TÉCNICO LISBOA

Motion Planning for Cooperative Autonomous Robots using Optimization Toos

Thomas David Pamplona Berry

Thesis to obtain the Master of Science Degree in

Electrical And Computer Engineering

Supervisor: Prof. António Manuel dos Santos Pascoal

October 2020

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring 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 thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisors Prof. Some Name and Prof. Some Other Name for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

Abstract

This is from IIEEC

This report presents an initial investigation of some of the methods for cooperative motion control as preparation for the study of the control of fleets of underwater autonomous robots. This will be carried out throughout the second semester of 2019/2020 and will be the subject of a master's thesis in Decision Systems and Control. 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 single vehicles in a 2D space. These programs will be extended for calculating trajectories for multiple cooperative robots. The programs will eventually be implemented in autonomous robots of the medusa class from ISR-Técnico.

Keywords

Maecenas tempus dictum libero; Donec non tortor in arcu mollis feugiat;Cras rutrum pulvinar tellus.

Resumo

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra. Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet, wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in turpis pulvinar facilisis. Ut felis. Aliquam aliquet, est a ullamcorper condimentum, tellus nulla fringilla elit, a iaculis nulla turpis sed wisi. Fusce volutpat. Etiam sodales ante id nunc. Proin ornare dignissim lacus. Nunc porttitor nunc a sem. Sed sollicitudin velit eu magna. Aliquam erat volutpat. Vivamus ornare est non wisi. Proin vel quam. Vivamus egestas. Nunc tempor diam vehicula mauris. Nullam sapien eros, facilisis vel, eleifend non, auctor dapibus, pede.

Palavras Chave

Colaborativo; Codificaçãoo; Conteúdo Multimédia; Comunicação;

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

DOF	Degress of Freedom
AUV	Autonomous Underwater Vehicle
TT	Trajectory-Tracking
PF	Path Following
GJK	Gilbert–Johnson–Keerthi
EPA	Extended Polytope Algorithm
DEPA	Directed Extended Polytope Algorithm

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. [1] A key-enabling element for the execution of such missions is the availability of advanced methods for cooperative motion planning that take explicitly into account temporal and spatial constraints, intrinsic vehicle limitations and energy minimisation requirements.

Some autonomous vehicle applications require groups of robots acting cooperatively. An application example that will greatly benefit from vehicle cooperation and that will be focus on this thesis is the control of groups of Autonomous Underwater Vehicles (AUVs) where the visibility is low and obstacles are not known in advance. The AUVs that together form a robot formation, can, for example, 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.

This project has evolved from earlier investigations on underwater mapping connected with the European R&D Project "MORPH" Project [2], [3] and the "WiMUST" Project [4] that Insitituto Superior Técnico was a part of. Figure 1.2 illustrates AUVs in operation on the Morph project.

The "WiMUST" project, in particular, was composed by a small fleet of AUVs towing streamers with hydrophones to acquire sub-bottom profiling acoustic data.

Recent advancements on the usage of *Bernstein Polynomials* for control also appear to be advantageous, [5], shows that it's possible to control a high number of vehicles with small computation time.

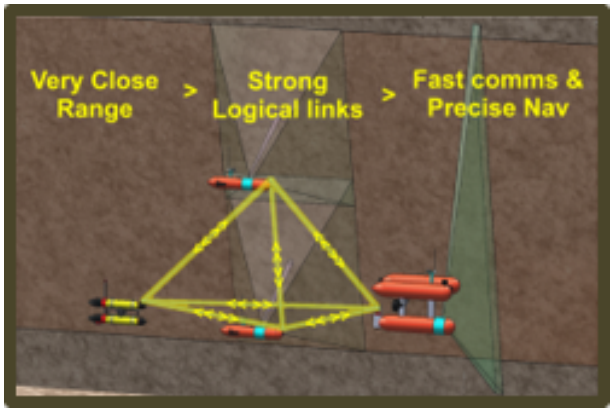


Figure 1.1: Morph Project

1.2 Background

The work discussed in this thesis emphasises in the use of motion planning for multiple cooperative vehicles. Motion Planning, as the name suggests, consists in planning motion for robots, such as mobile

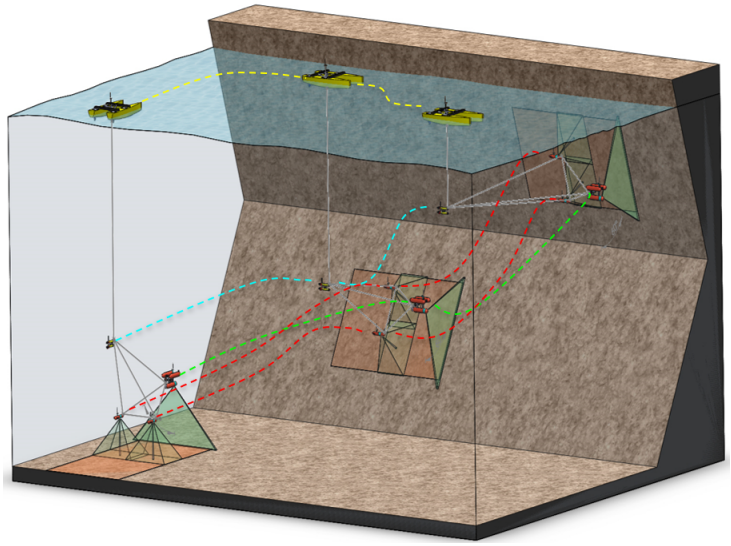


Figure 1.2: Morph Project

vehicles or robotic arms. The choice of control law, i. e., the input which is provided to a robot, will result in a motion which is optimal according to a certain criteria. This criteria could be, for example, minimising time or consumed fuel. Formulisation of such a problem is known as an optimal control problem.

There are two main families of techniques for solving optimal control problems: direct methods and indirect methods.

Indirect methods

Direct methods in optimal control convert the optimal control problem into an optimization problem of a standard form and then using a nonlinear program to solve that optimization problem.

Different models to represent vehicles exist such as a Dubin's car, Medusa Model.

1.3 Objectives

There are several ways to obtain a trajectory, 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.

- test some methods for obstacle avoidance

- compare some different parameterization methodoligies
- analize the complexity of increasing order and number of vehicles

This

explain that we want to show that bernstein methods work for non differentially flat models like medusa

1.4 Overview

A path is a parameterized curve, which is a function that maps a segment $[a, b]$ to $\mathbb{R}^3$. If the parameter of path p is time or a function of time, the map $t \mapsto p(t)$ is called a trajectory.

Path Following (PF) refers to the problem of making vehicles converge to and follow a path with no explicit temporal schedule while Trajectory-Tracking (TT) is the problem of making a vehicle track a trajectory such that both spacial and temporal schedules are satisfied simultaneously.

Motion Planning consists in the design of trajectories for different kinds of systems that can later be tracked.

A trajectory is the path that an object with mass in motion follows through space as a function of time. Trajectory tracking is the

The objective of motion planning is to find a trajectory to be tracked by a robot in an optimal way, based on a "cost function".

1.5 Problem Statement

Discuss the difference between trajectory tracking and motion planning

Discuss the different models for vehicles

Maybe discuss optimization software like fmincon (J Garcia does this)

Maybe discuss the use of log barrier function

1.6 Thesis Outline

In chapter 4.1, an overview of the different available numerical methods for the motion planning for a single vehicle will be presented.

exapnd

In chapter 4.1, 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

In chapger 6, the conclusion is made.

exapnd

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, methods for trajectory optimisation will be reviewed. These methods are generic, i.e., they can be applied to a wide range of dynamic systems. In this study, they will be applied to the control of a simple model for 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 Probelm

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

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

where x_0 is the initial state of the vehicle, the differential equation describes the model for the motion of the vehicle, $h(x(t),\ u(t))$ represents the path constraints, and $r(x(T)) = 0$ is the terminal constraint. T is known as the time "horizon", i. e., no matter what motion planning is used, the produced control law will only be valid within time $t \in [0; \infty]$.

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

$$h(x) = \|x - p_o\| - d_{\text{min}} \tag{2.2}$$

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

All of the available methods studied here will try to produce a control law that optimises some form of the above cost function.

In order to illustrate these methods, the control of a double integrator in 1 and 2 dimensions will be presented.

A double integrator is a linear system where the second derivative of the position variable y, is the acceleration a, that is,

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

The variables for the state-space form for the double integrator are given by

$$\begin{cases} x_1 = y \\ x_2 = v \\ u = a \end{cases} \tag{2.4}$$

where x_1 and x_2 are state variables that represent position and velocity, respectively, and u is the system input, acceleration.

The system dynamics can be represented in state-space form as

$$\begin{cases} \dot{x}_1 = \dot{x}_2 \\ \dot{x}_2 = u \end{cases} \tag{2.5}$$

or

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{2.6}$$

in matrix form.

A general linear system is represented by

$$\dot{x} = Ax + Bu \tag{2.7}$$

As a result, for a double integrator, the matrices A and B 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 x_1 and x_2 are variables of state x, u is the input fed to the system, and ρ is a scalar that penalizes "energy" consumption.

This dynamic system is deferentially flat 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 simpler and for demonstation purpose, uni dimensional problems still have a place in real life applications. One easy example is a train what follows a track. It cannot leave the track therefor, environmental constraints (obstacles) cannot be persistent if they are between it's initial and desired position. A non-persistent constraint could be another train that is joining the same line, therefor, the first train cannot be in certain positions in certain points in time.

worth mentioning this?

2.2 differentially Flat Systems

A nonlinear system of the form

$$\dot{x} = f(x(t), u(t)), \quad x(t) \in \mathbb{R}^n \tag{2.10}$$

is said to be *differentially flat* [7] or simply *flat* if there exsists a set of variables $y = (y_1, \ldots, y_m)$ called the *flat output*, such that

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

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

This implies that there is a fictitious *flat output* that can explicitly express all states and inputs in terms of the flat output and a finite number of derivatives.

2.3 Linear Quadratic Regulator

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

$$\dot{x} = Ax + B \tag{2.11}$$

where x is the state, u is the input, and A and B are state and input matrices of appropriate dimensions.

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

$$J = \int_0^T x^T Q x + u^T R u + 2 x^T N u dt \tag{2.12}$$

The stabilising control law is of the form

$$u = -Kx \tag{2.13}$$

where K is given by

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

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

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

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

$$A^T P + PA - (PB + N)R^{-1}(B^T P + N^T) + Q = 0 \tag{2.16}$$

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 and reducing the effect of external disturbances.

In order to visualise the resulting control input that will be fed to a system as a function of time, the ODE that combines 2.11 and 2.13 can be solved when provided with initial conditions.

`Matlab` implements the solution of the quadratic regulator in the functions `lqr` (for continuous systems) and `dqlr` (for digital systems).

The first presented example of control of the double integrator will be done in the context of linear quadratic regulator theory. Here, the initial conditions, which will be the same for all the 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 2.14 so that the resulting cost matches 2.9.

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

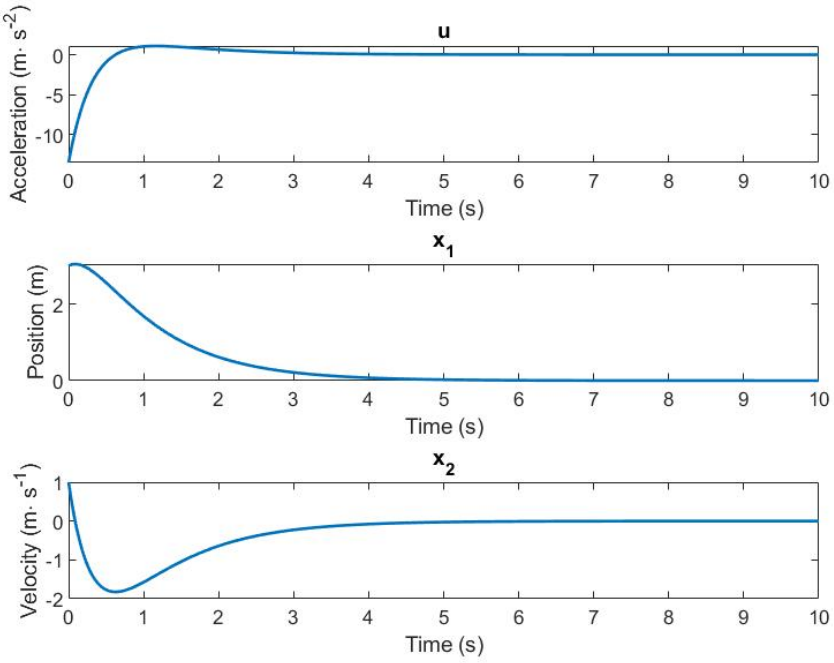


Figure 2.1: Result for a Linear Quadratic Regulator

2.4 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 ODE:

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

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 $q = (q_0, q_1, \ldots, q_{N-1})$ 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, a huge concern when choosing a trajectory planning method is 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 itegration 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.

2.5 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

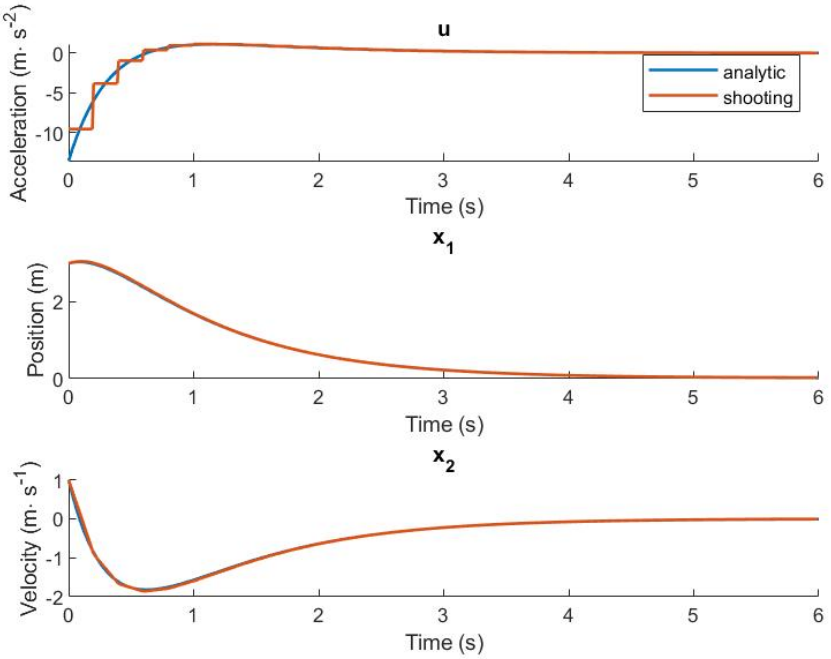


Figure 2.2: Direct Optimisation via a piecewise constant input
Computation time: 2.44 s

be added to the problem. This constraint is given by

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

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

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

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.

2.6 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 2.12 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

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

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 aditional 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.

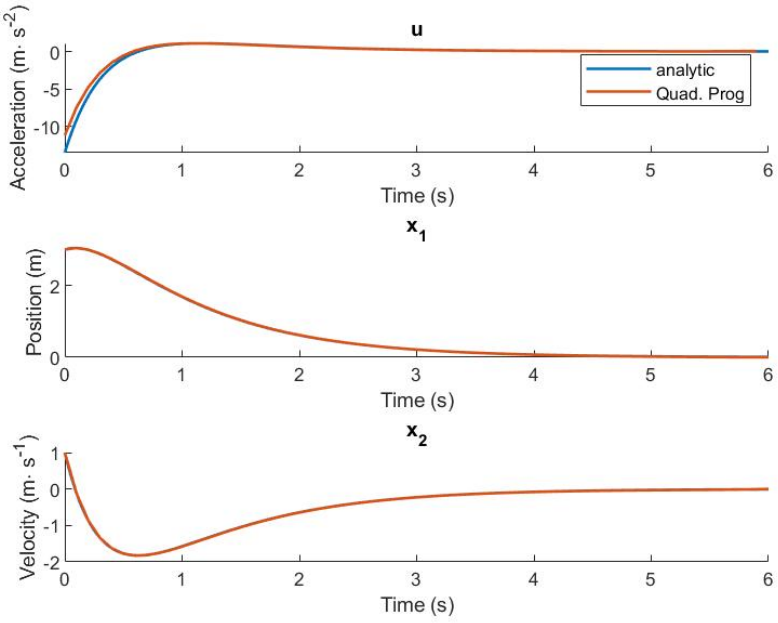


Figure 2.3: Quadratic Programming solution

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

$$P(\tau) = \sum_{k=0}^{n} p_k B_{k,n}(\tau), \qquad \tau \in [0,1] \tag{2.35}$$

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

$$\tau = \frac{t}{T}, \qquad t \in [0,T] \tag{2.36}$$

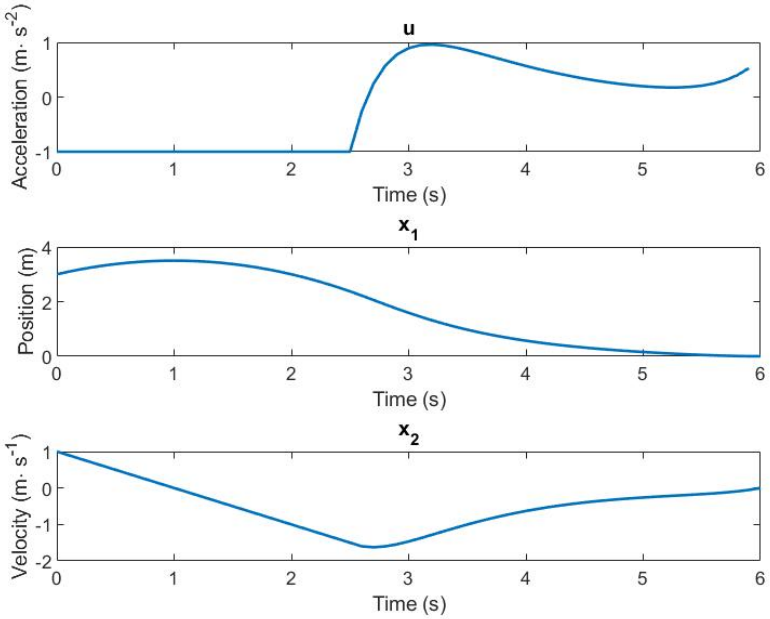


Figure 2.4: Quadratic Programming solution with bounded input

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

$$B_k^n(\tau) = \binom{n}{k}(1-\tau)^{n-k}\tau^k \tag{2.37}$$

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 oders, n, are ploted, on in each subplot of figure 2.5.

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

$$p = \begin{bmatrix} 0 \\ 0.5 \\ 1 \\ 0.7 \\ 0.3 \\ -0.7 \\ -1 \\ -0.5 \\ -0.1 \end{bmatrix} \tag{2.38}$$

The plot of the polynomials respective to vector p is in figure 2.6. The control points on vector p are plotted along time time axis unifomally. What can be seen is that the control points ”attract” the curve towards themselves.

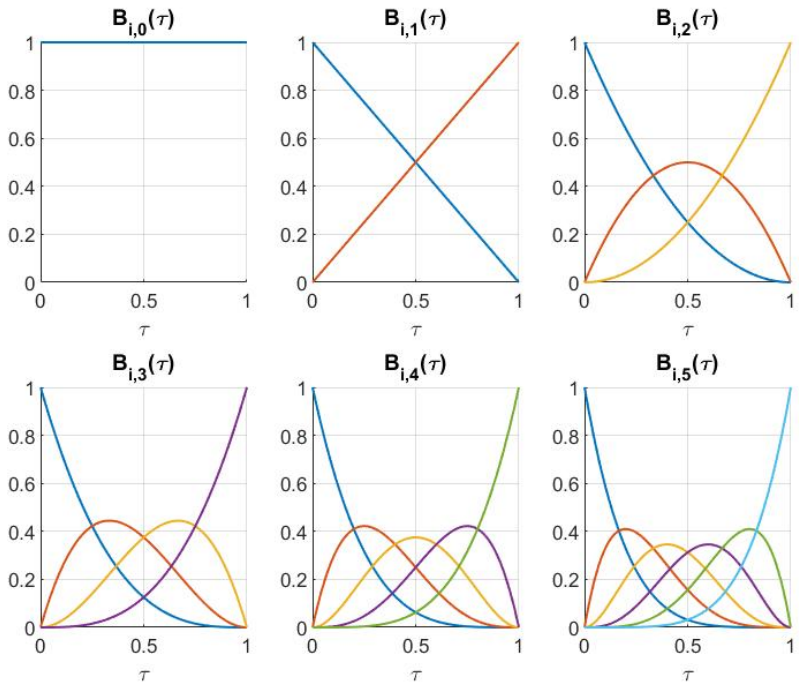


Figure 2.5: Representation of the Bernstein Basis for $\tau \in [0, 1]$ for oders 0 to 5

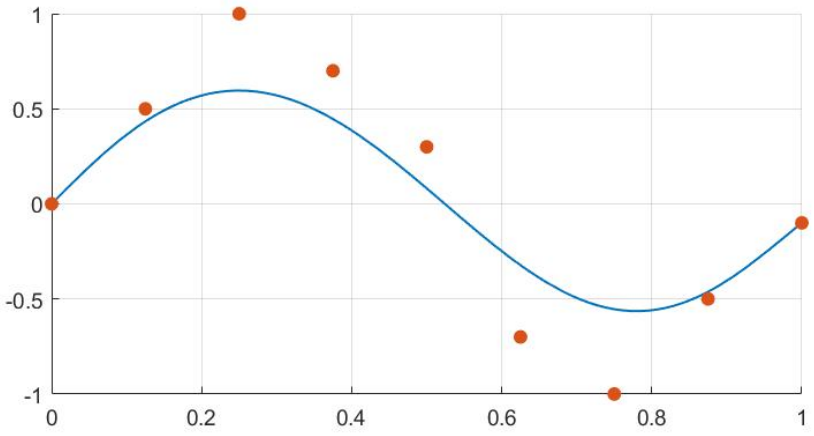


Figure 2.6: A Bernstein Polynomial

Two Bernstein Polynomials of the same order, i.e., same number of control points, can be plotted against eachother within the time domain $\tau \in [0, 1]$. The control points of the curvs together form $n + 1$ points in 2 dimensions. These are plotted together with the example 2-D plot of 2.7. 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 2.8 shows how the same control points can form a completely different curve.

Figure 2.7 shows a two dimensional Bernstein polynomial.

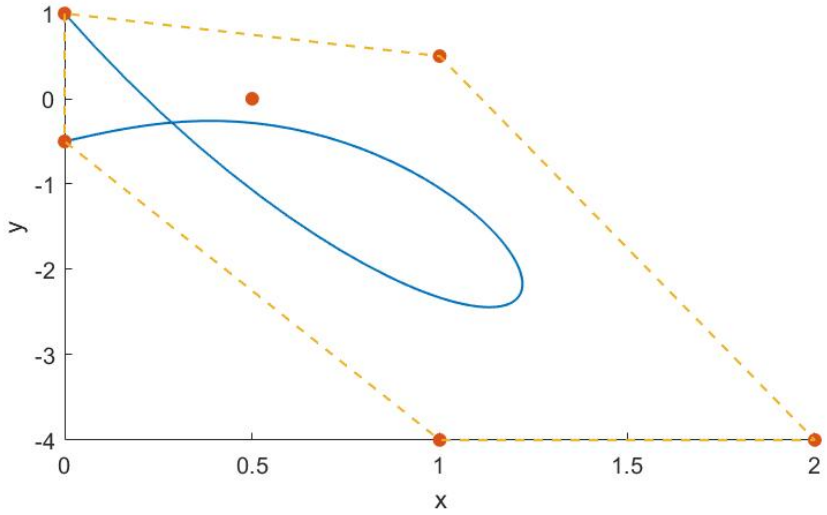


Figure 2.7: A 2D Bernstein Polynomial

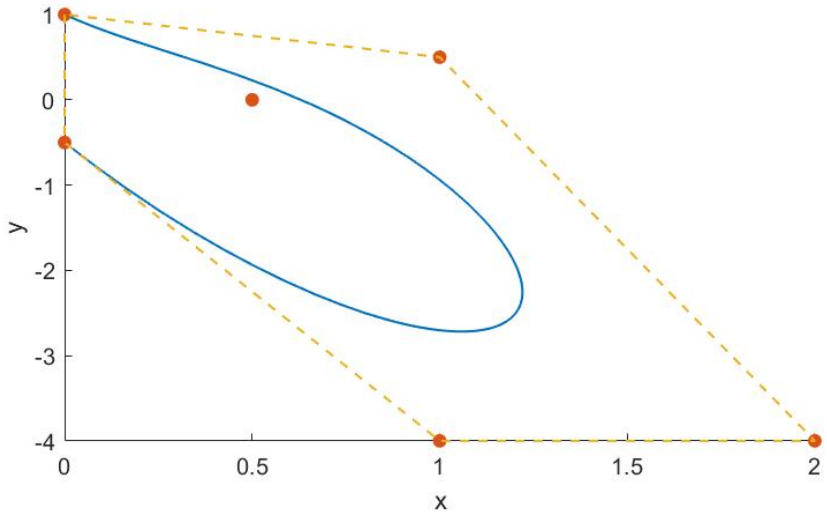


Figure 2.8: A 2D Bernstein Polynomial

One of the first properties than can be observed in figure 2.6 is that, within the domain $t \in [0, T]$, the polynomial is limited by a convex hull formed by the control points.

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

$$\begin{aligned} P(0) &= p_{n,0} \\ P(T) &= p_{n,n} \end{aligned} \tag{2.39}$$

and the derivative and integral are given by

$$\dot{P}(t) = \frac{n}{T}\sum_{k=0}^{n-1}(p_{k+1,n} - p_{k,n})B_{k,n-1}(t) \tag{2.40}$$

$$\int_0^T P(t)dt = \frac{T}{n+1}\sum_{k=0}^{n} p_{k,n} \tag{2.41}$$

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

$$\text{Derivation Matrix} = \begin{bmatrix} 1 & -1 & & \dots & & \\ & 1 & -1 & \dots & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & \dots & & -1 & 1 \end{bmatrix} \tag{2.42}$$

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

$$p = A \cdot \dot{p} + p_0 \tag{2.43}$$

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

$$\begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ 1 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 1 & 1 & \dots & 1 & 1 \end{bmatrix} \tag{2.44}$$

Multiplication and addition is given by

$$f(t)g(t) = \sum_{i=0}^{m+n}\left(\sum_{i=\max(0,i-n)}^{\min(m,i)} \frac{\binom{m}{j}\binom{n}{i-j}}{\binom{m+n}{i}} f_{j,m}g_{i-j,n}\right) B_{i,m+n}(t) \tag{2.45}$$

$$f(t) \pm g(t) = \sum_{i=0}^{m}\left(f_{i,n} \pm \sum_{j=\max(0,i-m+n)}^{\min(n,i)} \frac{\binom{n}{j}\binom{m-n}{i-j}}{\binom{m}{i}} g_{j,n}\right) B_{i,m+n}(t) \tag{2.46}$$

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

explain how the deCasteljau algorithm can obtain control points for each half of the segment

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 2.9 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.

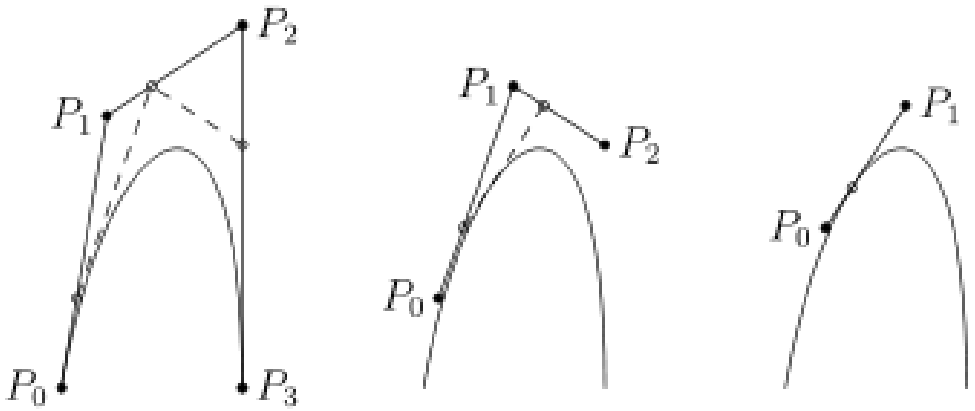


Figure 2.9: Visual representation of the deCasteljau algorithm

The use of Bernstein polynomials as an alternative to monomial polynomials, as presented in the previous section, is preferable (see [5]).

The convex hull property of these polynomials will prove useful in an algorithm that calculates distance between curves, which will be used in de-confliction 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 state variables need to be defined and optimised but will be subject to the constrained imposed by the system's dynamics. The optimization problems developed for motion planning in this work center around the usage of what is known as Bernstein polynomials.

2.8 Minimum Distance Between Trajectories

For N_v vehicles, any of $\binom{N_v}{2}$ pairs could lead to a collision, therefor, all pairs must be tested, which means a quadratic complexity, therefor, 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 greated than a certain value. As a result, these paths cannot intersect. If TT is used, then the minimum distance between vehilces will depend on time $t \in [0T]$, which means that the trajectories must intersect in space but not in time.

2.8.1 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 eachother. The choice of number of samples will determine how big this deviation can be. The opitmisation algorithm will stop once it finds a minimum distance that is greater than a certain value, therefor, 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 eachother than they are at the samples, so lets assume the worst case scenerio: half of the time between samples is spent moving towards eachother at maximum speed, which imples moving away from eachother during the other half of time at maximum speed, therefor, if the time sample is $10\,\mathrm{ms}$, during $5\,\mathrm{ms}$ the vehicles can travel $1\,\mathrm{cm}$ towards eachother, 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 guranatee $1.01\,\mathrm{m}$, 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.

2.8.2 Bezier Curve to a point

The next method is based on calculating the minimum distance between Bezier curves [9]. This algorithm is addapted 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, therefor, 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 deCasteljau 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.

2.8.3 GJK Algorithm

The Gilbert–Johnson–Keerthi (GJK) algorithm is an effitiend 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

$$D = A - B = \{a - b | a \in A, b \in B\} \tag{2.47}$$

where D is a new convex shape given by the subtraction of every point in A by every point in B. Figure shows 2 shapes on the left handside which intersect and the resulting Minkowski Difference on the right hand side. Figure 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, therefor, the second part part of the problem is to look for that closest point to the origin.

figure of 2 intersecting shapes plus figure

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 effitiency 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.

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 availabe, depending on which has a dot product greater than 0, and the not used vertex gets removed from the simplpex. 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}.

2.8.4 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 refered as the Directed Extended Polytope Algorithm (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, therefor, 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 orgin with direction d. Once this edge is found, the other edges of the polygon are ignored and an interative 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 resuting 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.

2.9 Minimum Distance to Convex Shapes

Earlier, in section 2.8.2, 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. Aftwards, these intersection points are used to calculate a ”penetration” of the curve in the shape.

First thing to note is, during the recusion 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 anymore 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 deCasteljau 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 DEPA algorithm explained in section 2.8.4 is performed with the obstacle shape along a predifined direction d. The bigger the penetration depth, the more the trajectory is ”deeper” in the curve.

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 (3.2). Finally, the motion equations for the MEDUSA model are presented (the characterization of the vehicle MEDUSA (3.3).

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

3.1 Reference Frames and Notaion

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 axix (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 intertial 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 difined 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

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

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

The first step towards describing the motion of an AUV

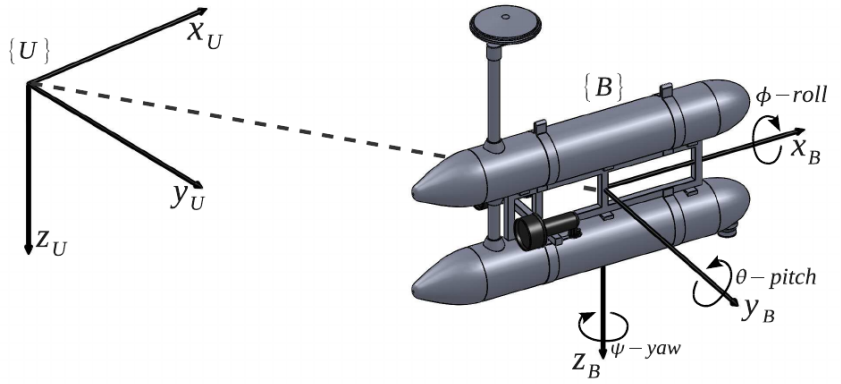


Figure 3.1: Coordinate frames

3.2 Dubin Car

In geometry, the term Dubins 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 traveling the path can only travel forward [10].

A dubin's car is a simple model for a vehicle that transveses a dubin's path.

A simple kinematic car model for the systems is:

$$\begin{aligned} \dot{x} &= u\cos\psi \\ \dot{y} &= u\sin\psi \\ \dot{\psi} &= \omega \end{aligned} \tag{3.2}$$

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

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.

$$\begin{aligned} \dot{\omega} &= r \\ \dot{u} &= a \end{aligned} \tag{3.3}$$

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

$$\begin{aligned} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix} \cdot \begin{bmatrix} u \\ 0 \end{bmatrix} \Leftrightarrow \\ \begin{bmatrix} u \\ 0 \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix}^{-1} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \Leftrightarrow \\ \begin{bmatrix} u \\ 0 \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & \sin(\psi) \\ -\sin(\psi) & \cos(\psi) \end{bmatrix} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \end{aligned} \tag{3.4}$$

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

$$\psi = \arctan \frac{\dot{y}}{\dot{x}} \tag{3.5}$$

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

$$\tau_1 = \dot{u} \tag{3.6}$$

$$\tau_2 = \dot{\omega} \tag{3.7}$$

which can be used as inputs to the system.

3.3 Medusa

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

$$\dot{\eta} = J(\eta)\nu \tag{3.8}$$

with

$$J(\eta) = \begin{bmatrix} {}^{U}_{B}R \end{bmatrix} \tag{3.9}$$

where

$$ {}^{U}_{B}R(\Theta) = \begin{bmatrix} c\psi c\theta & c\psi s\theta s\phi - s\psi c\phi & c\psi s\theta c\phi + s\psi s\phi \\ s\psi c\theta & s\psi s\theta s\phi + c\psi c\phi & s\psi s\theta c\phi - c\psi s\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix}, s\cdot = \sin(\cdot), c\cdot = \cos(\cdot) \tag{3.10} $$

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

$$ {}^{U}_{B}R(\Theta) = R_{z,\psi}R_{y,\theta}R_{x,\phi} \tag{3.11} $$

and

$$ T_{\Theta}(\Theta) = \begin{bmatrix} 1 & s\phi\, t\theta & c\phi t\theta \\ 0 & c\phi & -s\phi \\ 0 & c\phi/c\theta & c\phi/c\theta \end{bmatrix}, \theta \neq \pm 90 \tag{3.12} $$

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.

3.3.1 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

$$ M_{RB}\dot{\nu} + C_{RB}(\nu)\nu = \tau_{RB} \tag{3.13} $$

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

$$ \tau_{RB} = \tau + \tau_A + \tau_D + \tau_R + \tau_{dist} \tag{3.14} $$

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,

$$ \tau_A = -M_A\dot{\nu} - C_A(\nu)\nu \tag{3.15} $$

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

$$\tau_D = -D(\nu)\nu \tag{3.16}$$

where $D(\nu)$ denotes the hydrodynamic damping matrix (positive definite).

4. τ_R - Restoring forces and torques due to gravity and fluid density,

$$\tau_R = -g(\eta) \tag{3.17}$$

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

Replacing (3.14) on (3.13), taking into account (3.15), (3.16), the dynamic equations can be written as

$$\underbrace{M_{RB}\dot{\nu} + C_{RB}(\nu)\nu}_{\text{rigid-body terms}} + \underbrace{M_A(\dot{\nu}) + C_A(\nu)\nu + D(\nu)\nu}_{\text{hydrodynamic terms}} + \underbrace{g(\nu)}_{\text{restoring terms}} = \tau + \tau_{dist} \tag{3.18}$$

which can be simplified to

$$M\dot{\nu} + C(\nu)\nu + D(\nu)\nu + g(\nu) = \tau + \tau_{dist} \tag{3.19}$$

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

3.3.2 Simplified Equations of Motion

This thesis will focus on movement along a 2-D plane, therefor, 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

$$\begin{aligned} \dot{x} &= u\cos\psi - v\sin\psi, \\ \dot{y} &= u\sin\psi + v\cos\psi, \\ \dot{\psi} &= r. \end{aligned} \tag{3.20}$$

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

$$\begin{aligned} \tau_u &= F_{sb} + F_{ps}, \\ \tau_r &= l(F_{ps} - F_{sb}) \end{aligned} \tag{3.21}$$

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

$$\begin{aligned} m_u\dot{u} - m_v vr + d_u u &= \tau_u, \\ m_v\dot{v} + m_u ur + d_v v &= 0, \\ m_r\dot{r} - m_{ub} uv + d_r r &= \tau_r, \end{aligned} \tag{3.22}$$

where

$$\begin{aligned} m_u &= m - X_{\dot{u}} \quad d_u = -X_u - X_{|u|u}|u| \\ m_v &= m - Y_{\dot{v}} \quad d_v = -Y_v - Y_{|v|v}|v| \\ m_r &= I_z - N_{\dot{r}} \quad d_r = -N_r - N_{|r|r}|r| \\ m_{uv} &= m_u - m_v \end{aligned} \tag{3.23}$$

All the previous equations were expressed without considering the influence of external factors like ocean currents. If a constrant irrotational ocean current, v_c, is introduces, 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 (3.22) become

$$\begin{aligned} m_u\dot{u} - m_v(v_r + v_c)r + d_u u &= \tau_u, \\ m_v\dot{v} + m_u(u_r + u_c)r + d_v v &= 0, \\ m_r\dot{r} - m_{ub}(u_r + u_c)(v_r + v_c) + d_r r &= \tau_r, \end{aligned} \tag{3.24}$$

where the expressions for masses and drags remain the same as in (3.23).

3.3.3 *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 rearanging the kinematics:

$$\begin{aligned} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} \Leftrightarrow \\ \begin{bmatrix} u \\ v \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix}^{-1} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \Leftrightarrow \\ \begin{bmatrix} u \\ v \end{bmatrix} &= \begin{bmatrix} \cos(\psi) & \sin(\psi) \\ -\sin(\psi) & \cos(\psi) \end{bmatrix} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \end{aligned} \tag{3.25}$$

and the inputs from the dynamics:

$$\begin{aligned} \tau_u &= m_u\dot{u} - m_v vr + d_u u, \\ \tau_r &= m_r\dot{r} - m_{ub} uv + d_r r, \end{aligned} \tag{3.26}$$

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

$$m_v\dot{v} + m_u ur + d_v v = 0 \tag{3.27}$$

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.

4

Implementation

Contents

In this chapter, implementation in both Matlab and Python will be discussed. This chapter focuses on formalising the optimisation problem for multipe vehicles. Once the optimisation problem is formalised, it can be solved by a whole range of non linear optimisation algorithms.

The Motion Planning problem for multiple vehicles that will be focused for the project consists in a go-to formation manoeuvre [12]. The go-to formation manoeuvre consists of 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.

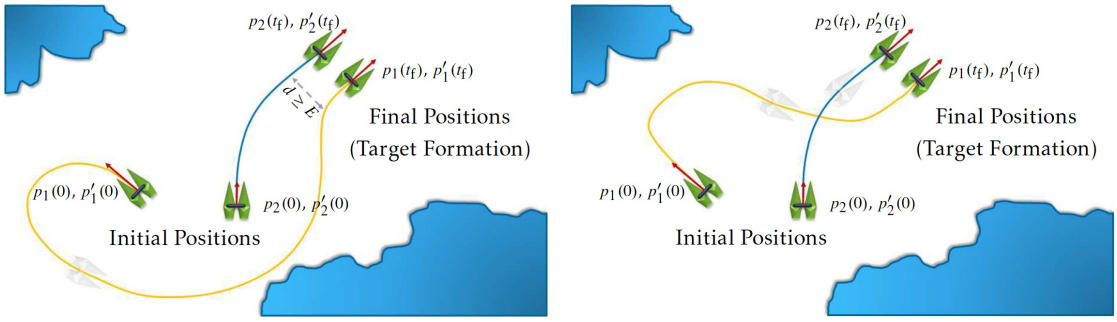
The optimal control problem can be redefined for multiple vehicles as

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

This problem will also produce a solution within time horizon 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.

There are 2 ways of preventing inter-vehicle collision; *spatial deconfliction* and *temporal deconfliction* [13]. Spatial deconfliction 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 requires that two vehicles will never be ”at the same place at the same time”. However, their spatial



(a) Spacial Deconfliction **(b)** Temporal Deconfliction

Figure 4.1: Inter Vehicle deconfliction Solutions

paths are allowed to intersect. Figure 4.1 illustrates the two types of deconfliction strategies. Temporal deconfliction allows an extra degree of freedom and will intuitively lead to cheaper dynamic costs.

A simple method to assure temporal deconfliction of 2 vehicles is to assure that the norm of the distance of each point in time $t \in [0, T]$ is on each point in time subtract the n-dimensional trajectories of each pair of vehicle

4.1 Description of the implemented code

The variables for the motion planning problem are each vehicle's state variables and inputs, as explained in chapter . Each of these variables will be refered to as curves. Optimisation algorithms cannot take continuous functions as variables, therefor, some form of parameterisation of each curve is necessary, as exemplified in some of the algorithms of chapter . Here, each curve will be represetend as a Bernstein Polynomial with order N, which will require $N + 1$ control points. A destinction is made between state variables and inputs: state variables must have establised initial and final conditions, inputs do not. This implies that for state variables, the initial and final control points must be fixed, therefor, 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 perfom operations on the curves

$$\begin{bmatrix} x_0^0 & y_0^0 & \psi_0^0 & u_0^0 & v_0^0 & r_0^0 & \tau_{u_0}^0 & \tau_{r_0}^0 \\ x_1^0 & y_1^0 & \psi_1^0 & u_1^0 & v_1^0 & r_1^0 & \tau_{u_1}^0 & \tau_{r_1}^0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_N^0 & y_N^0 & \psi_N^0 & u_N^0 & v_N^0 & r_N^0 & \tau_{u_N}^0 & \tau_{r_N}^0 \end{bmatrix} \tag{4.2}$$

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, therefor, a function called `matrify()` is necessary in order to transform the flattened optimisation variable to the matrix of equation (4.2).

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 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.

reference "the paper" (of Venanzion, Isaac, Pascoal, etc)

A more precise way to calculate the minimum distance of a Bezier curve to a point or to another curve, compared to the one implemented on section **??** is via the algorithm presented in [9]. This algorithm takes into account the convex hull property of Bezier Curves and the *deCasteljau* algorithm for subdividing curves. It will also employ the GJK algorithm, a fast and efficient way of calculating distances between 2 convex shapes[5].

The fast calculation of distances between these curves allows Bezeir Curves to be appropriate for testing non-linear constraints in multiple vehicle motion planning optimisation.

This fancy algorithm performed poorly because it was iterative. A simpler way to calculate the minimum distance was necessary. This consited in subtracting each pair of 2-D curves for each vehicle to eachother, then getting a rough guess of the closest point of that resulting curve to 0: by performing degree elevation to an order 10 times the original then looking for the closest point to the origin.

Minimum distance of the curves to objects was done with GJK. but it was slow

Collision to circles is calculated by subtracting the 2-D curve tto the centre of the circle, performing degree elevation of the resultign curve, calculating the closest to the origin and checking wether that distance is greater to the radius. Performing the check for every control point was also done but wasn't advantageous (slower runtime), the reason could be because the derivative of each of these distances or the elevated curve with respect to the position of the control points depends on more than 1 control points of the non elevated curve, therefor, some computation is redundant.

The optimisation problem is formulated by constructing a data structure with the fields of table 4.1. In Python the same fields are necessary, however, they must be stored in a python dictonary.

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`
- `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

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

dynamic constraints are respected

The data structure for the nonlinear optimisation problem is then passed to
The running cost function will be based on minimisation of the energy:

$$J = \int tau_u^2 + tau_r^2 \tag{4.3}$$

4.1.1 dynamics

the dynamics in matlab is

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);
];
```

- diffmat preserves the order
- the equailty is maintained in the control points, not the values of the curve itself so some error is expected

diff ma

explain the ordinary differential equaitons

calculation of the limits of accel-eration of the medusa model

5

Results

Contents

Sequential Quadratic Programming is the first constraint optimisation algorithm of choice for soliving these particular nonlinear problems.

5.1 No obstacles

Single vehicle, no obstacles, right turn

5.2 Obstalces

5.3 Mulitple Vehicles

5.4 Variation of cost with order

point out how no limits on the variables makes the problem harder to solve

an example with a square obstacle

an example with a circle obstacle

examples with few vehicles

examples with many vehicles

6

Conclusion

Contents

With this initial study, problems associated with cooperative motion planning was studied with the objective of developing and testing selected algorithms on a number of prototype vehicles that are property of ISR-Técnico. A suite of programs has been developed that perform calculations on Bezier Curves which will prove useful for the later stages of the project. Also, an introduction with Matlab's optimisation tools has allowed an important understanding of their limitations and benefits. Basic concepts like trajectories, paths and linear and non-linear constraints for optimisation problems have been researched.

This initial study has provided me with a clear understanding of how to carry out a research project. This includes how to define a scientific problem, investigating its importance and relevance within the wider scientific area, how to investigate the state of the art and how to develop the research over a limited time-scale.

This project was cool.

Thomas fazer conclusao

6.1 Conclusions

6.2 System Limitations and Future Work

Aliquam aliquet, est a ullamcorper condimentum, tellus nulla fringilla elit, a iaculis nulla turpis sed wisi. Fusce volutpat. Etiam sodales ante id nunc. Proin ornare dignissim lacus. Nunc porttitor nunc a sem. Sed sollicitudin velit eu magna. Aliquam erat volutpat. Vivamus ornare est non wisi. Proin vel quam. Vivamus egestas. Nunc tempor diam vehicula mauris. Nullam sapien eros, facilisis vel, eleifend non, auctor dapibus, pede.

Bibliography

[1] E. Xargay, V. Dobrokhodov, I. Kaminer, A. M. Pascoal, N. Hovakimyan, and C. Cao, "Time-critical cooperative control of multiple autonomous vehicles: Robust distributed strategies for path-following control and time-coordination over dynamic communications networks," IEEE Control Systems Magazine, vol. 32, no. 5, pp. 49–73, 2012.

[2] J. Kalwa, A. Pascoal, P. Ridao, A. Birk, M. Eichhorn, L. Brignone, M. Caccia, J. Alves, and R. Santos, "The european r&d-project morph: Marine robotic systems of self-organizing, logically linked physical nodes," IFAC Proceedings Volumes, vol. 45, no. 27, pp. 226–231, 2012.

[3] A. Aguiary, J. Almeida, M. Bayat, B. Cardeira, R. Cunha, A. Hausler, P. Maurya, A. Oliveira, A. Pascoal, A. Pereira, et al., "Cooperative autonomous marine vehicle motion control in the scope of the eu grex project: Theory and practice," in Oceans 2009-Europe, IEEE, 2009, pp. 1–10.

[4] 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.

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

[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] 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.

[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] 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.

[10] 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.

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

[12] 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.

[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.

A

Code of Project

Nulla dui purus, eleifend vel, consequat non, dictum porta, nulla. Duis ante mi, laoreet ut, commodo eleifend, cursus nec, lorem. Aenean eu est. Etiam imperdiet turpis. Praesent nec augue. Curabitur ligula quam, rutrum id, tempor sed, consequat ac, dui. Vestibulum accumsan eros nec magna. Vestibulum vitae dui. Vestibulum nec ligula et lorem consequat ullamcorper.