SAPIENZA
UNIVERSITÀ DI ROMA

Autonomous and Mobile Robotics

Academic Year 2020/2021

Discrete-time Control Barrier Functions for safe vehicle control

Master's Degree in Artificial Intelligence and Robotics

Rossella Bonuomo, 1923211

Marco Pennese, 1749223

Veronica Vulcano, 1760405

Contents

1 Introduction

Every engineered system should be designed to be safe. Intuitively, safety means that "bad" things are avoided. Therefore, safety-critical optimal control is one of the most important problems in robotics. In this work, we focus on the field of autonomous systems. In particular, we unify Nonlinear Model Predictive Control with Control Barrier Functions that generate safety constraints for obstacle avoidance. We are going to study the case of the unicycle by proposing different scenarios (e.g., fixed or moving obstacles, different paths, single or multiple cars). We are going to make a comparison between discrete time and continuous time, and finally we are going to provide a dynamic extension; by introducing new control inputs, we are going to increase the relative degree of the system.

Code and simulation results are available here: `https://github.com/MarcoPenne/AMR_MPC-CBF.git`.

2 Tools

2.1 Model Predictive Control

Model Predictive Control includes a wide range of control methods used to achieve optimal performance while satisfying a set of constraints. The main characteristic of MPC is that there is an explicit use of a model to predict the process output on a finite prediction horizon (e.g., future time instants). This is done by finding a sequence of optimal control actions as a result of an optimization problem that takes into account some constraints.

The success of MPC is due to the fact that it can be used to control a great variety of processes. In fact, MIMO systems are easily dealt with it and tuning is simpler than PID controllers. Moreover, constraints are systematically included during the process design. This implies that we are able to avoid constraints' violation while computing the optimal control action, so we can provide early warning of potential problems. However, a significant drawback is that we need an accurate model of the process to predict the output.

Formally, the procedure of the MPC can be illustrated as follows. At each time instant t, we consider a prediction horizon N that goes from t to $t+N$ and we predict the output of the process by using the process model. The output depends on the current state and output, and on the present and future control actions. To compute the control actions, it solves an optimization problem considering a cost function; the actions should be chosen in order to satisfy some constraints along the predicted trajectory. In fact, usually, we want that the process model follows as close as possible a predetermined trajectory that is our reference signal without violating some constraints that can be linked, for instance, to safety reasons or to mechanical limitations. Finally, we apply just the first computed control action $u(t|t)$ while the other predicted control actions are discarded (receding horizon strategy). In this way, we obtain the state and the output at the next unit of time $t+1$. Then, it just repeats the process; in fact, usually $u(t+1|t+1) \neq u(t+1|t)$.

A generic MPC finite-time optimal control discrete formulation is described as follows:

$$\begin{aligned} J_t^*(x_t) = \min_{u_{t:t+N-1|t}} & \; p(x_{t+k|t}) + \sum_{k=0}^{N-1} q(x_{t+k|t}, u_{t+k|t}) \\ & s.t. \\ x_{t+k+1|t} &= f(x_{t+k|t}, u_{t+k|t}), k = 0, .., N-1 \\ x_{t+k|t} &\in X, u_{t+k|t} \in U, k = 0, .., N-1 \\ x_{t|t} &= x_t \end{aligned} \tag{1}$$

where $p(x_{t+N|t})$ is the terminal cost, $q(x_{t+k|t}, u_{t+k|t})$ is the stage cost to consider at each step, then there is the system discrete dynamics, the input and output constraints and finally the starting condition.

The optimal solution at time t is represented by a sequence of control actions $u^*_{t:t+N-1|t} = [u^*_{t|t}, .., u^*_{t+N-1|t}]$. Anyway, we just apply the first one $u(t) = u^*_{t|t}(x_t)$.

In this project, the system dynamics is nonlinear, therefore we are going to refer to it as **Nonlinear Model Predictive Control**. As for MPC, NMPC requires to find the optimal solution on a finite prediction horizon. Anyway, the problem to solve is nonlinear and non-convex, therefore finding the global optimum is more difficult and convergence is harder to ensure. Moreover, the algorithms are time consuming with respect to the linear case.

2.2 Control Barrier Functions

Control Barrier Functions are a method to express constraints which intrinsically enforce the system to be safe. In general, safety is defined in a way that the system should be in a certain set. In the CBF context, this set is defined by a function $h(x)$ which should be greater or equal than 0.

The CBF constraints can be used directly in the optimization problem and allow the system to avoid obstacles even when the reachable set along the horizon is far away from the obstacles. In fact, we bind the variation rate (derivative) of the control barrier function h, so that the system does not approach the obstacle too quickly.

Given a nonlinear control system $\dot{x} = f(x) + g(x)u$, with $x \in \mathbb{R}^n$ and $u \in U \subset \mathbb{R}^m$, we consider a set $\mathcal{C}$ defined as a superlevel set of a continuously differentiable function h:

$$\mathcal{C} = \{x \in \mathcal{X} \subset \mathbb{R}^n : h(x) \geq 0\}$$

This set $\mathcal{C}$ is called safe set. Then, the function h becomes a control barrier function if there exists an extended class $\mathcal{K}_\infty$ function α such that for the control system h satisfies:

$$\sup_{u \in U} [L_f h(x) + L_g h(x) u] \geq \alpha(h(x)), \qquad \forall x \in D$$

where $L_f h(x) + L_g h(x) u = \dot{h}(x)$.

One important property of Control Barrier Functions is that if the system is not safe, then it will converge asymptotically to the safe set $\mathcal{C}$ thanks to the CBF. However, this property is not so relevant in the case of obstacle avoidance, because it would mean that the system is into the obstacle.

We can extend this to the discrete-time domain:

$$\Delta h(x_k, u_k) \geq -\gamma h(x_k), \quad 0 < \gamma \leq 1$$

Where $\Delta h(x_k, u_k) := h(x_{k+1}) - h(x_k)$. The lower is γ, the more constrained the safe set will be. In fact, it influences the way in which the system is going to overcome the obstacle; as γ decreases, the system starts to avoid the obstacle earlier if we consider a fixed prediction horizon. In this work, we are going to add CBF constraints to the NMPC problems; in this way we can benefit from both.

2.2.1 Exponential Control Barrier function

So far, the safety-critical constraints have been assumed to be of relative-degree 1, which means that the first-time derivative of the CBF has to depend on the control input. This is clearly a restrictive assumption which is not always held for safety constraints of robotic systems. In order to deal with high relative-degree safety constraints, we need to use Exponential Control Barrier Functions.

We can relax the assumption that the relative degree is 1, and we can consider systems with relative degree $r \geq 1$. In this case, the r^{th} time derivative of $h(x)$ can be written as:

$$h^{(r)} = L_f^r \, h(x) + L_g L_f^{r-1} \, h(x) u \tag{2}$$

so that the input appears. We also assume that $L_g L_f^{r-1} h(x) \neq 0$, while $L_g L_f h(x) = L_g L_f^2 h(x) = L_g L_f^{r-2} h(x) = 0$.

Then, we define:

$$\eta_b = \begin{bmatrix} h(x) \\ \dot{h}(x) \\ \ddot{h}(x) \\ \vdots \\ h^{r-1}(x) \end{bmatrix} = \begin{bmatrix} h(x) \\ L_f h(x) \\ L_f^2 h(x) \\ \vdots \\ L_f^{r-1} h(x) \end{bmatrix} \tag{3}$$

and we assume that we can write:

$$L_f^r h(x) + L_g L_f^{r-1} h(x) u = \mu \tag{4}$$

where $\mu \in U_\mu$. Since $h(x)$ has relative degree as r, then μ is a scalar. Therefore, the dynamic of $h(x)$ can be written as:

$$\begin{aligned} \dot{\eta_b}(x) &= F\eta_b + G\mu \\ h(x) &= C\eta_b(x) \end{aligned} \tag{5}$$

with:

$$F = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \quad G = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix} \tag{6}$$

By choosing a state feedback $\mu = -K_a \eta_b(x)$, then $h(x(t)) = Ce^{(F - GK_a)t} \eta_b(x_0)$.

The function $h : D \to \mathbb{R}$ becomes an Exponential Control Barrier Function if there exists a row vector $K_a \in \mathbb{R}^r$ such that:

$$\sup_{u \in U} [L_f^r h(x) + L_g L_f^{r-1} h(x) u] \geq -K_a \, \eta_b(x) \tag{7}$$

In order to properly design the Exponential CBF, we need to define K_a. This can be done by using pole placement strategies of feedback theory.

We define $K_a = [\alpha_1 \cdots \alpha_r]$; then, the characteristic polynominal of $F - GK_a$ will be $\lambda^r + \alpha_r \lambda^{r-1} + \cdots + \alpha_2 \lambda + \alpha_1 = 0$. In order to guarantee that $\mu \geq -K_a \eta_b(x)$ is an Exponential CBF, we need to chose K_a in such a way that the roots of the characteristic polynominal are real and negative.

3 Model

3.1 Unicycle

The model used in this project is the kinematic model of the unicycle. It is a mobile robot with a single orientable wheel whose configuration is described by $q = \begin{bmatrix} x, & y, & \theta \end{bmatrix}$, where (x, y) are the Cartesian coordinates of the contact point of the wheel with the ground and θ is the orientation of the wheel with respect to the x axis.

$$\begin{cases} \dot{x} = v \cos\theta \\ \dot{y} = v \sin\theta \\ \dot{\theta} = \omega \end{cases} \tag{8}$$

where v and ω are the control inputs. They have a physical interpretation: v is the driving velocity, that is the magnitude (with sign) of the Cartesian velocity vector of the contact point, and ω is the steering velocity.

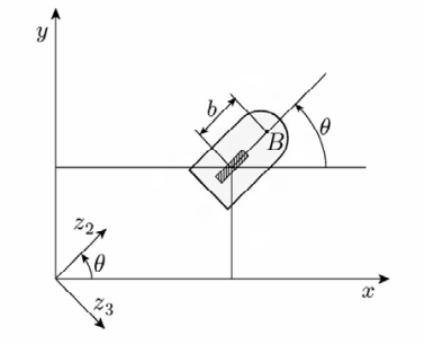


Figure 1: Generalized coordinates

The model is obtained under the assumption that the robot should move without slippage of its wheel.

3.2 Serret-Frenet frames

We are going to consider the model of the unicycle expressed in the Serret-Frenet frame. It describes the kinematic of a point moving along a continuously, differentiable curve. A curve is characterized by a curvature κ; intuitively, it expresses the amount by which a curve deviates from being a straight line. From a mathematical point of view, it is expressed as the inverse of the radius of the circle tangent to the path at a point characterized by a parameter s. In fact, s expresses the arc length that a point has travelled at a given time t. To develop the model of the unicycle in the Serret-Frenet frame, we have to consider some additional quantities that can be represented as follows:

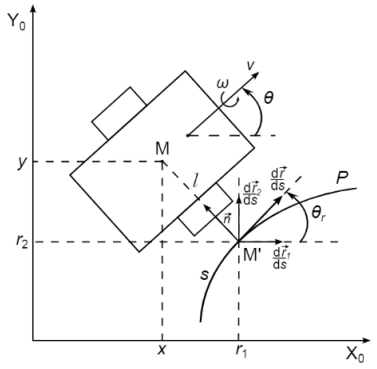


Figure 2: Unicycle in the Serret-Frenet frame

l is the distance between the points M and M', therefore, it is the distance between the considered point and the curve; the curviliar abscissa of M' is exactly s. Finally, we consider the orientation error $\tilde{\theta} = \theta - \theta_r$.

Once we have defined the needed quantities, we can introduce the kinematic model of the unicycle in the Serret-Frenet frame:

$$\begin{cases} \dot{l} = v \sin \tilde{\theta} \\ \dot{s} = \frac{v \cos \theta}{1 \pm \kappa(s) l} \\ \dot{\tilde{\theta}} = \omega \pm \frac{\kappa(s) v \cos \tilde{\theta}}{1 \pm \kappa(s) l} \end{cases} \tag{9}$$

Where the signs + or − depend on whether the robot is moving in the clockwise or counterclockwise direction.

As before, the control inputs are v and ω.

4 Implementation

The simulations have been implemented in Python using the acados solver for Nonlinear MPC. The code is mainly based on four different files:

- *utils*: it contains all the functions used to create the videos and the plots.
- *Path*: it contains the Python class used to define the path.
- *CarModel*: it is the file in which we define the unicycle model by specifing the state variables, the controls, the parameters, the dynamics and the Control Barrier Function expressions.
- *main*: it is the file in which we define and solve the NMPC problem.

4.1 Acados

Acados is a software package for finding the solution of optimal control and estimation problems. It provides some libraries written in C which expose very simple interface to Python, and it is compatible with the language of CasADi. The modelling language of CasADi is based on graphs; this makes the code faster and more suitable for online applications. Moreover, for linear algebra operations, it makes use of BLASFEO that enables a considerable speed up in the computation of matrix operations.

Acados solves a nonlinear Optimal Control Problem (OCP) whose aim is to minimize a cost function given the system dynamics, the initial conditions and the nonlinear path constraints. The OCP is discretized through the multiple shooting approach and the resulting problem is solved by implementing the Sequantial Quadratic Programming (SQP) algorithm. It looks as follow:

$$\begin{aligned} w^{[i+1]} &= w^{[i]} + \Delta w^{[QP]}, i = 0, 1, ... \\ \pi^{[i+1]} &= \pi^{[QP]}, i = 0, 1, ... \\ \mu^{[i+1]} &= \mu^{[QP]}, i = 0, 1, ... \end{aligned} \tag{10}$$

where $\Delta w, \pi, \mu$ are the solutions of a Quadratic Program (QP).

Acados offers also a method to approximate the Hessian matrix used to solve the problem; in all the simulations, we are going to use Gauss-Newton approximation that represents a valid approximation of the exact Hessian.

An important part consists in the integration of the differential equations that represent the system's dynamics. Acados offers different integration schemes; we are going to use Explicit Runge Kutta (ERK) method for the continuous-time simulations and the discrete integration for the discrete-time ones.

There exist different solution strategies for QP interfaced from acados. We mainly focus on *qpOASES* and *HPIPM*.

- qpOASES: it is one of the *active-set* methods, so it is based on the principle that if we are able to know the set of active constraints at the solution, the optimization reduces to solving a set of linear equations. Thus, they propose in each iteration a current guess of the set of active constraints and, when not at the solution, update this guess accordingly. qpOASES updates are based on the fact that successive problems in real time are close to each other. We use it with the condensing approach; we solve a QP problem in which only the initial state and the control inputs are the optimization variables; therefore, we eliminate the dynamic equality constraints.
- HPIPM: it is one of the *interior-point* methods. Violation of inequality constraints are prevented by augmenting the objective function with a barrier term that causes the optimal unconstrained value to be in the feasible space. We use it with the partial condensing approach; we are going to eliminate only some of the state variables leading to a better trade-off between horizon length and number of optimization variables.

Due to the high computational time, most of the times we are going to use **Real-time iteration** (RTI) scheme. It solves an inequality-constrained QP problem in each iteration, so one full iteration of SQP is performed.

4.2 Car Model

Our car has the kinematic model of an unicycle. In order to define it, we have implemented a Python class in which we specify in input the path, the dimensions of the car, the obstacles whose position do not change in time, the number of laps and finally two parameters for building the CBF. Moreover, in the discrete model, we also need to specify the length of the time interval.

At first, we chose the model name and we load the track parameters; this means that we obtain the values for the curvature along all the path, that will be used in defining the kinematic model for the car in Serret-Frenet frames. Then, we define the state and the control variables. Since we are using the Serret-Frenet coordinates, the state variables will be s, l and $\tilde{\theta}$, while the control inputs will be v and ω. We define the parameters that in our case are the coordinates of the moving obstacles, and we express the dynamics both in an implicit and explicit way.

Finally, we specify the Control Barrier Functions for each obstacle, considering both the moving ones and the fixed one. The CBF used to perform obstacle avoidance is:

$$h_t^i = \frac{(s_c - s_{obs}^i)^4}{(2l_1)^4} + \frac{(l_c - l_{obs}^i)^4}{(2l_2)^4} - \alpha \tag{11}$$

where (s_c, l_c) is the position of the controlled car, (s_{obs}, l_{obs}) is the position of the i^{th} obstacle and α is a parameter defining how close the car goes to the obstacles.

While defining the constraint, we have to compute the quantity $\Delta h(x_k, u_k) := h(x_{k+1}) - h(x_k)$, so we need a way to integrate the state, according to the chosen inputs, in order to compute x_{k+1}.
In first instance, we tried to use a simple Euler integrator for performing this task, but it turned out not to be enough, and the simulation stopped with some errors. In fact, when we moved to a more accurate Runge-Kutta integrator, the errors disappeared.

In could happen that, due to the inaccuracies in Euler integration, in some situations the car moves outside the safe-set, and this led to a problem of infeasibility that causes the errors during the simulation. Having moved to a better integrator, the errors have vanished.

4.3 Path

The car has to move along the closed circuit composed by four sides linked by a quarter of circumference.

Concerning its implementation, we have defined a Python class; the path can be created by just specifying in input the vertical and horizontal lengths and the radius of the circumferences. It is a function of the arc length s; in fact, depending on the value of s, we can distinguish between the angular or the straight part. Moreover, when s is equal to the total length of the path, the car has completed one lap.

The Path class has some useful methods; the function *get_k* is used to return the curvature of the path and will be used to define the kinematic model of the car in the Serret-Frenet frame, while the function *get_theta_r* is used when we will have to transform the kinematic model of the unicycle from the Serret-Frenet frame to the Cartesian frame. Finally, the function *get_len* is used to return the total length of the path.

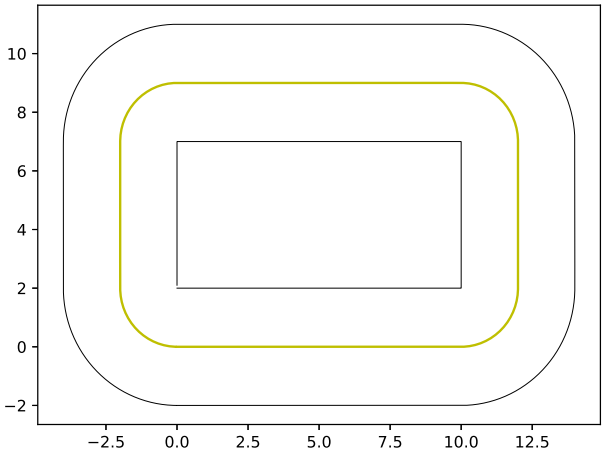


Figure 3: Path

5 Tests and results

We have done experiments considering different scenarios. We are going to illustrate them in the following sections.

5.1 Convergence to trajectory

One of the main issues related to control problems is asymptotic stability; we have to be sure that even when the car is far from the reference trajectory, it is able to approach to it.

In order to analyze the convergence of the system, we give an initial condition to the car that is far from the desired trajectory; if the car is able to reach the trajectory despite the initial position, then we are sure that the system is able to converge.
In particular, we considered four different initial conditions.

The first starting pose we consider is $q = \begin{bmatrix} -0.5, & -1.3, & -1.396 \end{bmatrix}$:

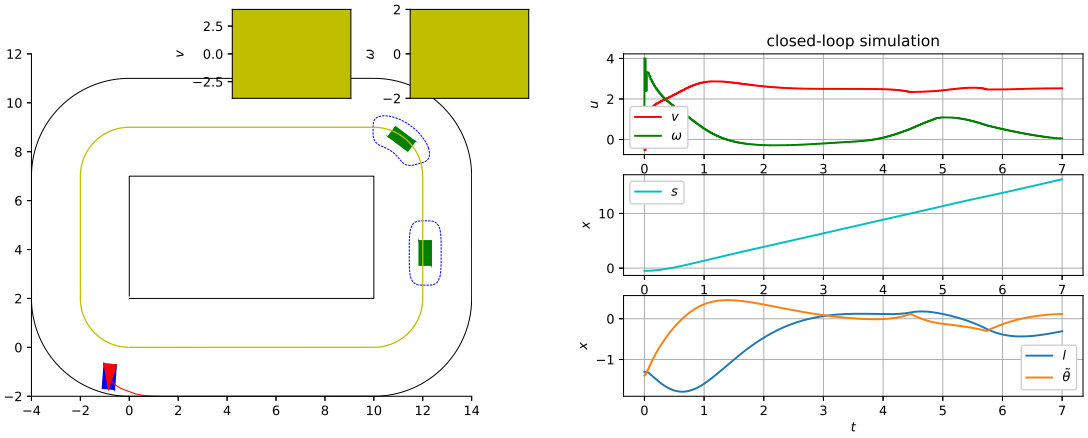


Figure 4: Starting pose of the car - 1

In this case, the car is oriented opposite with respect to the path, so it has a negative orientation error. Therefore, it has to move with a positive steering velocity in order to drive the orientation error to zero.

Concerning the linear velocity, it suddenly reaches the target value and keeps it almost constant. In this case, since there are no negative values of the velocity, the car always moves forward.

Then, we consider $q = \begin{bmatrix} -0.8, & 1, & 1.74532925 \end{bmatrix}$ corresponding to:

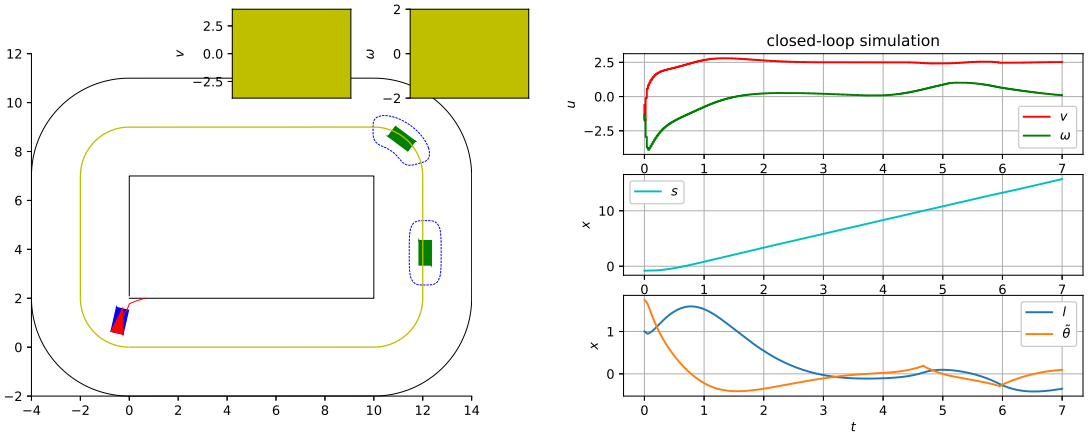


Figure 5: Starting pose of the car - 2

As before, the car always drives forward; anyway, now it has a positive orientation error, so in the first time instants it has a negative steering velocity in order to drive this error to zero.

The next starting configuration is $q = \begin{bmatrix} 0.3, & 0.6, & 2.7925268 \end{bmatrix}$:

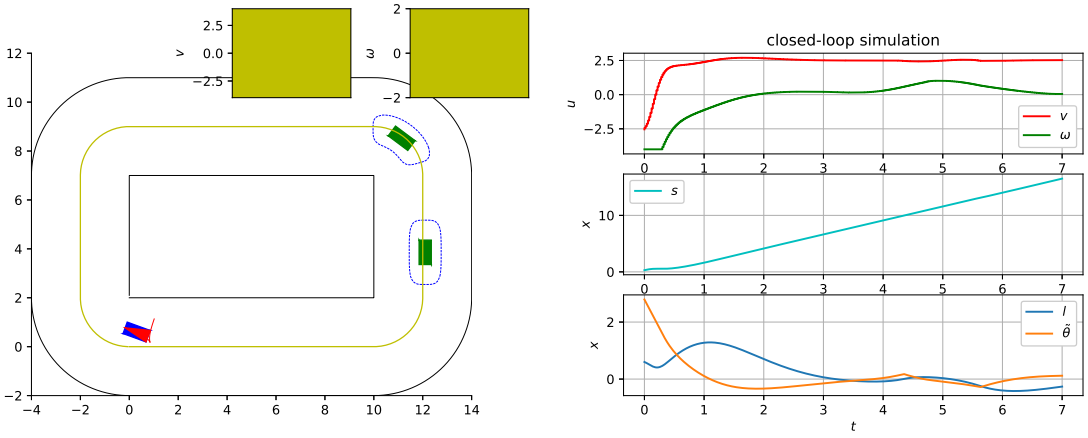


Figure 6: Starting pose of the car - 3

In the velocity plots, we can notice that there is a small time interval in which the driving velocity is negative; this is due to the fact that for just some seconds, the car moves backward. Concerning the steering velocity, as before, it has a negative value in the first time instants so to balance the orientation error.

Finally, we consider $q = \begin{bmatrix} -1, & 0.6, & 3.05432619 \end{bmatrix}$ corresponding to:

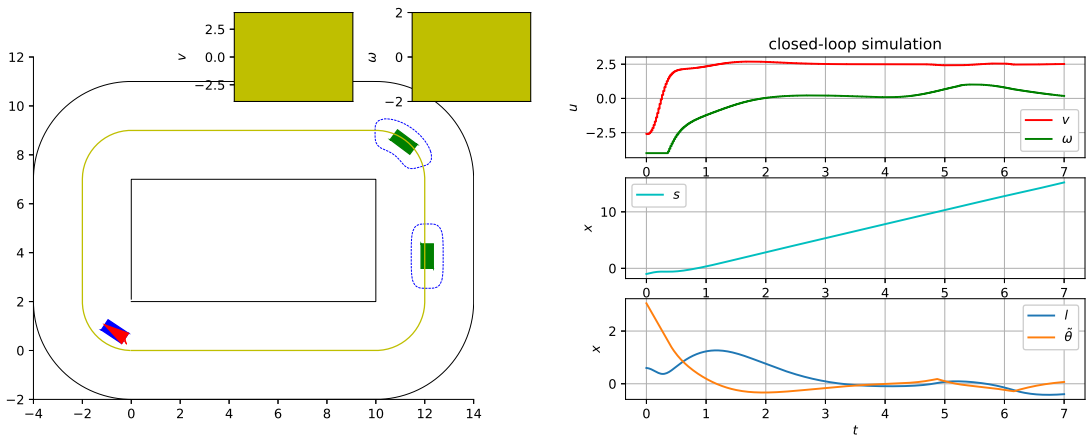


Figure 7: Starting pose of the car - 4

This last case is quite similar to the one seen before for the steering velocity, but we have a bigger interval of time in which the car moves backward.

In all the situations, after some maneuvers, the car is able to approach the path correctly. In fact, as we can see from the plots, at the beginning l has quite a large value, but it tends to zero as time goes by. Concerning s, at the beginning it does not grow linearly due to the maneuvers.

Since the s coordinate always follows an increasing linear trend, whose slope depends on the velocity, from now on we decide not to plot it in the state (except for some significant cases).

5.2 Single car with fixed obstacles

After studying the convergence, we start doing some simulations with a single car moving on a path with fixed obstacles. The goal of the car is to move along the specified path while avoiding obstacles thanks to the CBFs.

There are two fixed obstacles: one on the first straight segment and the other on the second curve. Around the obstacles, we can see a dashed contour which represents the safe set and it depends on the coordinates s and l.

All these experiments are done in discrete time. The car starts from the zero configuration and moves with a target velocity equal to $2.5m/s$. The simulation ran for $20s$.

The QP solver used is FULL_CONDENSING_QPOASES, while the NLP solver is SQP.

5.2.1 Study gamma and prediction horizon

In these experiments, we decide to focus on two main parameters to see how they affect the simulation; the first one is the parameter γ in the CBF, while the other one is the prediction horizon. The first influences the way in which the car is going to approach the obstacle, by making the car deviating in a time depending on the value of γ; the latter describes how much we are able to predict in the future. Obviously, we would like to keep a prediction horizon as long as possible, but this is not possible due to the computational complexity. Therefore, the challenge is to be able to find a compromise between performance and complexity.

We tested three different values of γ: 0.1, 0.5 and 1. In these tests, the prediction horizon is equal to $1s$.

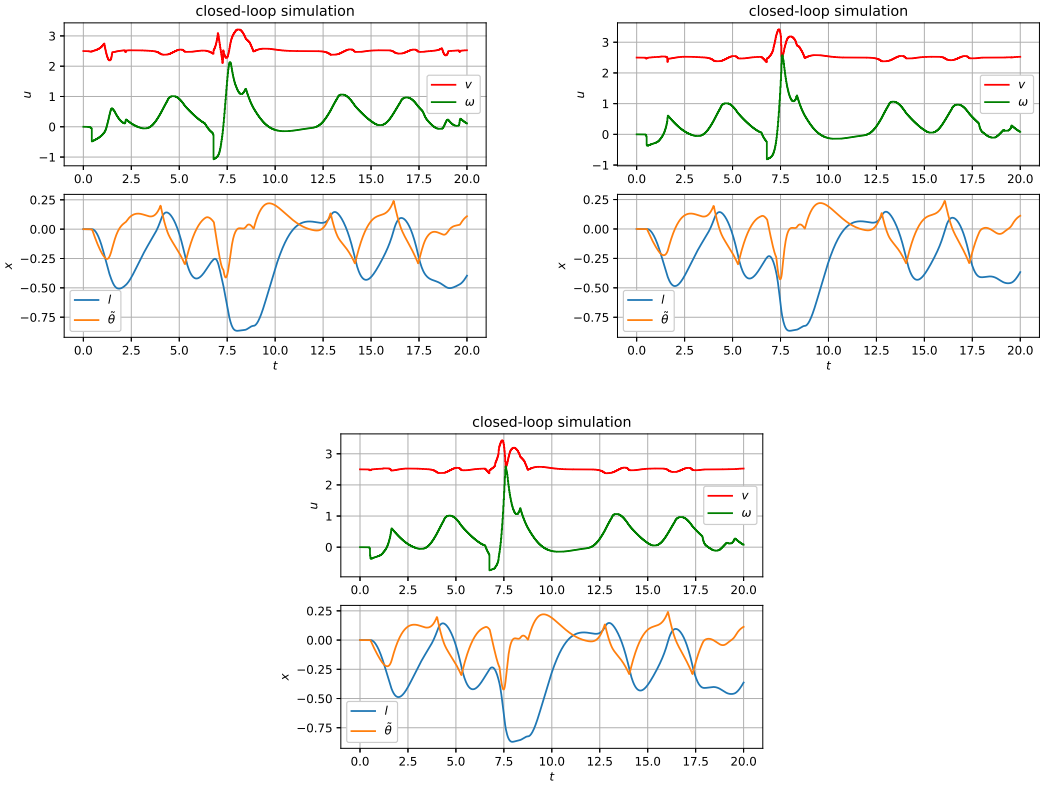


Figure 8: Plots for $\gamma = 0.1$, $\gamma = 0.5$ and $\gamma = 1$, respectively

By looking at these plots, we can see that they only slightly differ. The main difference is on the angular velocity. We can understand it by looking Fig. 9. In fact, the higher is γ, the higher is the angular velocity during the overtake. This is because the larger is γ, the more the car begins to overtake the obstacle when it is very close to it; therefore, the car will have to rotate more to overcome the obstacle.

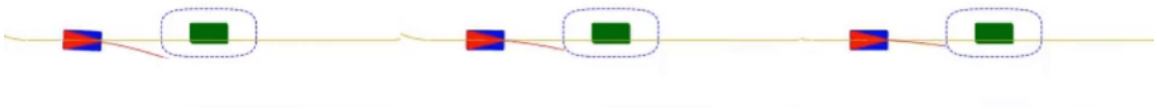


Figure 9: Behaviour for $\gamma = 0.1$, $\gamma = 0.5$ and $\gamma = 1$, respectively. We can see that with lower values of γ, the car starts steering earlier to avoid the obstacle.

It could be interesting to analyze the computational time required to solve the optimization problem while changing the parameters.

T_f	γ	mean time	std dev	max time	cost integral
1s	0.1	0.0296	0.0476	0.7414	134.93
1s	0.5	0.0244	0.0315	0.4348	135.88
1s	1	0.0261	0.0680	1.6073	136.00

We can see in the table that the mean over all the iterations seems to not be affected too much by the changing of γ.

The max time is the time that the slowest iteration takes in order to finish. It can be interpreted as a measure of the most complex situation from which the car has to compute the solution and this can depend from a lot of factors. As we can see, it has high variability among different values of γ and it does not grow as γ grows; in fact, we have the minimum value in correspondence of γ=0.5. In order to

validate this result, we did many experiments and the results we obtained are pretty similar (up to the 2nd or 3rd decimal digit for the max time).

The cost integral is given by the formula $\sum_k u_k^T u_k \Delta T$, so it could be interpreted as a global measure of the control effort. It is very interesting to see that this value seems to be correlated with the value of γ, in fact, the lower the value of γ the lower the control effort.

Then, we wanted to test the influence of the length of the prediction horizon on the performances. We fix γ to 0.5, and let the only discriminant to be the parameter T_f. We tested 4 different values for the prediction horizon: $0.8s$, $1s$, $2s$ and $5s$.

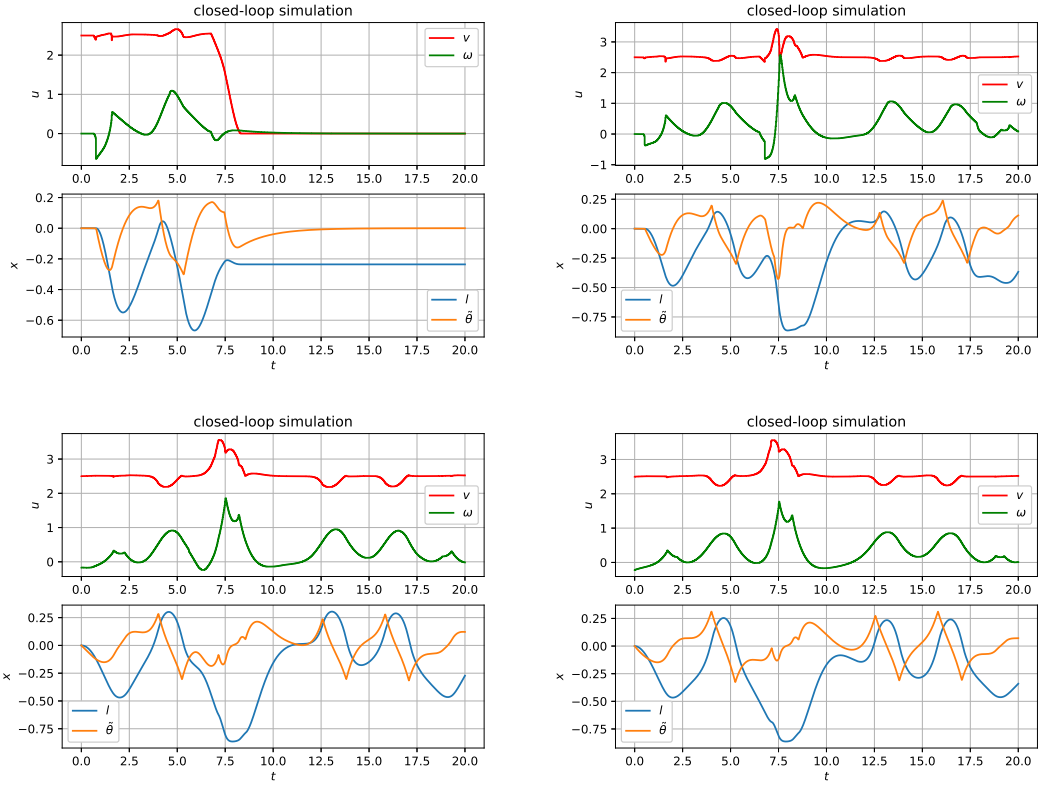


Figure 10: Plots for $T_f = 0.8s$, $T_f = 1s$, $T_f = 2s$, $T_f = 5s$ respectively

As expected, with longer prediction horizons (2s and 5s) the car has a smoother behaviour than with short ones. We can see that when the prediction horizon is too small (e.g. $0.8s$), the car stops in correspondence of the second obstacle and the s coordinate stops to increase (see Fig. 11).
This is because the car is not able to predict the future long enough and doesn't find a way to escape the obstacle in which is stuck. This can be observed also by looking at the inputs plot; suddenly, v and ω start to be zero.

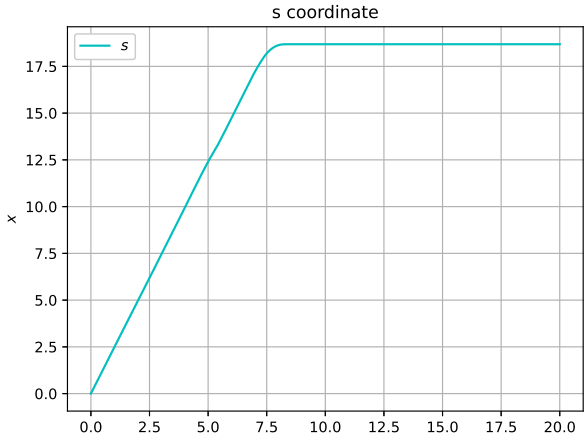


Figure 11: Evolution of s coordinate: $T_f = 0.8s$ and $\gamma = 0.5$

Good results are obtained with larger prediction horizons, but of course the higher is the prediction horizon, the higher is the computational cost.

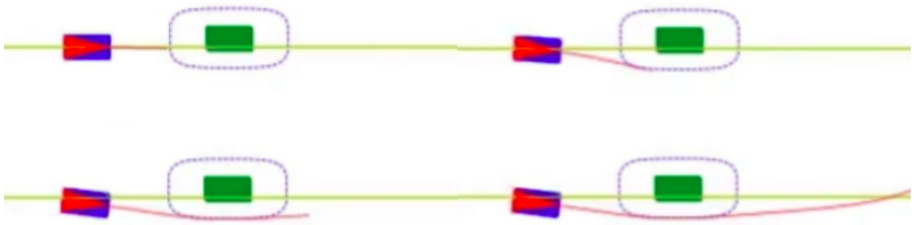


Figure 12: Behaviour for different T_f while avoiding first obstacle. We considered $T_f = 0.8s$, $T_f = 1s$, $T_f = 2s$, $T_f = 5s$ respectively. We can see that, if the prediction horizon is longer, the car is able to predict a better trajectory.

In the table are reported the values to analyze the computational time.

T_f	γ	mean time	std dev	max time	cost integral
0.8s	0.5	0.0128	0.0073	0.0781	×
1s	0.5	0.0244	0.0315	0.4348	135.88
2s	0.5	0.1265	0.1979	1.7619	134.03
5s	0.5	2.2235	2.7018	15.2301	135.26

As we expected, increasing the prediction horizon increases the computational time. From the values in the table we recognize that, increasing T_f, the average iteration time increases more than linearly. So, if we require an upper bound for the computational time (e.g. needed by real-time controllers), we need to be very careful in setting the value of T_f.

We would like to combine what we learned so far by choosing both γ and T_f in a way that we can obtain a desired behaviour, while keeping the computational time small.

First of all, we tried to tune the γ parameter in a way that the car is able to avoid the obstacle also with T_f =0.8s. We found that, with a very low γ (e.g. γ =0.05), the car is perfectly able to run over all the path while avoiding all the obstacles, even if the prediction horizon is limited to T_f =0.8s. This is due to the fact that reducing γ, the car starts to steer much earlier without remaining stuck against the obstacle.

T_f	γ	mean time	std dev	max time	cost integral
0.8s	0.05	0.0244	0.0455	0.6829	124.36

We also did some tests analyzing the computation time of some parameters combinations. The idea is to reduce the prediction horizon in order to have shorter computation time, and to compensate for the loss of optimality that is brought by this, with the reduction of γ. The following combinations led to very similar behaviour of the car, while having very different computation time over all the iterations.

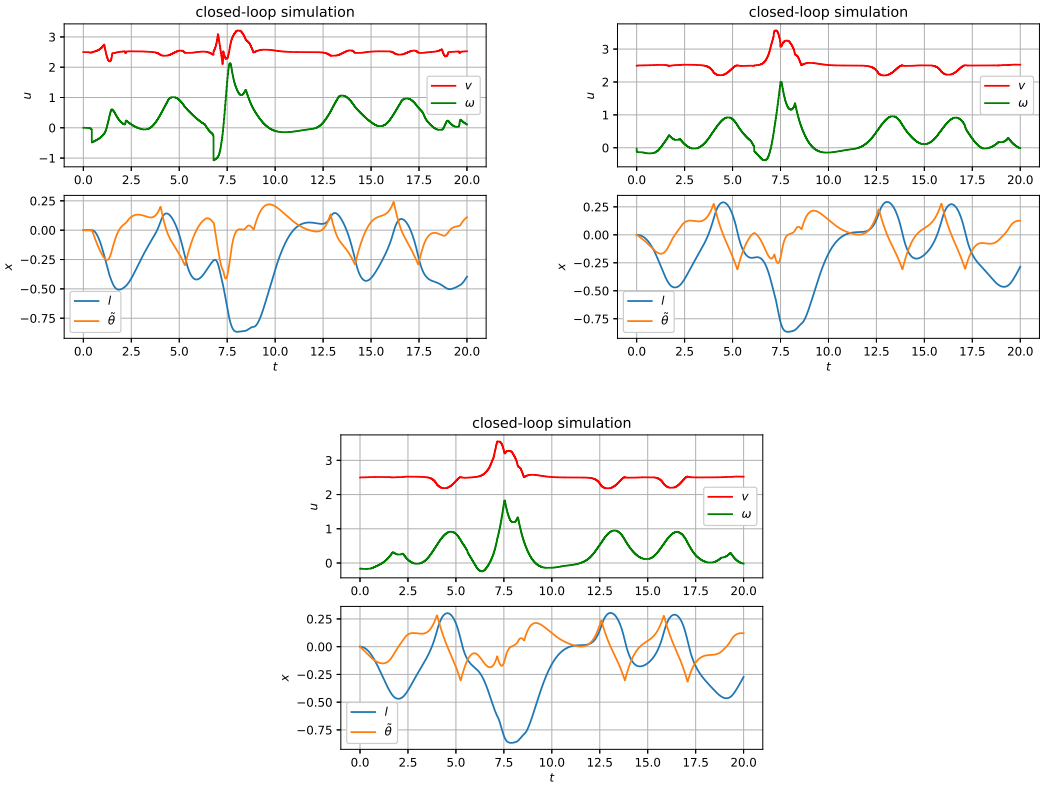


Figure 13: Plots for $T_f = 1s$ and $\gamma = 0.1$, $T_f = 1.5s$ and $\gamma = 0.5$, $T_f = 2s$ and $\gamma = 1$ respectively

T_f	γ	mean time	std dev	max time	cost integral
1s	0.1	0.0296	0.0476	0.7414	134.93
1.5s	0.5	0.0636	0.0998	0.9168	134.06
2s	1	0.1184	0.1778	1.8536	134.05

It is clear that if we have to choose between these three alternatives, the one with $T_f = 1s$ and $\gamma = 0.1$, best fits our requirements.

5.3 Single car with moving obstacles

Now, we are going to provide the same setting as before, but this time the obstacles are not fixed; they are going to move with a constant velocity equal to $1m/s$. In studying the case of a single car with moving obstacles, we will focus on the comparison between discrete time and continuous time.

The controlled car starts from an initial configuration of $x_0 = \begin{bmatrix} 0.5 & -1.2 & -1.39 \end{bmatrix}$ and has a target velocity of $2.5m/s$; since it moves much faster than the obstacles, it is going to overcome them without crashing against them.

We are going to run two different experiments: one with a very small discretization step ($\Delta T_1 = 0.02s$) and the other with a bigger one ($\Delta T_2 = 0.1s$) so to see if eventually there are differences between the

two performances.

The QP solver used is PARTIAL_CONDENSING_HPIPM, while the NLP solver is SQP_RTI. So, in this case, we are speeding up the computation using Real-time Iterations.

5.3.1 Discrete time vs continuous time

We are going to compare the results achieved in discrete time with the same model computed in continuous time.

As we already explained in 2.2, in continuous time, the CBF constraint is different from what we have in discrete time. In fact, we have that the discrete constraint is:

$$\Delta h(x_k, u_k) \geq -\gamma h(x_k), \quad 0 < \gamma \leq 1 \tag{12}$$

while the continuous CBF constraint has to be expressed as:

$$\dot{h}(x_k, u_k) \geq -\gamma(h(x_k)), \gamma \in \mathcal{K}_\infty \tag{13}$$

As a consequence of this, we had to compute the derivative of our CBF with respect to time in order to properly set the safety constraint. According to the $h(x_k)$ function we choose, the derivative is the following:

$$h_t^i = \frac{(s_c - s_{obs}^i)^4}{(2l_1)^4} + \frac{(l_c - l_{obs}^i)^4}{(2l_2)^4} - \alpha \tag{14}$$

$$\dot{h}_t^i = \frac{4(\dot{s}_c - \dot{s}_{obs}^i)(s_c - s_{obs}^i)^3}{(2l_1)^4} + \frac{4(\dot{l}_c - \dot{l}_{obs}^i)(l_c - l_{obs}^i)^3}{(2l_2)^4} \tag{15}$$

The term $\dot{h}_t^i$ simplifies a bit in the case of fixed obstacles, or in the case in which the obstacle follows a path using a constant l_{obs}^i, in fact in that case $\dot{l}_{obs}^i$ will be zero.

Discrete time We are going to consider a prediction horizon of $1s$ and we set γ in the CBF as 0.5. At first, we set the discretization step at $0.02s$ and we got the following result:

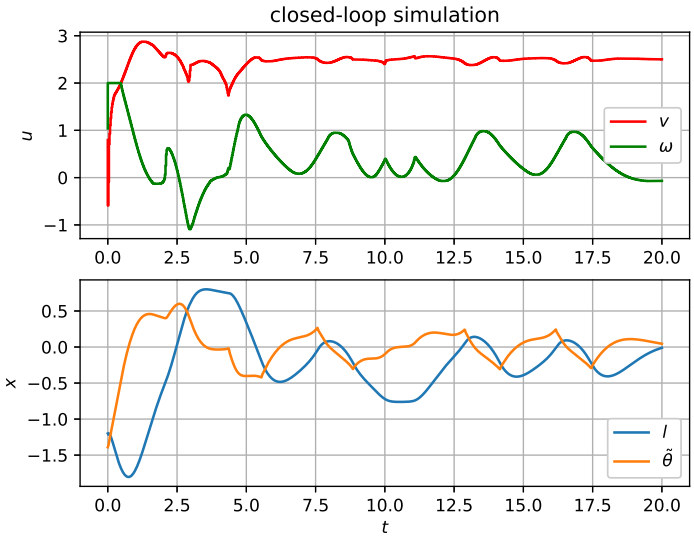


Figure 14: Single car with moving obstacles in discrete time - discretization step of 0.02

Then, we set the step as $0.1s$ and we got:

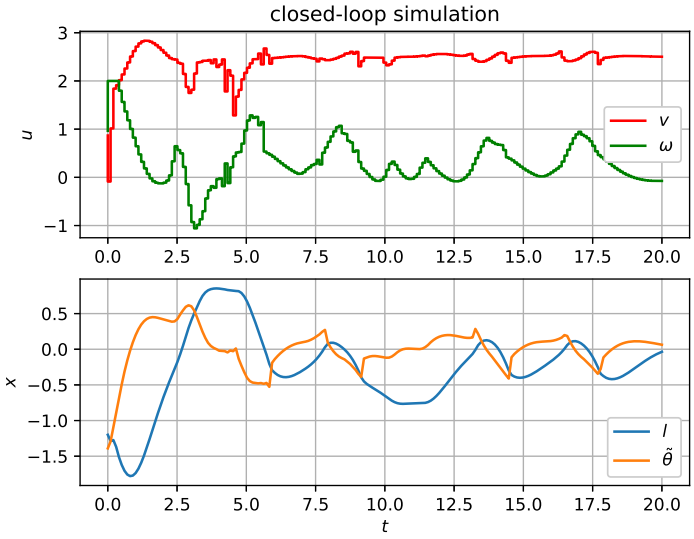


Figure 15: Single car with moving obstacles in discrete time - discretization step of 0.1

There is an evident difference: the second input's plot looks to be piecewise constant more than the first one and it is noisier; this is obviously linked to the different discretization step. However, the two plots follow more or less the same shape, so also in this case, the car is able to track the desired trajectory; this is also evident from the plots of the state variables.

Continuous time In this case, we are still considering a prediction horizon of $1s$ but since we are in continuous time, we set $\gamma(h(x)) = 50h(x)$. At first, we have considered a step of $0.02s$ obtaining the following result:

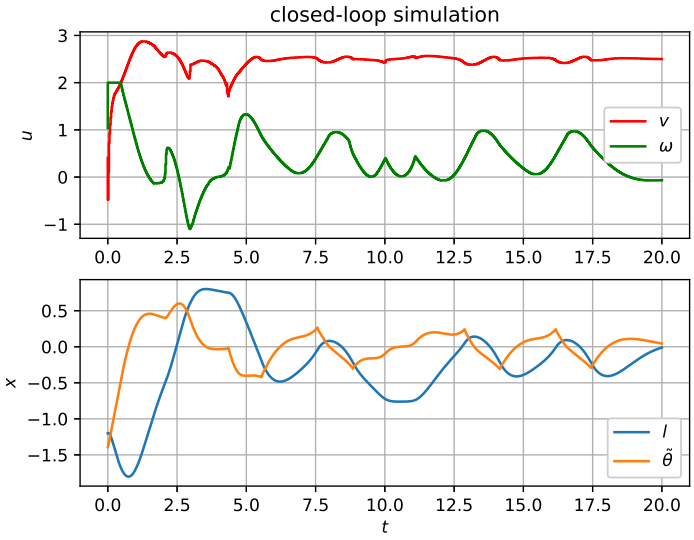


Figure 16: Single car with moving obstacles in continuous time - discretization step of 0.02

Then, we have considered a step of $0.1s$:

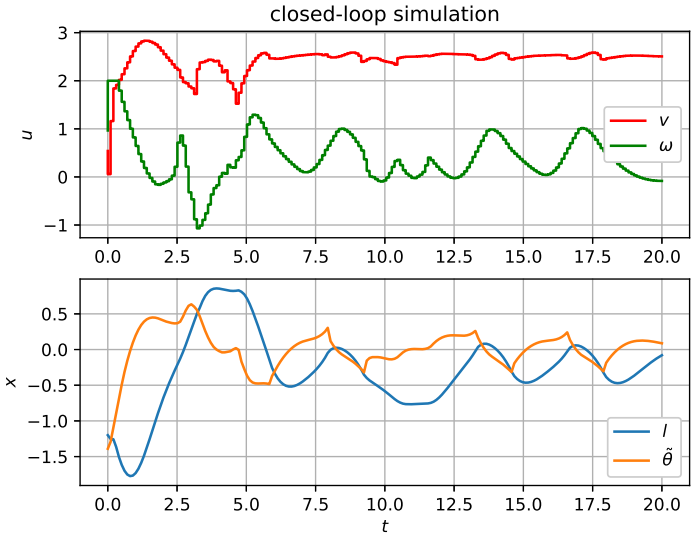


Figure 17: Single car with moving obstacles in continuous time - discretization step of 0.1

Also in this case, the second inputs plot is piecewise constant for the same reason of the discrete time case.

By analyzing the results obtained in these simulations, we get very similar performances in discrete time and in continuous time. However, we would prefer to use the discrete time version so to simplify the problem; in fact, using the discrete formulation, we can avoid computing the time derivative of the Control Barrier Function.

Concerning the performance, we can see that the discrete time works as well as the continuous one, so there is no loss of information. In fact, at first, we could think that the very similar performance was due to the fact that we were using a very small discretization step; however, this is not true because even with a bigger discretization step, the discrete time version is able to work well.

Therefore, by using the discrete time version that is quite simpler than the continuous time one, not only we are able to get very good results, but we can avoid computing time derivatives.

Computational Time Comparison Finally, we want to analyze the computational time both for the discrete time and continuous time. Therefore, we run other two experiments in which the controlled car starts from the zero initial configuration and has a target velocity of $2.5m/s$. The QP solver used is PARTIAL_CONDENSING_QPOASES, while the NLP solver is SQP, so we do not use Real Time Iterations. The prediction horizon is set equal to $1s$.

In continuous time, we set $\gamma(h(x)) = 1.5h(x)$; then, in discrete time we choose $\gamma = 0.03$ so as to obtain performances that are as similar as possible. In fact, if we look at the plots of the closed loop simulation, the results are almost the same:

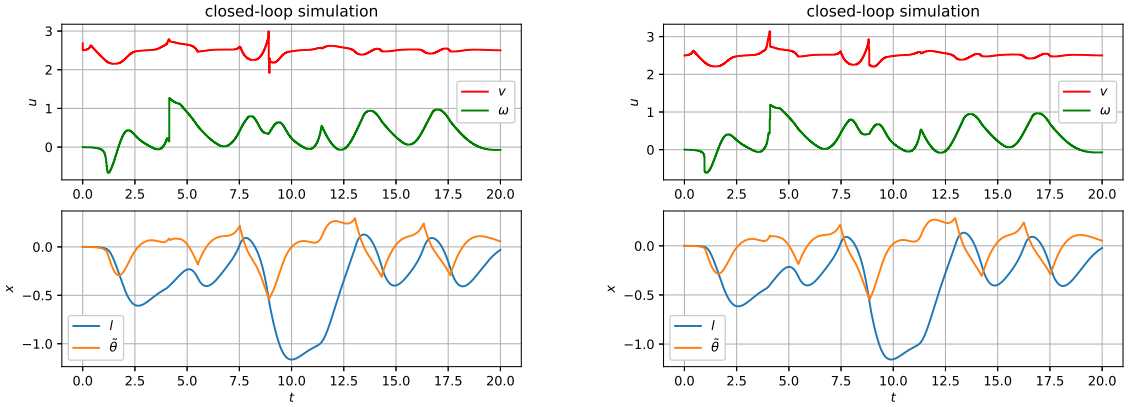


Figure 18: Continuous time (left) vs Discrete time (right)

Then, we report in a table the values concerning the computational time:

	mean time	std dev	max time	cost integral
Discrete Time	0.0389	0.0554	0.7089	129.0932
Continuous Time	0.0391	0.0605	0.7497	128.9603

Also in this case the results obtained are pretty similar. Thus, we can state that there are no significant differences in considering a discrete or a continuous version of the problem.

5.4 Multiple cars

So far, we have considered the case in which a single controlled car moves among fixed/moving obstacles; in order to make things more challenging, we tried to put together multiple controlled cars moving on the same circuit among obstacles.

5.4.1 Centralized vs Independent approach

We have considered two different approaches:

- A centralized approach, in which all the cars are in the same model; therefore, the number of state variables is $n \times m$, where n is the number of generalized coordinates and m is the number of cars. In this case, we have a single optimization problem in which each car is aware of the others' behaviour.
- An independent approach, in which each car has its own model which is independent from the others. In this case, we have as many optimization problems as the number of controlled cars.

Obviously, the centralized approach is just an ideal behaviour since in real applications it is not possible that a car is aware of the behaviour of the others. For this reason, we expect this approach to be better with respect to the second one.

The second one is an approach that we can find also in real applications; the car is aware only of its own movements and has to be careful about other cars' future actions.

For the experiments, we have considered the discrete case in which we have three controlled cars moving along a path full of fixed obstacles. Each controlled car moves with a different velocity: $1m/s$, $2m/s$ and $3m/s$; also the target values for l are different. The value chosen for γ in the CBF is 0.5, and the prediction horizon is equal to $2s$. The QP solver used is FULL_CONDENSING_HPIPM, while the NLP solver is SQP_RTI. We have decided to use RTI because now the setting is complex and the complexity is higher; therefore, in this way, we can speed up the computation.

This is the result of the simulation for the centralized approach:

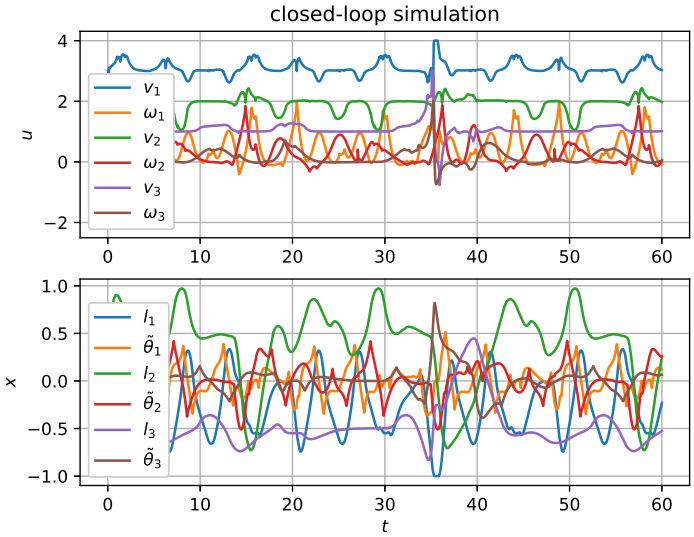


Figure 19: Multiple cars: centralized approach

While this is the result of the simulation for the independent approach:

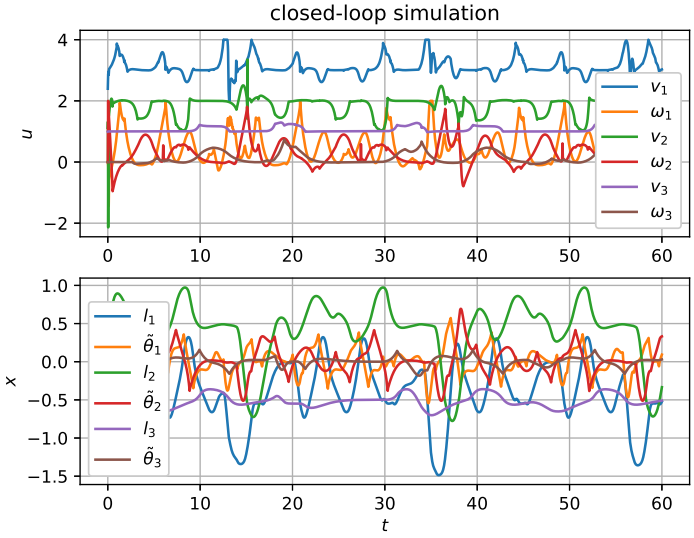


Figure 20: Multiple cars: independent approach

Differences appear when the cars approach each other. In particular, the state variable l_1 has different peaks between the two plots. The reason can be understood by looking at the Fig. 21

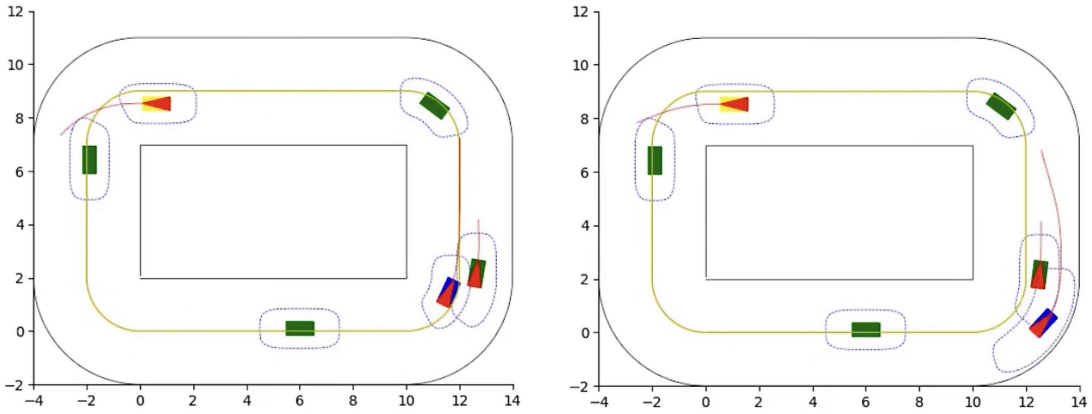


Figure 21: Overtaking: on the left we have the centralized approach, while on the right we have the independent approach

If we look at this figure, on the left we have the centralized case, while on the right the independent one. We can see that in the former the blue car overtakes the green one on the left, while in the latter the blue car overtakes the green one on the right. This implies that in the latter case, the blue car has to travel a longer distance.

Similar behaviours can be observed throughout all the simulation; for this reason, if we look at the final frame, the cars in the centralized approach are moved a bit forward with respect to the ones in the independent approach:

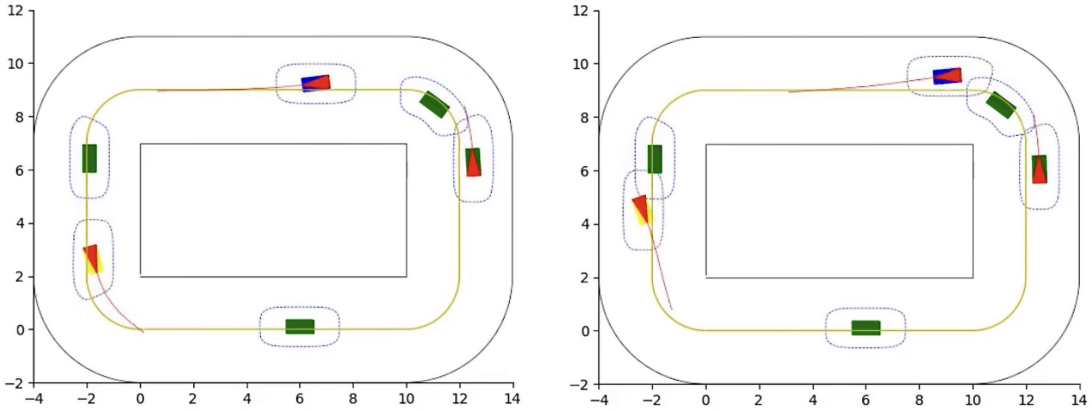


Figure 22: Final configuration after 60 sec of simulation: on the left we have the centralized approach, while on the right we have the independent approach. We can see that using the centralized approach the cars managed to travel for a longer run in the same amount of time. This could mean that the centralized approach was able to find a more optimized trajectory, because when the car overtakes an obstacle the distance traveled is less, therefore "useless" maneuvers are avoided.

5.5 Extension

After proving that this procedure works also considering multiple cars inside the same path, we tried to push our work even further. In fact, we wanted to recreate a real world situation in which cars don't follow an isolated path, but often intersect with other cars. This happens, for instance, in a real world intersection, where the cars have to negotiate right of way.

Also in this case, we considered the two different approaches: the centralized approach and the independent one.

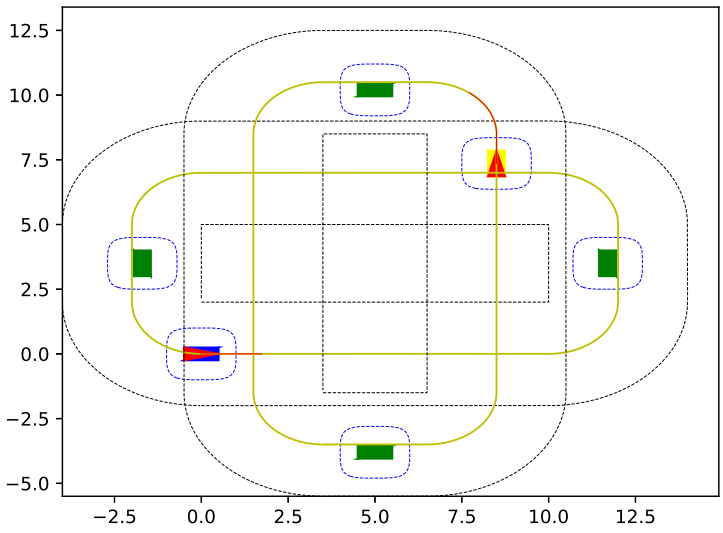


Figure 23: Two paths intersecting in 4 points. In the experiments we considered both moving obstacles and controlled cars.

5.5.1 Multiple car with intersecting trajectories: centralized vs independent approach

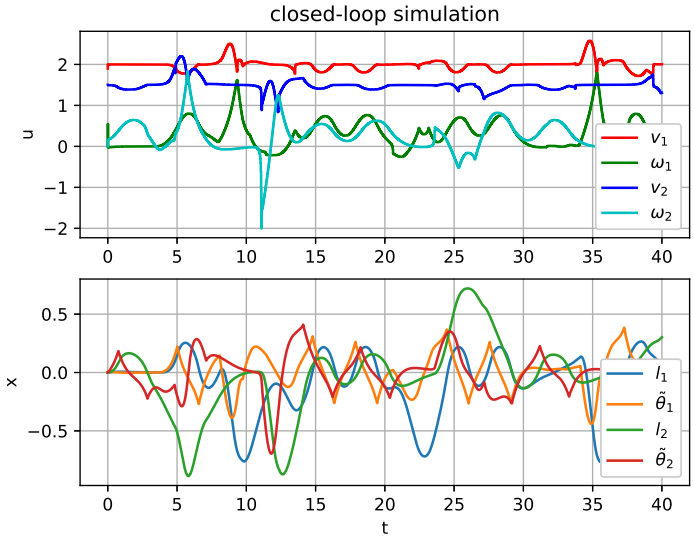
Now, we analyze the results we obtained applying the two different approaches.

For the experiments, we have considered the discrete case in which the obstacles are moving with a constant velocity equal to $0.5m/s$. Each controlled car moves with a different velocity ($v_1 = 2m/s$ and $v_2 = 1.5m/s$), while they have to maintain the center of the line ($l_1^d = l_2^d = 0$). The value chosen for γ in the CBF is 0.5, and the prediction horizon is equal to $2s$. The QP solver used is FULL_CONDENSING_HPIPM, while the NLP solver is SQP_RTI.

The initial condition and all other parameters are equal in both the cases, so that the only discriminant is the controller (centralized or independent).

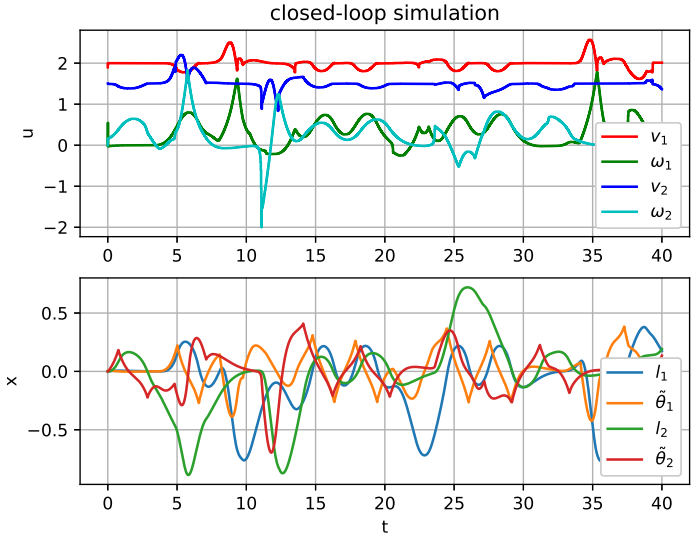
Centralized cars In this experiment, we analyzed the behaviour of two cars controlled by a single controller that knows the state, the target and the intention of both cars, joined to the knowledge of the position and velocity of each obstacle. As both cars are controlled by a single controller, we expect an optimal negotiation when these two cars intersect each other.

This is the result of the simulation for the centralized approach:



Independent cars In this experiment, we analyzed the behaviour of two independent cars controlled by two different controllers. Each car knows the position and the target velocity of the other car and of the obstacles, but it has no way of knowing the prediction of the other car (i.e. the future states that the other car has predicted). Dealing with such incomplete knowledge, we expect that the cars don't follow the optimal trajectory, but they act in a more conservative way. This is more evident when they meet at the intersection; in fact, each car is assuming that the other will proceed at its target velocity, so they both slow down or recompute trajectory in a way to be more cautious.

This is the result of the simulation for the independent approach:



Now, we want to emphasize the difference between these two approaches focusing on a specific situation.

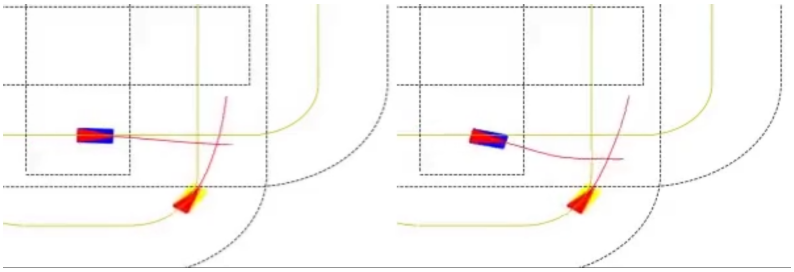


Figure 24: On the left the centralized controller managed to find a smooth and undemanding trajectory from the point of view of inputs, while on the right the independent controllers are predicting to follow two trajectories that are a bit longer and require more control effort than what is actually needed.

On the other hand, the centralized controller becomes more and more complex as the number of controlled cars increases, so in a real world application we couldn't rely on this approach in an extensive way, but we should prefer the independent one.

5.6 Dynamic extension

Finally, we do a so-called dynamic extension, which consists in changing the inputs of our model. In fact, we use as new inputs the linear acceleration a and the angular acceleration a_ω instead of velocities; for this reason, two integrators in the model are needed, which now becomes:

$$\begin{cases} \dot{l} = v \sin \tilde{\theta} \\ \dot{s} = \frac{v \cos \theta}{1 \pm \kappa(s) l} \\ \dot{\tilde{\theta}} = \omega \pm \frac{\kappa(s) v \cos \tilde{\theta}}{1 \pm \kappa(s) l} \\ \dot{v} = a \\ \dot{\omega} = a_\omega \end{cases} \tag{16}$$

Now, the state variables are $q = \begin{bmatrix} l & s & \tilde{\theta} & v & \omega \end{bmatrix}$, while the output ones are $y = \begin{bmatrix} l & s & \tilde{\theta} \end{bmatrix}$.

5.6.1 Modifying our CBF: Discrete vs Continuous time

Since we are using two different inputs at the acceleration level, the relative degree changes and becomes equal to 2. At discrete time, the CBF remains the same, because no matter what the relative degree of the continuous-time system, the relative degree of the discretized equivalent model is always equal to 1. Therefore, at continuous time we have to use an exponential CBF because of a relative degree larger than 1. In fact, we will consider also the second derivative of the CBF in the constraint so that the input appears.

Before testing the different formulations of the CBFs, we run two experiments without obstacles in order to compare the computational times of both discrete and continuous time.

The trajectories we obtain are the following:

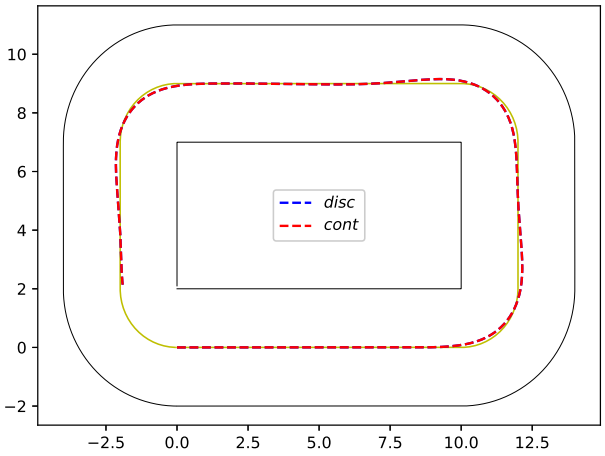


Figure 25: Plot of the resulting trajectories applying the discrete and the continuous version of the controller in a situation without obstacles

It makes sense that the trajectory is exactly the same, in fact we are optimizing the same cost function, without any CBF constraint. Also the plots of the control inputs and state variables look almost identical:

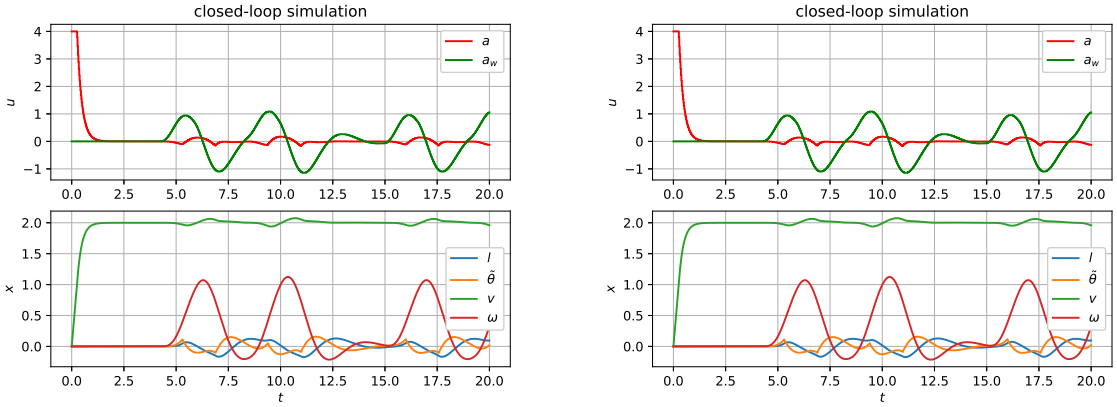


Figure 26: Continuous vs Discrete time

In this case, the computational times we obtained are the following:

	mean time	std dev	max time	cost integral
Continuous Time	0.0012	0.0002	0.0034	12.7786
Discrete Time	0.0012	0.0002	0.0034	12.7786

As expected, the results are identical (except for approximations). Later, we will test whether the introduction of the CBF, and thus of the obstacles, affects the computational times.

Continuous time In this case, we have to consider the Exponential Control Barrier Function. As we explained in 2.2.1, the exponential CBF for a system with relative degree equal to 2 is given by the following formula:

$$\ddot{h}(x) \geq -\begin{bmatrix} K_1 & K_2 \end{bmatrix} \begin{bmatrix} h(x) \\ \dot{h}(x) \end{bmatrix}$$

We have already computed the terms $h(x)$ and $\dot{h}(x)$ in 5.3.1. In the case of the Exponential CBF, we also need to compute the term $\ddot{h}(x)$, in order to have the possibility to apply the constraint.

$$\ddot{h}(x) = \frac{12(\dot{s}_c - \dot{s}^i_{obs})^2(s_c - s^i_{obs})^2 + 4\ddot{s}_c(s_c - s^i_{obs})^3}{(2l_1)^4} + \frac{12(\dot{l}_c - \dot{l}^i_{obs})^2(l_c - l^i_{obs})^2 + 4\ddot{l}_c(l_c - l^i_{obs})^3}{(2l_2)^4} \tag{17}$$

We are going to make experiments by changing the value of K_a.

K_a affects the position of the closed-loop poles, which are required to be real and negative. The choice of the poles will affect the way in which the car responds to the inputs; the more the poles will be near the imaginary axis, the slower will be the behaviour in response to the given input.

For a first comparison, we are going to consider moving obstacles and we are going to set a zero initial pose for the controlled car. Moreover, the car has to move with a target velocity of $2.0m/s$ and with a prediction horizon $T_f = 1$.

At first, we consider $K_a = \begin{bmatrix} 225 & 30 \end{bmatrix}$, so the two poles are both positioned at -15 along the real axis.

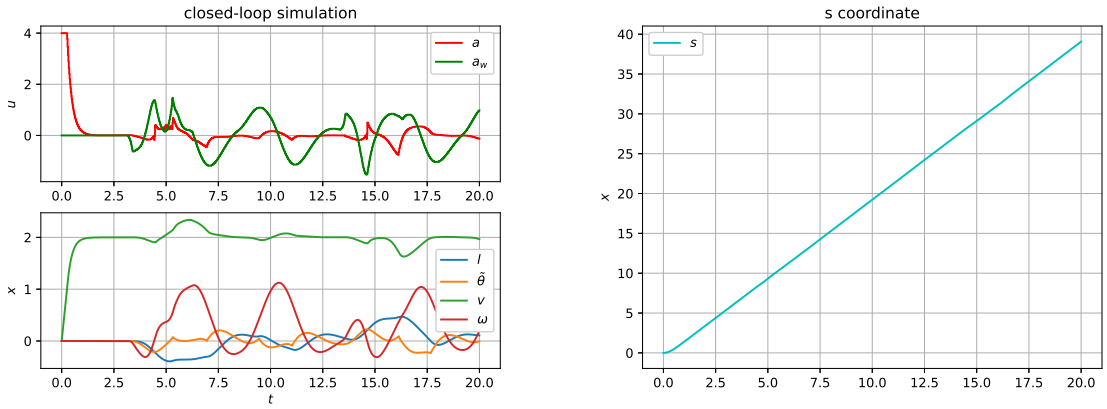


Figure 27: Plot for $K_a = \begin{bmatrix} 225 & 30 \end{bmatrix}$

As we can notice from the plot, the linear velocity v oscillates around its desired value, which is $2.0m/s$. Also l oscillates when the car has to overtake the obstacles. As expected, s grows linearly; this implies that the car does not block or hit against the obstacle, but it is able to follow the desired path in a correct way.

For a second experiment, we wanted to prove that the value of K_a is very relevant in determining the system's evolution. So, we have considered $K_a = \begin{bmatrix} 0.01 & 0.2 \end{bmatrix}$, in a way that the two poles are unnaturally small, in fact they are both positioned at -0.1. In this case, the response of the system will be much slower, so also the behaviour is not an optimal one. We can understand this from the plot:

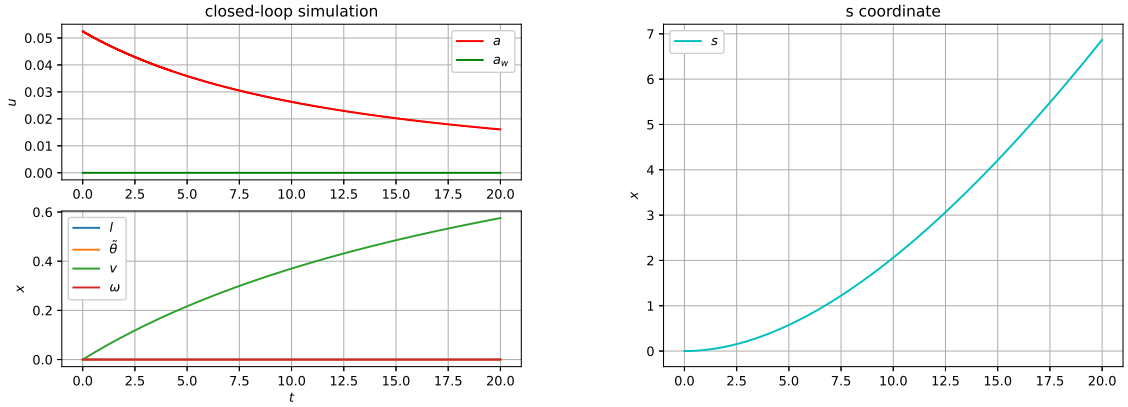


Figure 28: Plot for $K_a = \begin{bmatrix} 0.01 & 0.2 \end{bmatrix}$

In $20s$ of simulation, the car has moved only few meters. Moreover, also the controls have very small value; they are very close to be zero. In fact, the car is not even able to reach the first obstacle.

From the two simulations, it is evident that the choice of the poles is extremely important in order to determine the behaviour of the system.

Discrete time In this case, we don't need to rely on the Exponential Control Barrier Function because after a finite delay the control inputs will appear in the output. Therefore, the only thing that we need to change is the kinematic model of the car by introducing the two integrators; we are not going to change the CBF. This implies that the parameter that we are going to change to tune the behaviour of the car against the obstacle is still γ.

We are going to make a simulation considering $\gamma = 0.3$ and the prediction horizon as $T_f = 1s$. The car has to keep a target velocity of $2.0m/s$ and it starts from a zero configuration. The obstacles move along the path. We got the following results:

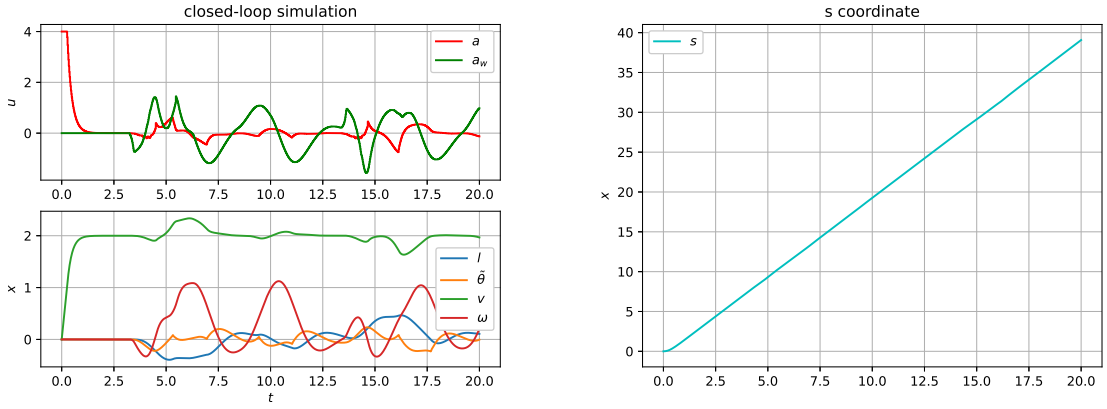


Figure 29: Plot for dynamic extension in discrete time

As we can see, the car is almost always able to keep the linear velocity target value. Moreover, l is always close to zero and oscillates around this value when the car has to overcome the obstacles.

This behaviour is quite similar to the continuous time one; therefore, it would be convenient to use the discrete time model instead of the continuous one because in that case we should design an Exponential CBF that is more complex.

The behaviour of the two controllers is also very similar as shown in the plot of the trajectories.

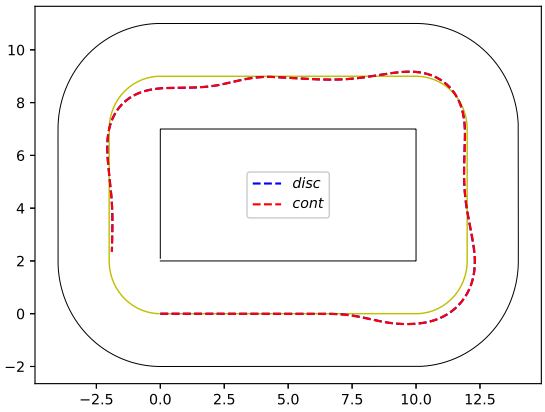


Figure 30: Plot of the resulting trajectories appling the discrete and the continuous version of the controller

Finally, we can study the time that the solver requires to solve the problem. We set the solver to be the same in the continuous and in the discrete case (SQP_RTI and FULL_CONDENSING_QPOASES), and we analyzed the results.

	T_f	γ/K_a	mean time	std dev	max time	cost integral
Continuous Time	1.0	$\begin{bmatrix}225 & 30\end{bmatrix}$	0.0025	0.0013	0.0101	15.0242
Discrete Time	1.0	0.3	0.0029	0.0012	0.0101	15.3281

As we can see, the complexity is very similar in both cases; the only difference is in the mean time that is higher in the discrete case with respect to the continuous one.

6 Conclusion

The goal of this project was to take advantage of both Control Barrier Functions and Model Predictive Control; we have proved that Control Barrier Function constraints can be used within the Model Predictive Control framework to ensure safety. In fact, nowadays, safety is one of the main challenges especially in the field of autonomous systems; these systems should be able to avoid that undesirable things happen. In our project, we have mainly focused on obstacle avoidance.

We have seen the advantages of CBF constraints, for instance the fact that they allow the system to avoid obstacles even when it is far from them. Moreover, it is possible to design a well suited CBF for the specific problem.

We have investigated different scenarios with increasing complexity; as the problem complexity increases, so does the computational time. Therefore, there might be problems in applying this to real-time applications; a new formulation of the problem is needed in such a way of speeding up things. In fact, in all the simulations, we have almost always used Real-time iteration (RTI) so to speed up things; if we don't use RTI and we want a result that is accurate enough, even for few seconds of simulations, minutes or even hours could be needed; this is the main drawback.
However, in this work we have obtained very good results; in most of the simulations, the car always safely races and overtakes the obstacles; it is able to keep the desired path and the desired target velocity even in the more challenging scenarios.

References

[1] Jun Zeng, Bike Zhang and Koushil Sreenath, *Safety-Critical Model Predictive Control with Discrete-Time Control Barrier Function*

[2] Aaron D. Ames, Samuel Coogan, Magnus Egerstedt, Gennaro Notomista, Koushil Sreenath and Paulo Tabuada, *Control Barrier Functions: Theory and Applications*

[3] J. Płaskonka, *The path following control of a unicycle based on the chained form of a kinematic model derived with respect to the Serret-Frenet frame.*

[4] Siciliano Bruno, Sciavicco Lorenzo, Villani Luigi, Oriolo Giuseppe, *Robotics: Modelling, Planning and Control*

[5] Robin Verschueren, Gianluca Frison, Dimitris Kouzoupis, Niels van Duijkeren, Andrea Zanelli, Branimir Novoselnik, Jonathan Frey, Thivaharan Albin, Rien Quirynen, Moritz Diehl *acados – a modular open-source framework for fast embedded optimal control*