

Several different running cost functions were tested such as

$$J = \int_0^T \frac{du}{dt}^2 dt \tag{5.1}$$

which minimizes tangent acceleration,

$$J = \int_0^T u^2 dt \tag{5.2}$$

which minimizes speed, and finally, for the Medusa model, specifically,

$$J = \int_0^T \tau_u^2 + \tau_r^2 \tag{5.3}$$

which minimizes the input.

All of these serve as proxies to the minimisation of spent energy.

Results for the two models presented in chapter 3 are presented. The unicycle model has a total of 5 state variables while the Medusa model has a total of 6 state variables plus 2 inputs. Upper and lower bounds for each variable for each vehicle were implemented as explained in section 4.5.1, by finding the biggest and smallest control points. For the examples presented in this chapter, the bounds that were used are those presented in table 5.1 which were chosen to closer represent a real Medusa vehicle.

	Variable	Starting Conditions	Final Conditions
Dubin's Car	x	$-\infty$	∞
	y	$-\infty$	∞
	ψ	$-\infty$	∞
	u	0	1.1
	r	$-\pi/4$	$\pi/4$
Medusa	x	$-\infty$	∞
	y	$-\infty$	∞
	ψ	$-\infty$	∞
	u	0	1.1
	v	$-\infty$	∞
	r	$-.74$	$.74$
	τ_u	0	25.9
	τ_r	-.113	.113

Table 5.1: Upper and lower bounds for each variable of each vehicle model

A significant number of experiments were performed to the optimisation algorithm in order to study its behaviour with changing parameters. Out of all of the experiments that were performed, the most relevant will be presented.

5.1 No obstacles

The first problem will be run for both a single Dubin's car and a single Medusa vehicle, with initial and final states described in table 5.2 and no obstacles. Figures 5.1 and 5.2 show solutions for order 20 and a time horizon of . Each example has a different combination of vehicle model and cost function. Both models contain control points to describe x and y positions, which is what the blue lines show. The red lines describe the solution of the Initial Value Problem as described in section 4.6. This figure, and all that remain, flip x and y axis which standard for marine vehicles.

Variable	Starting Conditions	Final Conditions
x	0	30
y	0	30
ψ	0	$\pi/2$
u	1	1
v	0	0
ω	0	0

Table 5.2: Initial and final conditions for a basic Motion Problem

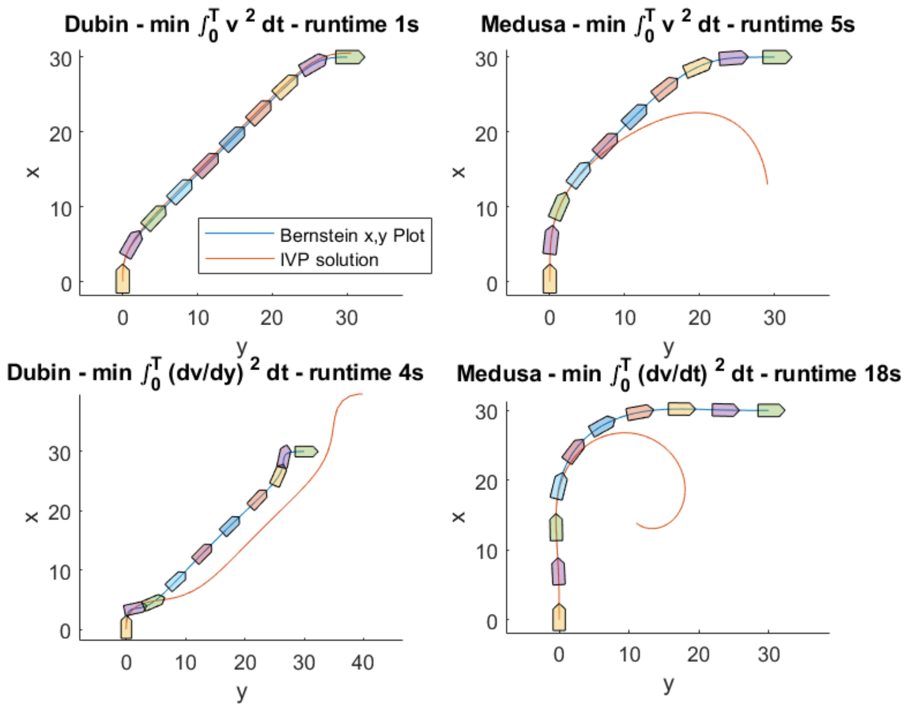


Figure 5.1: Solutions of order $N = 20$ without obstacles and final time $T = 60$

First thing to note is, despite all executions running successfully, i. e., the optimal and feasible solution was found, the Medusa's IVP solution resembles less the plot of the Bezier curve of the x and y control points when compared with the Dubin's car. This suggests that the order isn't high enough

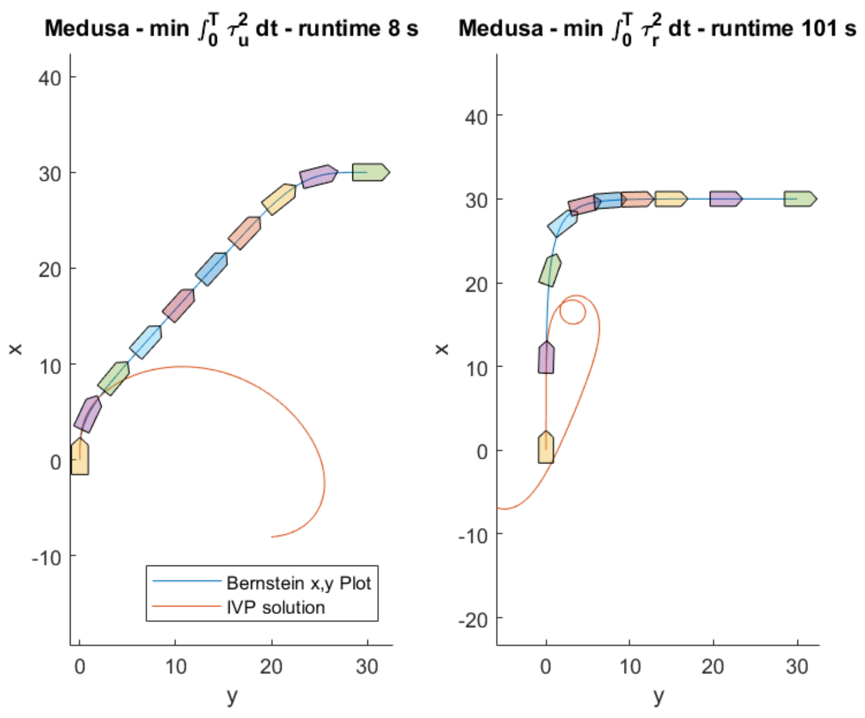


Figure 5.2: Solutions of order $N = 20$ that minimize τ_u^2 (left) and τ_r^2 (right) for the Medusa Model with time horizon $T = 60$

for the curves to accurately represent the real optimal solution's state variables for the Medusa model, which is more complex.

5.2 Obstacles

The following figures show solutions with circle or polygon obstacles whose collision avoidance algorithms are explained in sections 4.3 and **??**. Figure 5.3 is the baseline example without obstacles. It uses a Dubin's Car with the same initial and final conditions as in the previous section and minimises v^2. Figure 5.4, shows the solution with the added circle as a constraint. Figure 5.5 shows the solution with an obstacle but the constraint was moved to the log barrier functional as explained in section 4.2. Figure 5.6 shows the solution with log barrier funcitonal as well but the relative weight of the log barrier on the cost wasn't as big, and, as a result, the optimisation problem terminated successfully but did not prevent collision. The runtimes of these examples don't show how the usage of log barrier provides an advantage, however, for a polygon obstacle, such as in figures 5.7 and 5.8 show a huge difference in the usage of the log barrier functional. Both these results have a huge runtime when compared to circular obstacles because the algorithm is iterative.

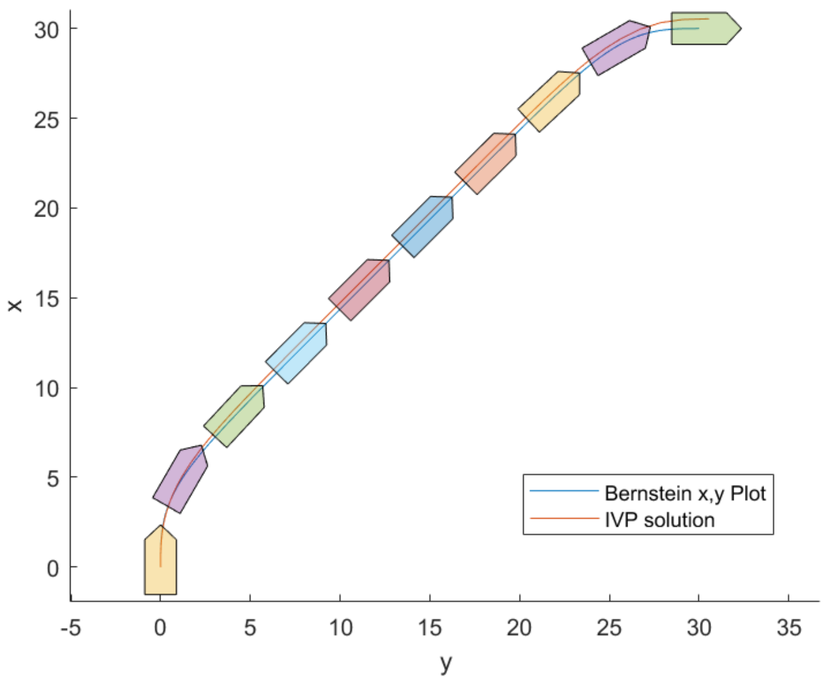


Figure 5.3: Solution of order $N = 20$ without obstacles
computation time = 2s

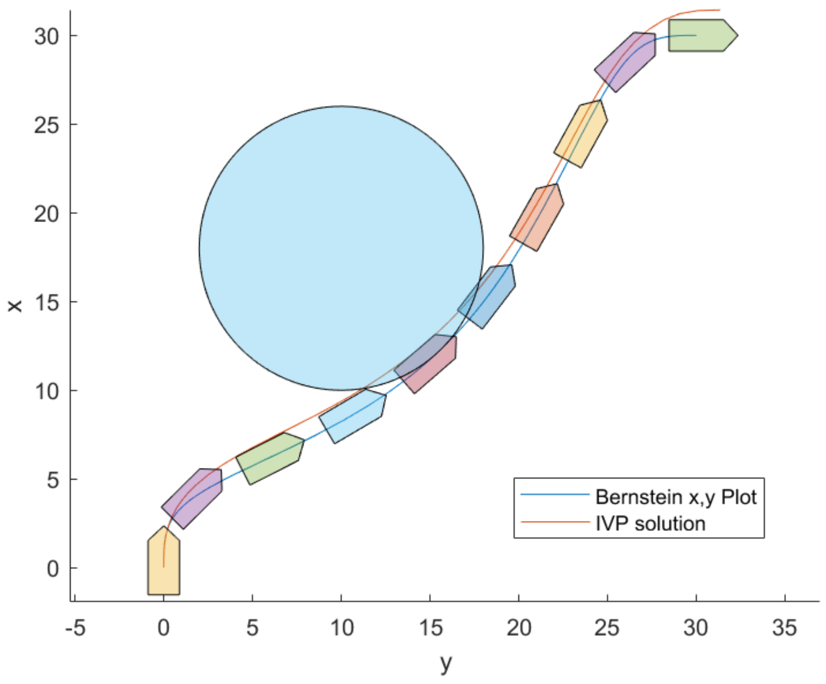


Figure 5.4: Circle osbtacle: computation time 5s

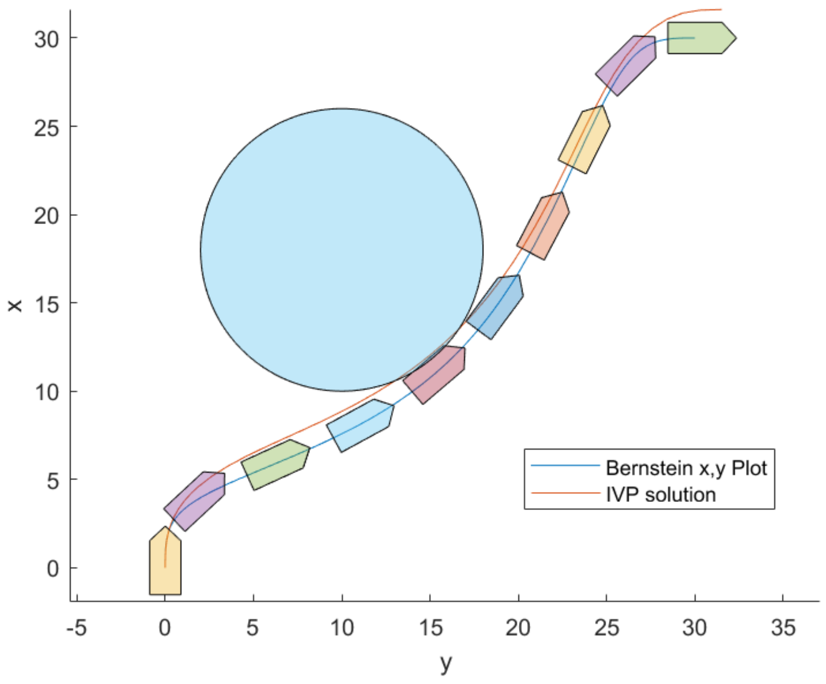


Figure 5.5: Circle obstacle plus log bar: computation time 4s

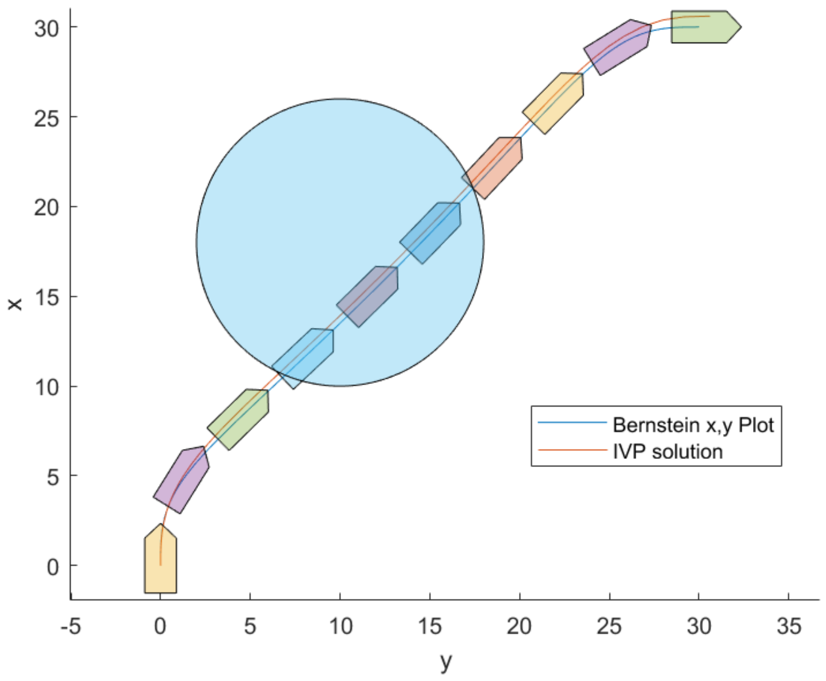


Figure 5.6: Circle obstacle plus log bar: computation time 3s

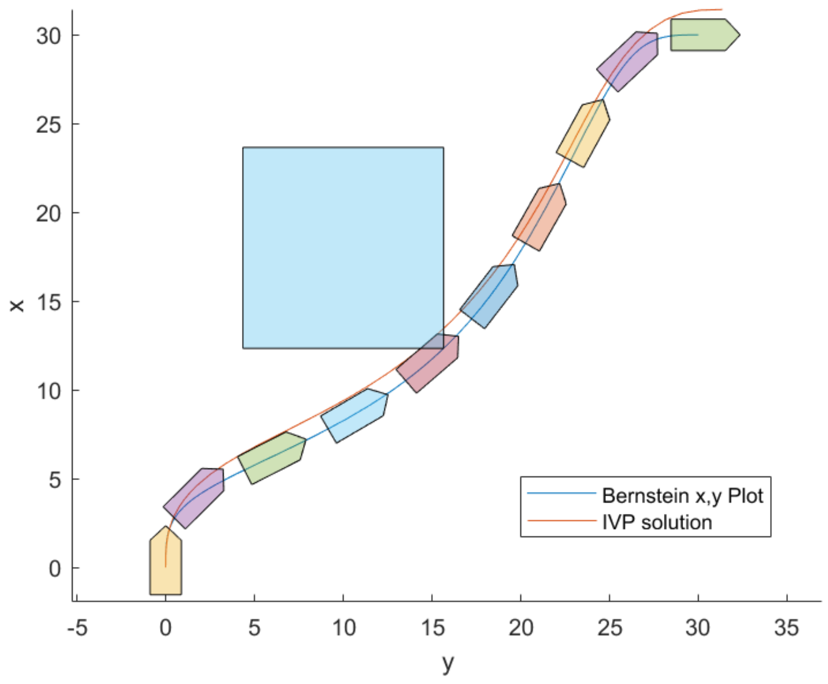


Figure 5.7: polygon obstacle: computation time 307s

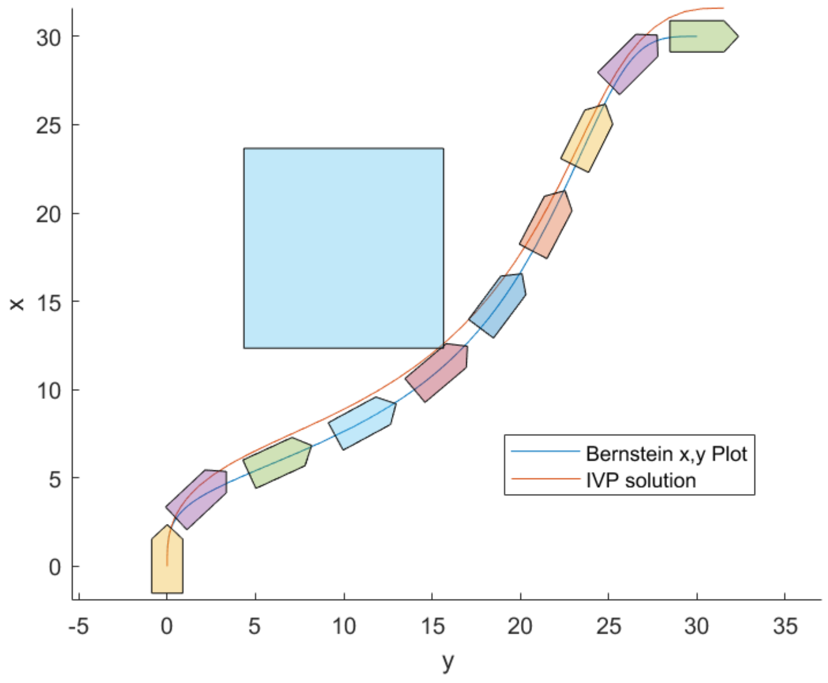


Figure 5.8: polygon obstacle: computation time 157s

5.2.1 Variation of cost with order

The next step was to implement a way of calculating solutions for high orders while maintaining low computation time. This is acheived by taking the solution of a low order, perform degree elevation and re-feed that solution as initial guess for a higher order. Figure 5.9, show some solutions of this process. The top left figure started is the solution order 10, this solution has it's order increased by 10, and used as initial guess for another run resulting in the top right figure and so on and so on. The iterative process stopped with order 70 because the relative final cost differs from order 60 by less than 1%. The same solution of oder 70 took a total of 334 seconds when using a random initial guess which shows how this iterative method can save computation time.

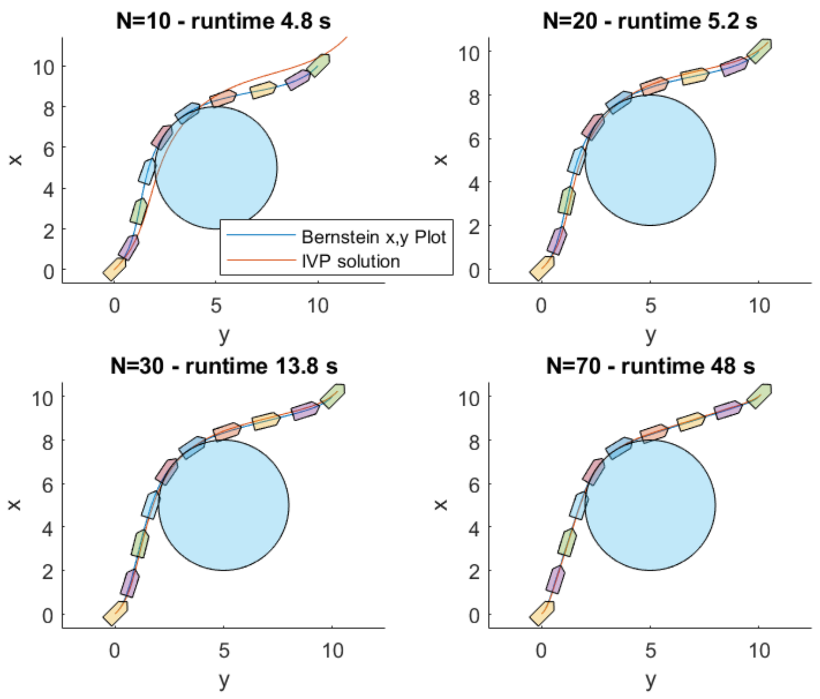


Figure 5.9: Results of Interatively increasing order

We can see how to cost varies with increase of order N and how the solution of the IVP becomes closer and closer to the plot of xy.

5.3 Multiple Vehicles

In figure 5.10 we can see how computation time quickly grows with even a small number of vehicles for a Medusa model and using the sampling approach for deconfliction discussed in 4.3.

And at last an example with 3 vehicles and 1 obstacle, in figure 5.11

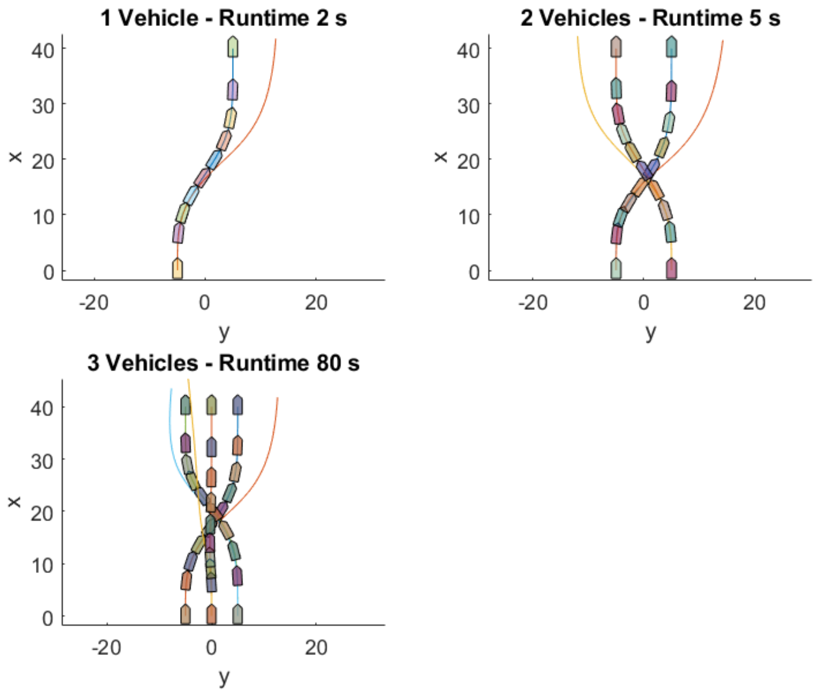


Figure 5.10: Solutions of order $N = 10$ with multiple vehicles

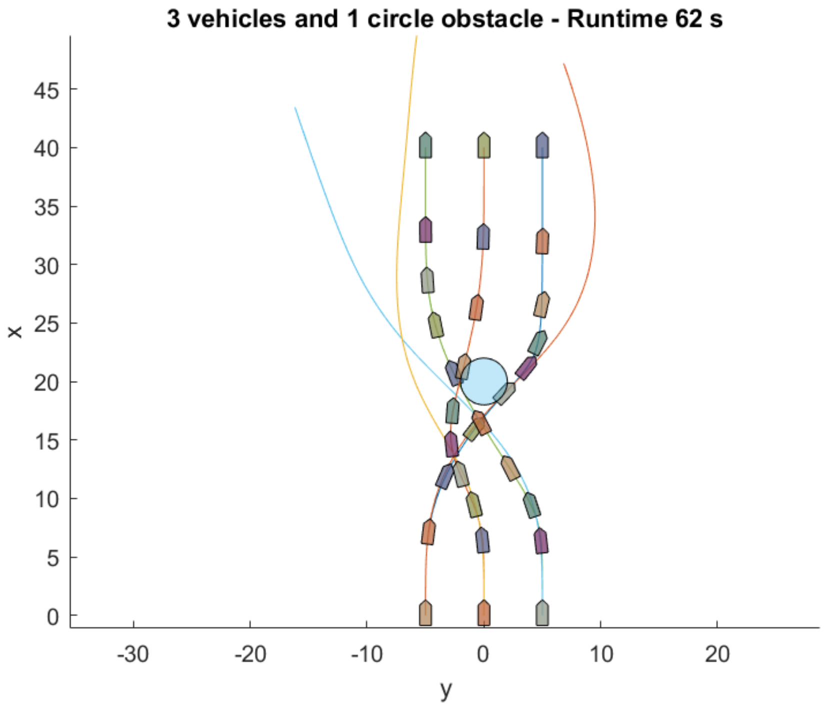


Figure 5.11: Solution of order $N = 10$ with 3 vehicles and 1 circle obstacle

6

Conclusion

In this work, a fast optimal motion planning algorithm was designed. It consisted in solving an optimal control problem, by finding its equivalent optimisation problem. The algorithm's final form was chosen based on the comparison of several parameterisation methods and how well each one can handle dynamic and environmental constraints. Bezier curves was the final choice of parameterisation to approximate the optimal trajectory. They have very good properties for motion planning that greatly simplify the optimisation problem.

Two AUV models, the unicycle and the Medusa, were studied and afterwards, how can their characteristics reap the benefits from the proposed motion planning algorithm. Specifically, it is shown that the use of a Bernstein based polynomial representation of trajectories is no longer limited for differentially flat systems and how, in fact, defining all of the state variables and inputs and linking them via the dynamics simplifies the computation of running costs and calculating feasibility of the solution.

Results also show, however, that time complexity quickly increases with a high number of vehicles, specially when a high order of approximation is used for every vehicle's state and input variables. These time complexity limitations could potentially be negligible if a more advanced computational power is available.

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 active navigation localization. The tools developed to plan optimal trajectories can now be explored to conduct further research.

Bibliography

[1] P. Abreu, G. Antonelli, F. Arrichiello, A. Caffaz, A. Caiti, G. Casalino, N. C. Volpi, I. B. De Jong, D. De Palma, H. Duarte, et al., "Widely scalable mobile underwater sonar technology: An overview of the h2020 wimust project," Marine Technology Society Journal, vol. 50, no. 4, pp. 42–53, 2016.

[2] G. G. Lorentz, Bernstein polynomials. American Mathematical Soc., 2013.

[3] V. Cichella, I. Kaminer, C. Walton, N. Hovakimyan, and A. Pascoal, "Bernstein approximation of optimal control problems," arXiv preprint arXiv:1812.06132, 2018.

[4] P. C. Abreu, J. Botelho, P. Góis, A. Pascoal, J. Ribeiro, M. Ribeiro, M. Rufino, L. Sebastião, and H. Silva, "The medusa class of autonomous marine vehicles and their role in eu projects," in OCEANS 2016-Shanghai, IEEE, 2016, pp. 1–10.

[5] Wikipedia contributors, Dubins path — Wikipedia, the free encyclopedia, [Online; accessed 19-October-2020], 2019. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Dubins_path&oldid=918571120`.

[6] M. Diehl, H. G. Bock, H. Diedam, and P.-B. Wieber, "Fast direct multiple shooting algorithms for optimal robot control," in Fast motions in biomechanics and robotics, Springer, 2006, pp. 65–93.

[7] J. Riccati, "Animadversiones in aequationes differentiales secundi gradus (observations regarding differential equations of the second order)," Actorum Eruditorum Supplementa, vol. 8, no. 1724, pp. 66–73, 1724.

[8] A. V. Rao, "A survey of numerical methods for optimal control," Advances in the Astronautical Sciences, vol. 135, no. 1, pp. 497–528, 2009.

[9] B. Sabetghadam, R. Cunha, and A. Pascoal, "Cooperative motion planning with time, energy and active navigation constraints," in 2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV), IEEE, 2018, pp. 1–6.

[10] M. Fliess, J. Lévine, P. Martin, and P. Rouchon, "Flatness and defect of non-linear systems: Introductory theory and examples," International journal of control, vol. 61, no. 6, pp. 1327–1361, 1995.

[11] T. Fossen and A. Ross, “Nonlinear modelling, identification and control of uuvs,” in Peter Peregrinus LTD, 2006, ch. 2.

[12] J. Hauser and A. Saccon, “A barrier function method for the optimisation of trajectory functionals with constraints,” in Proceedings of the 45th IEEE Conference on Decision and Control, IEEE, 2006, pp. 864–869.

[13] A. J. Häusler, “Mission planning for multiple cooperative robotic vehicles,” Ph.D. dissertation, Ph. D. thesis, Department of Electrical and Computer Engineering, Instituto Superior Técnico, 2015.

[14] J.-W. Chang, Y.-K. Choi, M.-S. Kim, and W. Wang, “Computation of the minimum distance between two bézier curves/surfaces,” Computers & Graphics, vol. 35, no. 3, pp. 677–684, 2011.

[15] N. I. M. Gould and P. L. Toint, “Sqp methods for large-scale nonlinear programming,” in System Modelling and optimisation, M. J. D. Powell and S. Scholtes, Eds., Boston, MA: Springer US, 2000, pp. 149–178, ISBN: 978-0-387-35514-6.