Universidade NOVA de Lisboa

School of Science and Technology

Department of Electrical and Computer Engineering

Cyber-Physical Control Systems

Project - Part I & II

André Branco 59758
Gonçalo Rombo 60061
Roberto Teixeira 60423
Tomás Pedreira 60854

Supervised by
Prof. Daniel Silvestre (Ph.D.)

Fall Semester of 2024

Table of Contents

List of Figures

List of Tables

Abstract

This report is divided into two main parts. The first part describes the development and implementation of controllers for cyber-physical control systems applied to mobile robots. The tasks addressed include position control, trajectory tracking, and cooperation among multiple robots. The controllers were designed to meet speed and real-time computation constraints, with performance evaluated using metrics that assess the accuracy and efficiency of the robots' movements. The work emphasizes the use of optimization and discretization techniques to ensure robust and effective control, with simulations demonstrating the feasibility and effectiveness of the proposed solutions.

The second part focuses on implementing one of the tasks from the first part in a real-world system. This requires translating the code developed in `MATLAB` into `Python` to create an interface between the software and the hardware. The hardware will be simulated using `ROS` (Robot Operating System). While the final step would ideally involve testing the implementation in a real-world scenario, this will not be conducted due to the arena being unavailable for use at this stage.

Chapter 1

Introduction

The field of Cyber-Physical Control Systems (CPCS) integrates computational algorithms with physical processes to create highly responsive and adaptive systems. In recent years, mobile robots have emerged as a prominent application of CPCS, offering solutions across various domains, including autonomous navigation, industrial automation, and cooperative robotics.

This first project focuses on the design and implementation of control strategies for mobile robots, addressing key challenges such as trajectory tracking, cooperative behavior, and real-time computational constraints. By leveraging advanced optimization and control techniques, the project aims to enhance the performance and robustness of robotic systems, with results validated through simulation and performance analysis.

The challenge of this project lies in executing the movements described in the outlined tasks. It allows flexibility in adopting any controller structure, modifying cost functions, and selecting the discretization method that best suits each situation.

Additionally, a performance metric, calculated with a sampling interval of 5 ms, will be incorporated as follows:

$$c(k) = 10\|\mathbf{x}(0.005k) - \mathbf{x}_{\text{ref}}(0.005k)\|^2 + \|\mathbf{u}(0.005k)\|^2 + \text{penalty}(k) \tag{1.1}$$

The penalty function is defined as $10e^{200(\text{cpuTime}(k)-0.025)}$, which exponentially increases the cost if the CPU time exceeds 25 ms. Given the objective of real-time computation, any value above 30 ms will be considered a computational failure, and the last control action will be used as the controller's current output.

To minimize this metric, choices will be carefully made to reduce processing time while maintaining the required functionality and performance.

With the introduction of the first part of the project complete, we now proceed to the second phase.

This second part of the project builds upon the foundations established in its first phase. While the initial phase focused on applying control techniques and optimization strategies within simulated environments, it lacked a direct pathway for implementation on physical robotic platforms.

In this second phase, the goal is to bridge this gap by implementing one of the tasks previously developed in a real-world system. This involves utilizing the Robot Operating System (ROS), an open-source middleware widely used in robotics for its robust communication framework and hardware abstraction capabilities. Additionally, the NVIDIA Jetbot platform, powered by the Jetson Nano, serves as the hardware interface, offering real-time AI processing capabilities and sufficient computational power for optimization-based controllers.

The primary objective is to integrate the simulation with physical systems using the ROS-based software developed for the FCT-UNL robotics arena. This involves translating the MATLAB code to Python and ensuring seamless compatibility with ROS for efficient communication between software and hardware. However, due to the unavailability of the arena, the scope of this work is limited to simulations. Nonetheless, this project establishes a robust foundation for future physical implementations in this application.

With the introduction of the second part of the project complete, we now turn to the analysis of the robot's dynamic model.

Chapter 2

Robot Dynamic Model

In this chapter, we will address the dynamic model of the considered system and all aspects related to the physics of this device's movement that must be taken into account. The following figure presents all the variables we need to consider:

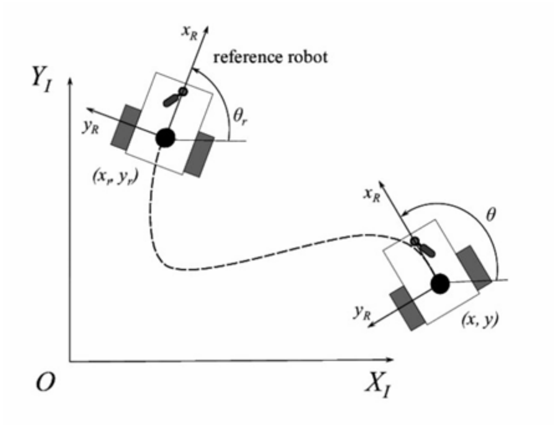


Figure 2.1: Quadrotor Reference Frames.

As illustrated in Figure 2.1, let us consider a ground robot whose inputs correspond to the linear velocities of the left and right wheels, v_l and v_r. To simplify its model, the differential equations can be described based on the linear velocity $v = 0.5(v_r + v_l)$ and the angular velocity $\omega = \frac{(v_r - v_l)}{b}$, where b represents the distance between the wheels (track width).

The full continuous-time dynamics can be expressed by the following set of equations:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \mathbf{u} \tag{2.1}$$

where the control input is given by $\mathbf{u} = [v\ \omega]^\top$.

The robots are subject to a set of constraints, such as a maximum speed of $5\,\mathrm{m/s}$ and acceleration limited to $1\,\mathrm{m/s^2}$. It is also assumed that the vehicle always starts from the origin of the inertial frame, with the initial attitude as the identity and at rest.

Whenever tasks do not specify a reference value for some state variables, those values will be assumed to equal their current state, meaning they will not contribute to the performance metric. To minimise interference from processing time, these values will be collected through a Monte Carlo simulation with 10 runs.

With the problem's requirements and constraints properly specified, we can proceed to the development of the project.

Chapter 3

Project - 1st Part

In this chapter, we will explain what was used and done in the first part of the project. All controllers implemented and the results for each project question will be presented, leading to the necessary conclusions.

3.1 controllerSimple

In this question, we need to implement a controller that drives the robot to the reference point $[10, 10]$. For this purpose, we implemented both an MPC and a Pure Pursuit controller.

3.1.1 MPC

Our first approach was to implement an MPC (Model Predictive Controller).

With this in mind, the first step was to define key parameters for the MPC. A discrete-time approach was adopted, in which the state of the robot is updated at regular intervals, defined by a sampling time of $T_s = 5\,\text{ms}$. The controller predicts the system behavior on a prediction horizon of $N = 4$, enabling a forward-looking strategy to compute optimal control actions.

The state vector, x, represents the robot's position and orientation, while the control vector, u, corresponds to the wheel velocities. To regulate the trajectory, the weight matrices Q and R were utilized. These matrices penalize deviations from the desired reference trajectory and excessive control efforts, respectively.

Another critical consideration was the incorporation of constraints on maximum velocity ($5\,\text{m/s}$) and acceleration ($1\,\text{m/s}^2$), ensuring that the system remains within the predefined limits.

Cost Function

The implementation of the MPC employs a cost function designed to balance trajectory tracking accuracy and control effort. The primary goal is to minimize the deviation between the robot's predicted state and the desired reference trajectory while penalizing excessive control inputs.

This is achieved through a weighted quadratic penalty, as represented in [3.1].

$$J = J + (x_{\text{index}} - [10; 10; 0])^\top Q (x_{\text{index}} - [10; 10; 0]) + u_{\text{index}}^\top R u_{\text{index}} \tag{3.1}$$

Simulation Process

The robot starts at the origin with an initial velocity of zero. At each simulation step, the MPC algorithm solves an optimization problem using the YALMIP optimizer, which computes the control input required to minimize the cost function while satisfying the defined constraints. These inputs are then applied to the robot, and its new state is determined by integrating the kinematic equations using the ode45 solver. This process is repeated iteratively throughout the simulation. The resulting trajectory is presented in Figure 3.1.

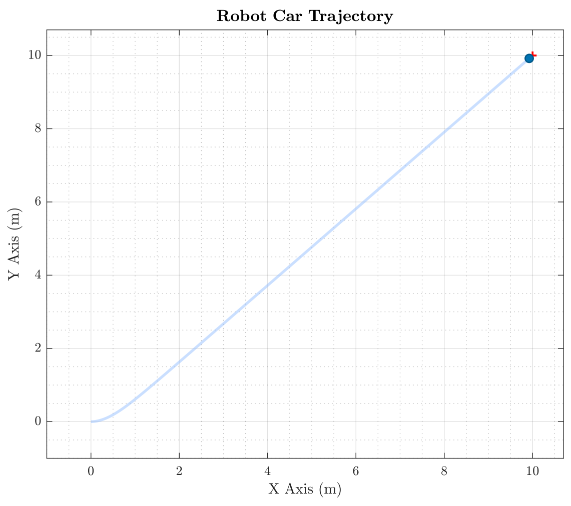


Figure 3.1: controllerSimple with MPC Result.

In addition, during one iteration, we calculated the execution time for each control cycle. The results, shown in Figure 3.2, confirm whether the control computation remains within the two defined limits: 25 ms (which would incur a performance penalty) and 30 ms (beyond which the control action for that instant would be discarded).

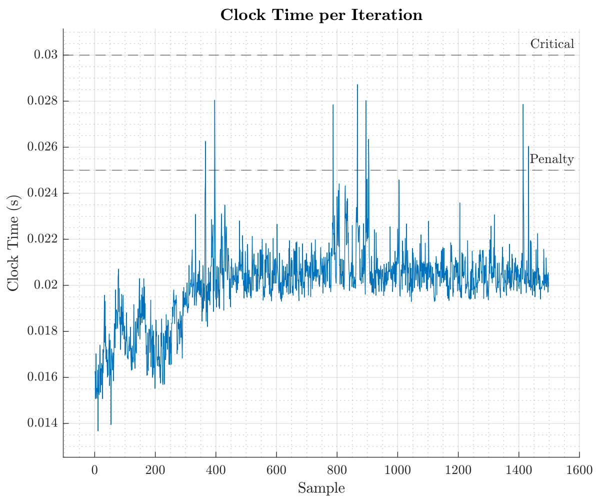


Figure 3.2: controllerSimple with MPC Clock Time per Iteration.

Finally, we performed a Monte Carlo simulation to calculate the overall average computation time over 10 iterations. The output obtained was roughly 21 ms.

Analyzing Figure 3.1, the simulation shows that the MPC effectively guides the robot to the target position while respecting the imposed velocity and acceleration restrictions. The robot successfully tracks the desired trajectory with minimal deviation, highlighting the robustness of the MPC framework. The weight matrices Q and R play a crucial role in achieving a balance between tracking accuracy and control effort, allowing fine-tuning of the controller's performance. However, the computational demand to solve the MPC optimization problem poses a significant challenge. As the prediction horizon increases or the dynamics of the system becomes more complex, the computational load grows, potentially limiting the applicability in real time. To address this, strategies such as reducing the prediction horizon or employing more efficient optimization solvers where implemented.

Analyzing Figure 3.2 and the Monte Carlo simulation, we observe that although the MPC successfully guides the robot to the desired target point, its computational complexity results in slower performance than desired, leading to multiple penalties throughout the process. As an alternative, we implemented a Pure Pursuit controller, which will be presented in the next point of this report.

3.1.2 Pure Pursuit

Due to the long simulation time of the MPC, we also decided to implement a Pure Pursuit controller. The Pure Pursuit controller was adapted from [1]. This paper explains that the Pure Pursuit algorithm, as described in Equation 3.2, calculates the steering angle required to guide a vehicle, in this case, a mobile robot, back to the reference path. The algorithm uses a look-ahead point, denoted as l_d in Equation 3.2, which acts as a target point along the desired trajectory that the vehicle should pursue. This look-ahead point is set to a value of 0.1 m, obtained through simulation. Note that this is the only parameter that can be adjusted in the expression presented in Equation 3.2, allowing us to fine-tune the controller. For smaller values, the trajectory becomes more direct toward the reference point, while for larger values, a smoother and more curved trajectory is achieved. Furthermore, the algorithm considers the vehicle's current orientation error, represented as α_{error}, and its wheelbase between the front and rear axles, represented as H.

$$\phi = \arctan\left(\frac{2H \cdot \sin \alpha_{error}}{l_d}\right) \tag{3.2}$$

In Figure 3.3, we present the results obtained using this controller.

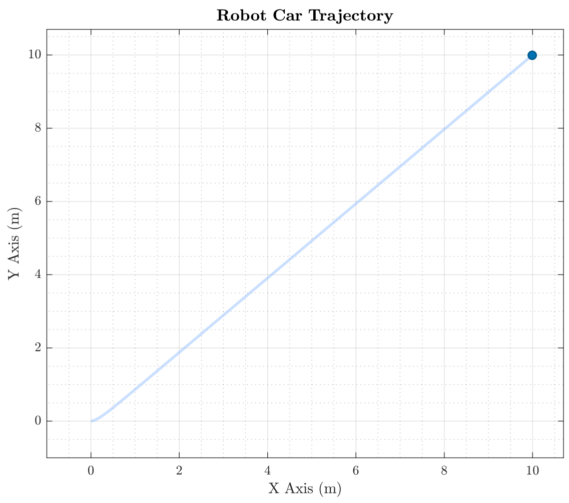


Figure 3.3: controllerSimple with Pure Pursuit Result.

Upon analyzing Figure 3.3, we observe that the mobile robot exhibits highly positive behavior, moving directly toward the desired point. Following this, we performed another simulation to calculate the execution time for each control cycle. The results are presented in Figure 3.4.

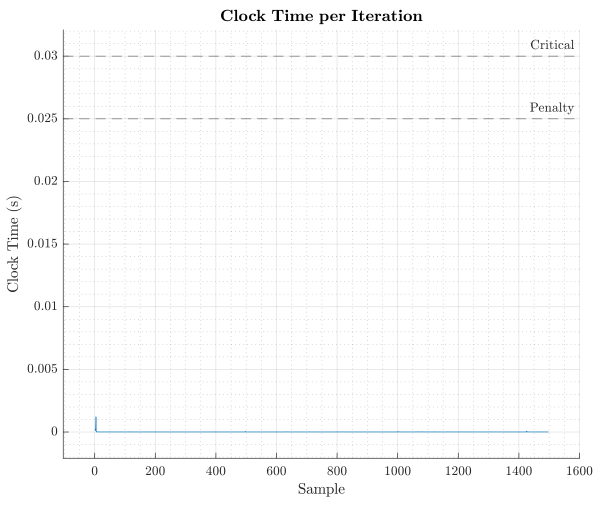


Figure 3.4: controllerSimple with Pure Pursuit Clock Time per Iteration.

Lastly, we conducted a Monte Carlo simulation to determine the overall average computation time over 10 iterations. The output obtained was 0.0077068 ms.

We conclude this section successfully, as both the MPC and Pure Pursuit controllers effectively guided the mobile robot from the point [0, 0] to the point [10, 10], as illustrated in Figures 3.1 and 3.3. However, the MPC exhibited longer computation times than expected, as shown in Figure 3.2, which motivated us to implement the Pure Pursuit controller. The Pure Pursuit method demonstrated significantly faster performance, as observed in Figure 3.4.

We now proceed to the next section, where we analyze a more complex trajectory.

3.2 controllerTracking

In this section, we implement a controller to follow a parameterized curve defined as a function of time $0 \leq t \leq 20\pi$, presented in Equation 3.3.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5\sin\left(\frac{t}{5}\right) \\ 5\cos\left(\frac{t}{10}\right) \end{bmatrix}. \tag{3.3}$$

Given the performance of the MPC and Pure Pursuit controllers in the previous section, we decided to implement a Pure Pursuit controller for this task. The look-ahead distance was set to 0.4 through simulation, a parameter that was previously explained. The resulting output trajectory is shown in Figure 3.5. It is important to note that the robot starts at the point [0,0], resulting in an initial adaptation phase. Once the robot reaches the reference curve, it follows it in a highly stable and accurate manner, as demonstrated in Figure 3.5.

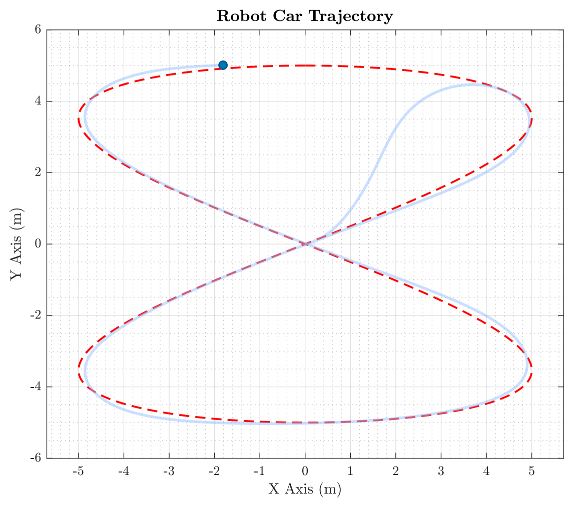


Figure 3.5: controllerTracking with Pure Pursuit Result.

After confirming that the obtained trajectory tracks the reference curve effectively, we analysed the clock time per iteration, presented in Figure 3.6.

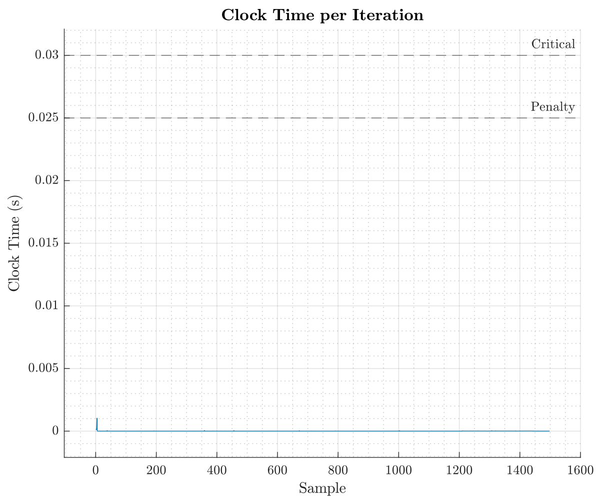


Figure 3.6: controllerTracking with Pure Pursuit Clock Time per Iteration.

Upon analysing Figure 3.6, we observe that at no point does the execution time approach the predefined threshold, which would introduce penalties to the controller's performance. This behaviour is an excellent indicator of the controller's proper functioning.

Finally, we performed a Monte Carlo simulation, obtaining an overall average computation time over 10 iterations of 0.0052713 ms. This is an extremely low value, further highlighting the excellent performance and computational efficiency of the Pure Pursuit controller.

We successfully conclude this section, as the implemented Pure Pursuit controller tracks the dynamic reference curve very effectively, as shown in Figure 3.5. Additionally, Figure 3.6 and the results of the Monte Carlo simulation demonstrate that the system operates at exceptional speed, remaining well within the imposed thresholds.

We now proceed to the next section, where we analyse a cooperative controller.

3.3 controllerCooperative

In this problem, we now have to control two robots with a maximum velocity of 2 m/s in order to make them follow, 1 meter below, a target vehicle with a constant speed of 10 m/s and a starting position of $(-5, 0)^\top$. For this, we once again developed, for each car, a Pure Pursuit controller that takes the states of the two robots as a combined stacked vector and outputs a stacked vector of the two control inputs. It is important to mention that due to the target vehicle path being known, we use its information on both robot controllers.

Regarding both robots' reference, we set them to 30 meters ahead of the actual position of the target vehicle in order to provide predictive capability and thus try to counteract their low maximum speed. Furthermore, through simulation, we also observed the existence of discontinuity points in the control action of the robots, caused by a high orientation error. For error values greater than 120°, the robots exhibited significant difficulty in resolving this issue due to the low action angles α generated by each robot at each iteration. To address this problem, we implemented a condition where, if the orientation error reaches excessively high values, we assign a null value to the speed of the vehicles and perform only the adjustment of the orientation error until more plausible values are achieved. Otherwise, we maintain the conditions described in the problem statement related to velocity and acceleration.

Both implemented Pure Pursuit controllers have similar configuration, and use Equation 3.2 for the steering angle calculation. Regarding the fine-tuning of this equation we defined, by simulation, a look ahead distance of 0.4. Furthermore, through the verification process, we noticed that the car's steering angles (α) had very small values. Therefore, we multiplied them by a gain factor of 10 in order to speed up the angle adjustment process and, consequently, accelerate convergence. Given that the vehicles move at very low speeds, this will not cause any problems.

Related with the simulation process, we did two simulation, A and B, where we considered the following initial positions and orientation for the two cars:

- Simulation A:

Car	**X Pos**	**Y Pos**	**Orientation**
Car 1	90	-0.25	0
Car 2	10	-1.5	0

Table 3.1: Position of the cars in simulation A.

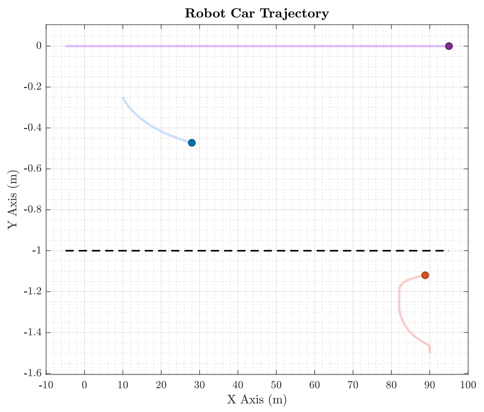
- Simulation B:

Car	**X Pos**	**Y Pos**	**Orientation**
Car 1	10	-0.25	0
Car 2	90	-1.5	0

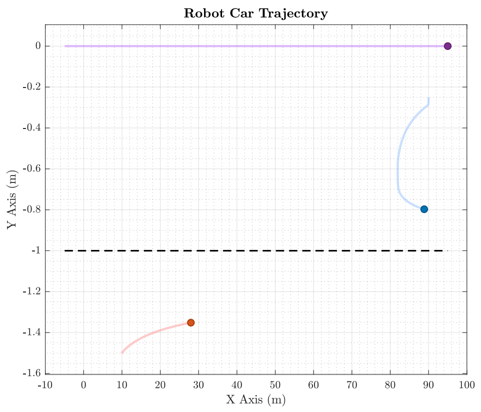
Table 3.2: Position of the cars in simulation B.

These initial points were chosen in such a way as to perform the simulation at opposite points, thus proving that convergence occurs over a wide range of values and areas on the map.

The next figures represent the results obtained for simulation A and B, respectively:



(a) 1st and 2_{nd} Car Simulation A



(b) 1st and 2_{nd} Car Simulation B

Figure 3.7: controllerCooperative with Pure Pursuit Result.

Based on the previous graphs, it can be observed that both Robot 1 (orange plot) and Robot 2 (blue plot) converge to the reference trajectory (black dashed line), which has the same path as the target vehicle (purple plot) but 1 meter below. Furthermore, it can be observed that the convergence occurs regardless of the initial positions chosen, as evidenced in both Figures 3.7a and 3.7b. The fact that the robots do not completely reach the target is due to the low velocity set for these vehicles. This does not affect the convergence itself but rather the time it takes to achieve convergence.

Upon verifying the correct convergence of both robots to their reference, we move on to analyse the clock time per iteration:

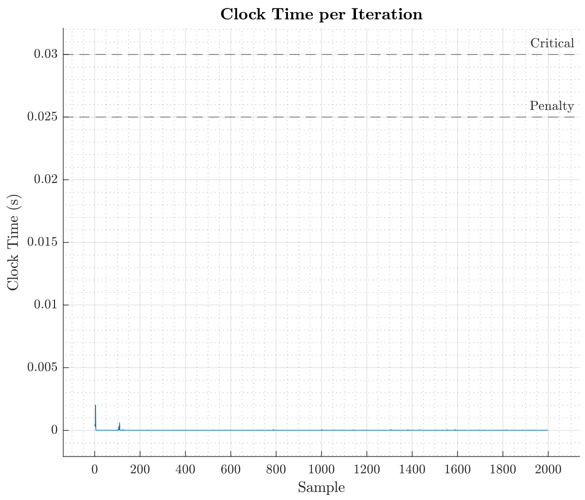


Figure 3.8: controllerCooperative with Pure Pursuit Clock Time per Iteration.

As can be observed, the simulation time remains well below both the penalty and critical thresholds. The initial iteration, which involves a step where the CPU was coincidentally more heavily utilized, as indicated by the small peak in processing time, still presents an excellent clock time, not coming close to the penalty or critical thresholds.

Next, a Monte Carlo simulation was performed to determine the average computation time over 10 simulations, which resulted in 0.012141 ms. This slight increase in average computation time, when compared with the previously obtained values, is due to the two if-else conditions used. Still, it remains well below the penalty and critical limits, highlighting once again the computational efficiency of the Pure Pursuit controller.

With this, we conclude that the implementation of this controller was successful, as we achieved reference tracking (or convergence) for different initial robot positions, as demonstrated in Figure 3.7. Additionally, Figure 3.8 and the Monte Carlo simulation results both demonstrate that the implemented controller operates at an exceptional computational speed.

3.4 controllerMovie

In this section, we address the final task of the first part of the project: implementing a controller that positions the robots on a circular trajectory centered around the target vehicle with a radius of 2 meters. The requirement is for the two cars to remain out of phase during rotation and complete two full spins, each in 40 seconds.

The first step was to create a function that generates the circular reference trajectory for both robot cars. Figure 3.9 presents this reference trajectory, which is generated at the start of the first iteration.

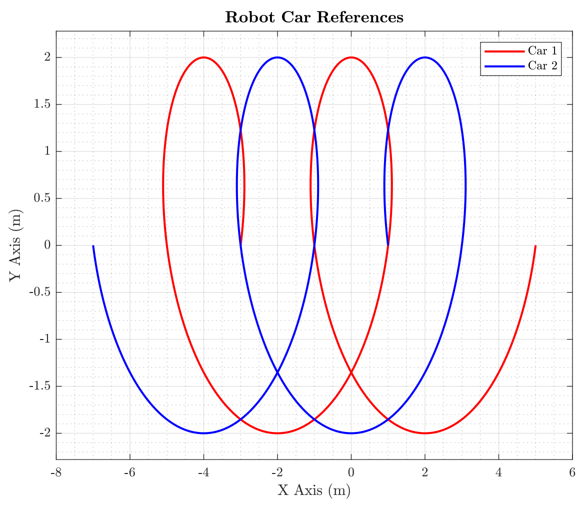


Figure 3.9: Robot Cars References.

Following this, the process was similar to the previous tasks. We implemented a Pure Pursuit controller with a look-ahead distance of 0.4 m, defined through simulation. The robots start at arbitrary points, in our case, the first starts at [-3;-2;$\pi/2$], the second at [-7;2;-$\pi/2$] and the reference robot at [-5;0;0]. It is important to note that in this task, the orientations of the cars are aligned with the first point of their respective reference trajectories. This ensures that they do not need to correct their direction at the start of the simulation.

During the execution, another parameter was fine-tuned via simulation: the reference vehicle's velocity. Initially, a velocity of 2.5 m/s was suggested by the professor. However, during implementation, we observed that at this speed, it was impossible for the cars to accurately follow the reference vehicle's trajectory while at the same time completing the circular rotations around it. Consequently, this value was reduced to 0.1 m/s.

Thus, the obtained output is presented in Figure 3.10.

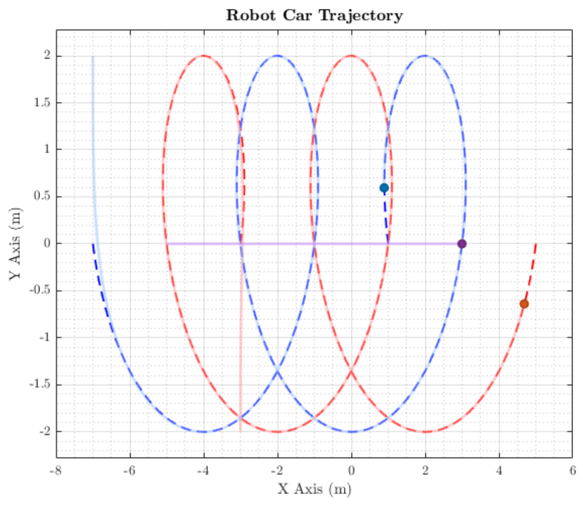


Figure 3.10: controllerMovie with Pure Pursuit Result.

Upon analysing Figure 3.10, we observe that the trajectories are exactly as expected. The cars successfully complete two full rotations around the reference vehicle while maintaining the required out-of-phase configuration, which can be clearly seen at the point where the simulation ends.

After confirming that the obtained trajectory meets the specified requirements, we analysed the clock time per iteration, as shown in Figure 3.11.

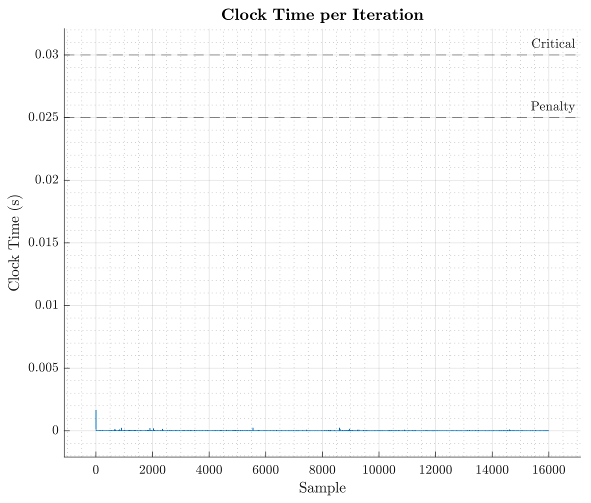


Figure 3.11: controllerMovie with Pure Pursuit Clock Time per Iteration.

Upon analysing the previous figure, we observe that at no point does the execution time approach the predefined threshold, which would introduce penalties to the controller's performance. This behaviour is an excellent indicator of the controller's proper functioning. It is worth noting that the initial iteration takes longer than the subsequent ones. This makes sense, as the reference trajectory is calculated during the first iteration, leading to higher computational time.

Finally, we performed a Monte Carlo simulation, obtaining an overall average computation time over 10 iterations of 0.0074468 ms. This is an extremely low value, further highlighting the excellent performance and computational efficiency of the Pure Pursuit controller.

We successfully conclude this section, as the implemented Pure Pursuit controller effectively tracks the reference robot while rotating around it, as shown in Figure 3.10. Additionally, Figure 3.11 and the results of the Monte Carlo simulation demonstrate that the system operates at exceptional speed, remaining well within the imposed thresholds.

Thus, we conclude the first part of the project and can proceed to the next chapter, where we will focus on implementing one of these controllers using ROS in Python.

Chapter 4

Project - 2nd Part

In this chapter, we will explain the work completed in the second part of the project, presenting all the results obtained and the conclusions drawn.

4.1 Python Implementation

The first step in this second part of the project was to implement one of the controllers developed and presented in the first part of the project using Python. Following the professor's suggestion, we decided to implement the Pure Pursuit controller, as previously used in controllerMovie.

To achieve this, the work conducted in MATLAB was translated into Python, including the function to generate the reference trajectories, the Pure Pursuit controller function, and the testbench to validate the controller in Python. All these implementations can be reviewed in the files submitted with this report. Figure 4.1 shows the output obtained in Python.

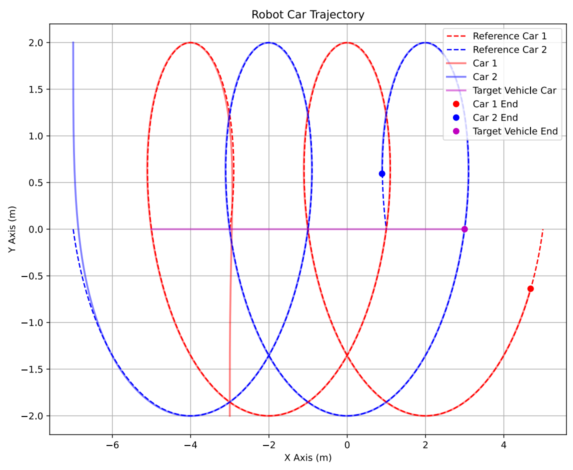


Figure 4.1: controllerMovie with Pure Pursuit Python Implementation Result.

By comparing Figure 4.1 with Figure 3.10, we can confirm that the controller's output in Python matches the behaviour observed in MATLAB during the first phase of the project. This successfully validates this step of the second part of the project.

The next step involves implementing the controller in ROS (Python), making the necessary adjustments on the code.

4.2 Jetbot ROS Implementation

For the Jetbot ROS implementation, we utilized the base code provided in the work statement, which included most of the necessary elements for the connection between the code and the Jetbot and even an example of a generic controller implementation. In this code, we added the necessary functions used in the controller, such as `wrapToPi`, which operates the same way as its original MATLAB counterpart, and `ssCar`, which contains the robot car dynamic continuous-time model equations.

Regarding the controller function, there is an initialization section where we calculate both robot cars' complete trajectories and initialize all necessary variables and arrays used in the code's main loop. In the controller main loop, we read data from the Jetbot, where most importantly, we get both robot cars' X and Y positions and orientation. This means that in this ROS implementation, we do not need to use the `ode45` method as utilized in MATLAB. Using these values, we compute the Pure Pursuit algorithm for both robot cars, resulting in new linear velocities and steering angles that will be used as the next control actions.

An important aspect to mention is that this code supports having two cars execute the maneuvers specified in Section 3.4. This means that if, as mentioned by the Professor, we only have two functional Jetbots and one of them should operate as the target vehicle, the only changes needed in the code would be the substitution of one of the Pure Pursuit algorithms with a constant 0.1 m/s velocity array. This would enable one of the Jetbots to move at 0.1 m/s while the other would circle around it following the desired trajectory.

It is worth noting that, although we did not test the controller in a laboratory environment, the results obtained in Section 4.1 provide a high degree of confidence that the controller would function correctly.

Chapter 5

Conclusion

For this project first part, we developed four different controllers that enabled us to perform a variety of maneuvers, ranging from simple reference tracking for a single robot car to more complex problems, such as a shooting scene for a movie requiring precise operation of two different robot cars.

The robot car we aim to control is defined by a continuous-time set of equations, as presented in (2.1), and has a set of constraints regarding its maximum achievable speed of $5\,\mathrm{m/s}$ and acceleration of $1\,\mathrm{m/s^2}$. The robot car's control action input is given by its linear and angular speeds, both calculated through the linear speeds of the right and left wheels, as presented in Section 1.

The initial task, described in Section 3.1, required us to create a controller to drive a robot to the reference position $[10 \quad 10]^\top$, while its initial position was $[0 \quad 0]^\top$. For this, we initially developed an MPC controller, as detailed in Section 3.1.1. In its early stages, this controller did not allow for real-time computation due to computational times per simulation iteration exceeding the allowed threshold. This prompted an optimization phase, which ultimately yielded a more plausible computational performance, as shown by the clock times depicted in Figure 3.2 and further corroborated by the Monte Carlo simulation, where the overall computation time over 10 simulations was around 21 ms.

Regarding this controller's results, as seen in Figure 3.1, the robot car reaches the reference without any issues. Even though this controller correctly solves the problem it was defined for, we consider that, even after optimization, its computational performance is not ideal. This limitation suggests that for the following problems, where complexity will increase, we might encounter severe issues with real-time computation. Given this concern, we decided to implement a Pure Pursuit controller to determine if we could achieve the desired performance.

The Pure Pursuit controller, described in Section 3.1.2, utilizes Equation 3.2 to calculate the required steering angle to guide the robot car to its desired point. As demonstrated by Figure 3.3, we once again successfully reach the reference position, and this time with a remarkable computational cost, as clearly backed up by the clock time per iteration plot (Figure 3.4) and the Monte Carlo result of 0.007 706 8 ms. These results clearly highlight the overwhelming performance of this controller in terms of computational cost compared to the previously implemented MPC controller. Consequently, we decided to use the Pure Pursuit controller for the remainder of this project.

Moving on to the next problem, we were required to create a controller that permitted the robot car to follow a parameterized curve defined as a function of time $0 \leq t \leq 20\pi$, presented in Equation 3.3, as seen in Section 3.2. This controller implementation was similar to the previous one, with the main difference being that the reference to follow now updated in each simulation iteration. The obtained results, shown in Figure 3.5, were highly positive, demonstrating that, apart from the initial phase where the robot car transitions from its starting point ($[0 \quad 0]^\top$) to the parameterized curve, it accurately follows the reference, exhibiting only small deviations due to its sharp turns. Regarding the computational cost, a most satisfactory result was once again obtained, as demonstrated by the Monte Carlo result of 0.005 271 3 ms and the clock time per iteration of one of the conducted simulations, shown in Figure 3.6.

The next problem, presented in Section 3.3, possessed more of a challenge due to the increase in complexity, as it required controlling two robot cars with a maximum speed of 2 m/s while having to follow, 1 meter below, a target vehicle travelling at a constant speed of 10 m/s and starting at position $[-5 \quad 0]^\top$. With this, it is clearly understood that following the target vehicle would be challenging due to the significant difference in speed. To try to minimize this, and since the target vehicle path is a known variable, we defined both robot car references to be 30 m ahead of the actual position of the target vehicle. In this way, the robot cars could "predict" the target vehicle's movement and thus provide a faster response.

Another important aspect considered was that the initial position of both robot cars is an unknown variable, meaning that we had to make the code robust to correctly behave under various starting scenarios. It was particularly important to consider an example scenario where one of the robots' initial position is close to the target vehicle's final position. In this scenario, the robot car will initially move in the opposite direction of the target vehicle as it attempts to get close to it. At a certain point, however, there will be a drastic change in orientation caused by the robot car reference being in the opposite direction it was initially moving. This drastic change produced some odd behaviours, primarily caused by the small steering angles calculated in this particular problem.

To solve this, we implemented two techniques. The first was to apply a gain of 10 to the previously used steering angle equation, making the car more reactive. The second was that, for orientation errors greater than 120º (as in the described example scenario), we set the robot car speed to zero, thus enabling it to make the necessary orientation adjustments while possessing low speed.

To verify that the implemented controller behaved as expected, we conducted two simulations with different robot car initial positions (see Tables 3.1 and 3.2). In both cases, the robot cars correctly converged to the desired reference despite the challenging difference in velocity, as seen in both plots of Figure 3.7. In terms of computational cost, the results were once again ideal, as demonstrated in Figure 3.8, and with the Monte Carlo simulation result of 0.012 14 ms. A final important remark is that, compared to the other Monte Carlo results, the value obtained for this problem was slightly larger simply due to the use of if-else conditions.

The final task for the first part of the project, detailed in Section 3.4, required the control of two robot cars for a total of 80 s. During this time, they would follow a target vehicle and perform two full revolutions, with a 2 m radius, around it while remaining out of phase during the rotation. For this problem, the only considered constraint was the robot cars' maximum velocity, leaving us to determine the target vehicle's velocity.

This task initially presented some challenges, but after some thought, we arrived at an idea that essentially rendered this problem almost identical to the one described in Section 3.2 for the `controllerTracking`. The idea was based on the fact that the robot cars' trajectory is already defined. Thus, we decided to create a function that would return both robot cars' complete trajectory and then, in the controller, we would call this function in the first simulation iteration and store the trajectories in persistent variables, enabling the robot cars to execute the desired maneuver. The main challenge with this was to correctly obtain the desired trajectories, mainly because the target vehicle's velocity did not allow the car to follow it while simultaneously completing the revolutions around it. This led to the selection of a target vehicle velocity of 0.1 m/s. The obtained references are shown in Figure 3.9, where it can be seen that both trajectories have the desired radius, the correct number of revolutions, and a 180º phase difference between the robot cars.

The controller's results are shown in Figure 3.10, demonstrating that, as expected, the behaviour is similar to the controller presented in Section 3.4: there is an initial phase where the robot cars converge to the reference, followed by accurate execution of the desired maneuver. Another notable observation from this plot is that, at the end of the simulation, both robot cars are approximately 2 m from the target vehicle and remain out of phase. Regarding the computation cost, we obtained once again the desired result, as seen in Figure 3.11 and by the Monte Carlo result of 0.007 446 ms.

With this, we conclude the first part of the project. We highlight the performance analysis conducted for the MPC and Pure Pursuit controllers in the first problem and the excellent results achieved with the Pure Pursuit controllers for the remaining tasks, both in terms of executing the desired maneuvers and achieving outstanding computational efficiency.

Regarding the second part of the project, we had to take the implemented `controllerMovie` in MATLAB and adapt it for use with the Jetbots in the laboratory.

For this, we started by simulating this controller just as was done in MATLAB to see if the Python implementation produced the same result, as shown in Section 4.1. As demonstrated by Figure 4.1, the Python simulation behaved exactly as the one in MATLAB (see Figure 3.10), thus verifying the correct functioning of the new code.

Moving on to the ROS implementation, as outlined in Section 4.2, it was mostly just about taking parts of the Python code and inserting and adapting them within the base code provided in the work statement. Here we highlight the greatest difference, aside from the interface with the Jetbot, being the fact that, due to directly receiving the X and Y positions in addition to the orientation from the Jetbot, there was no need to use the ode45 method as was previously done.

Unfortunately, we were unable to test this code in the laboratory, but taking into account the results obtained in the previously conducted Python simulation (Figure 4.1), we conclude, with a high degree of confidence, that the developed Jetbot ROS implementation for the arena would yield the desired result.

With this, we conclude the entire project, highlighting the excellent performance obtained with the Pure Pursuit controller in part one, and also extending it to the Python implementation executed in part two.

Chapter 6

Appendices

testingComplete: MATLAB script used to test all of the implemented Pure Pursuit controllers as well as to conduct the Monte Carlo simulation. At the beginning of this script, there is a section labeled "Code Setup" where we select which problem we want to test, as well as whether we want (1) or not (0) to plot the clock time per iteration graph or the animated result.

ssCar: MATLAB function used for the robot car dynamics equations in continuous-time, as well as to calculate its linear and angular speed.

controllerSimpleMPC_test: MATLAB script used to solve problem 1 of the project's first part using an MPC controller, as well as to plot the results and perform the Monte Carlo simulation.

controllerSimple: MATLAB function used to solve problem 1 of the project's first part using a Pure Pursuit controller.

controllerTracking: MATLAB function used to solve problem 2 of the project's first part.

controllerCooperative: MATLAB function used to solve problem 3 of the project's first part.

controllerMovie: MATLAB function used to solve problem 4 of the project's first part.

carReferences: MATLAB function used to calculate both robot car trajectories for problem 4 of the project's first part.

testCarReferences: MATLAB script that plots both robot car trajectories for problem 4 of the project's first part.

controllerMovie_Test: Python script used to verify the Python implementation of the MATLAB `controllerMovie`.

controllerMovie_Jetbot: Python script used to implement the MATLAB `controllerMovie` in the Jetbot.

References

[1] Rui Wang, Ying Li, Jiahao Fan, Tan Wang, and Xuetao Chen. A novel pure pursuit algorithm for autonomous vehicles based on salp swarm algorithm and velocity controller. *IEEE Access*, 8:166525–166540, 2020.