Universidade NOVA de Lisboa

School of Science and Technology

Department of Electrical and Computer Engineering

CPCS

Robot Missions

João Cardoso - 60346
Rodrigo Veríssimo - 67133
David Furtado - 58454

Supervised by
Prof. Daniel Silvestre

2024/2025

Table of Contents

List of Figures

Introduction

This project is divided into two parts, each addressing different aspects of the control of autonomous robots. In the first part, we focus on developing and evaluating control strategies for differential-drive robots in a simulated environment, while the second part transitions these strategies to a physical robotics platform for real-world implementation.

In Part I, the project explores optimization-based controllers for differential-drive robots using MATLAB. These controllers are tested in simulations to achieve several specific objectives:

1. Driving the robot to a fixed reference position;
2. Tracking a continuous circular path;
3. Following a faster target vehicle using two robots;
4. Coordinating two robots to record a faster target while completing two revolutions around it from opposite sides.

The control strategies employ proportional feedback to ensure responsiveness and simplicity while adhering to real-time computational constraints. Performance metrics combine tracking accuracy, control effort, and computational efficiency, with penalties for exceeding computational time limits. MATLAB proves to be an effective tool for modeling, control design, and simulation, facilitating an in-depth exploration of these techniques.

In Part II, the focus shifts from simulation to physical implementation. Specifically, it addresses the task outlined in Question 4 of Part I, implementing it on a physical robotics platform using the Robot Operating System (ROS) and the NVIDIA Jetbot. This task involves guiding a robot to orbit a target vehicle while maintaining spatial and orientation constraints, simulating a practical application scenario.

The transition to a real-world platform introduces new challenges, particularly in programming and system integration. ROS provides robust tools and libraries for managing robotic systems, while the NVIDIA Jetbot, powered by the Jetson Nano processor, offers the computational capabilities necessary for executing complex control algorithms in real-time.

This unified report begins with Part I, detailing the theoretical background, control strategies, and simulation results. It then transitions to Part II, emphasizing the modifications and challenges encountered during the implementation phase. Together, these parts demonstrate how well-designed control systems can bridge the gap between theoretical models and practical applications in robotics.

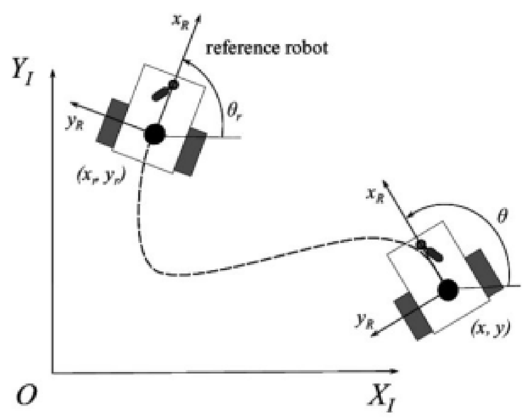


Figure 1: Quadrotor reference frames.

This figure illustrates the reference frames used for representing the robot's kinematics and dynamics. These frames are central to understanding the relationship between control inputs and the robot's movement, providing a foundational framework for both simulation and real-world implementations.

Chapter 1

Part I - Model Derivation and Control Theory

This chapter presents the mathematical derivation of the ground robot model used in this study, specifically focusing on a differential drive system. We examine how the control inputs influence the robot's motion, deriving the kinematic equations that describe its dynamics. It is important to note that while the robot model itself remains constant, constraints such as speed and acceleration may vary depending on the problem or situation at hand.

These constraints are part of the optimization problem, closely tied to the Model Predictive Control (MPC) framework used in this work. For instance, while the robot could theoretically operate at extremely high speeds, practical limits are imposed either by the manufacturer or by the controller to ensure safe and stable operation. The MPC model can adapt to different scenarios by incorporating varying constraints, which influence the minimization process central to the control strategy.

This forms the foundation for the development of adaptive control strategies in later chapters.

1.1 Continuous-Time Dynamics

The goal is to derive the robot's position and orientation over time. Let x and y represent the Cartesian coordinates of the robot's center, and let θ represent its orientation angle with respect to the inertial frame. The continuous-time kinematic equations can be derived by analyzing the robot's movement in the inertial frame, considering its orientation.

Assuming that the robot moves in the x- and y-directions relative to its orientation θ, we express the dynamics as:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}. \tag{1.1}$$

Here, $\dot{x}$ and $\dot{y}$ represent the rate of change of the robot's position in the x- and y-directions (inertial velocity), respectively, while $\dot{\theta}$ represents the rate of change in orientation.

The control inputs u in this model are given as:

$$u = \begin{bmatrix} v \\ \omega \end{bmatrix}. \tag{1.2}$$

This means that the robot's state evolves based on the linear and angular velocities applied at each instant.

1.2 Control Constraints

In real-world applications, the robot's movement is subject to physical limitations. These limitations include both linear and angular constraints, ensuring safe and feasible operation. The constraints are defined as follows:

$$|v| \leq 10 \text{ m/s, for the target,} \tag{1.3}$$

$$\text{and} |v| \leq 2 \text{ m/s, for the followers,} \tag{1.4}$$

$$|a| \leq 1 \text{ m/s}^2, \tag{1.5}$$

where v represents the linear velocity of the robot, and a denotes its linear acceleration. These constraints prevent the robot from reaching excessive speeds or accelerations, which could lead to instability or control difficulties. The limits for this work on speed and angular acceleration were set by the speed limit. but in real applications it can be useful to force these values:

$$|\omega| \leq 2 \text{ rad/s}, \tag{1.6}$$

$$|\alpha| \leq 0.5 \text{ rad/s}^2, \tag{1.7}$$

where ω represents the angular velocity of the robot, and α denotes its angular acceleration.

These constraints limit the robot's rotational dynamics, ensuring smooth and stable turns without abrupt changes in direction or excessive rotational speeds. When designing a controller, it is essential to enforce these constraints at each time step. The controller must adjust v, a, ω, and α accordingly to respect the robot's physical and operational limits.

Let the left and right wheels have linear velocities v_l and v_r, respectively. From these two velocities, we define:

- v: the linear velocity of the robot's center point,
- ω: the angular velocity about the robot's center point,
- b: the wheelbase, or the distance between the two wheels,
- R: the radius of the robot's circular trajectory.

The angular velocity ω is related to the linear velocity v by the radius R of the trajectory:

$$\omega = \frac{v}{R}, \tag{1.8}$$

where R can be expressed as:

$$R = \frac{b}{2} \cdot \frac{v_r + v_l}{v_r - v_l}. \tag{1.9}$$

This relationship highlights how the angular velocity depends on both the linear velocity and the curvature of the path, determined by the radius R. For a straight trajectory ($v_r = v_l$), $R \to \infty$, and thus $\omega = 0$.

1.3 Control Objective and Performance Metric

The control objective is to make the robot follow a designated trajectory while minimizing deviation from this path and adhering to the constraints. To assess the controller's performance, we define a performance metric $c(k)$ that evaluates how well the robot tracks a reference position x_{ref} and control input u over discrete time intervals. This performance metric is expressed as:

$$c(k) = 10 \left\| x(0.005k) - x_{ref}(0.005k) \right\|^2 + \left\| u(0.005k) \right\|^2 + \text{penalty}(k), \tag{1.10}$$

where $x(0.005k)$ is the robot's position at time $0.005k$ seconds, and $x_{ref}(0.005k)$ is the reference position at that instant. The term $\left\| x(0.005k) - x_{ref}(0.005k) \right\|^2$ penalizes deviation from the reference, while $\left\| u(0.005k) \right\|^2$ penalizes large control inputs, thereby encouraging smooth control actions.

The penalty function is defined as an exponential term:

$$\text{penalty}(k) = 10e^{200(\text{cpuTime}(k)-0.025)}, \tag{1.11}$$

which imposes a large penalty if the CPU time taken to compute the control action exceeds 25 milliseconds. If the computation exceeds 30 milliseconds, the controller's action is discarded, and the last control input is reused. This penalty term ensures real-time compliance, essential for practical implementations.

In this model, the robot begins at the origin in an inertial frame, with zero initial velocity and a neutral orientation. Any unspecified reference values for the state variables are assumed to match the current state, so they do not affect the performance metric.

1.4 Discrete-Time Representation

For practical implementation, the continuous-time dynamics are discretized to enable numerical simulations and real-time control computations. In this work, the discretization uses a sampling interval of 100 ms ($\Delta t = 0.1$ seconds).

To update the robot's position and orientation over each time step, the `ode45` solver in MATLAB is employed in the simulation. This solver utilizes an adaptive Runge-Kutta method of orders 4 and 5 to accurately approximate the continuous-time dynamics. In contrast, the Forward Euler method is used within the controller for simplicity and efficiency. The adaptive nature of the `ode45` solver ensures that the numerical error in the simulation remains within a specified tolerance by dynamically adjusting the step size as needed.

The solver integrates the following differential equations governing the robot's kinematics:

$$\dot{x} = v\cos(\theta), \quad \dot{y} = v\sin(\theta), \quad \dot{\theta} = \omega, \tag{1.12}$$

where v and ω are the robot's linear and angular velocities, respectively. By numerically solving these equations, `ode45` produces a discrete-time approximation of the robot's trajectory. This approach effectively handles the nonlinearities in the kinematic model, ensuring precise updates to the robot's states. The choice of Euler's method, despite its simplicity compared to more sophisticated approaches like the Runge-Kutta method, aligns with the demands of real-time implementation in non-linear MPC. Euler's method uses a single derivative evaluation per step:

$$y_{n+1} = y_n + \Delta t \cdot f(t_n, y_n). \tag{1.13}$$

While this method may introduce numerical errors for larger time steps, its computational efficiency allows for faster updates, making it suitable for real-time control. The use of Euler's method enables us to maintain a relatively short prediction horizon N_h, which reduces the computational burden and allows the controller to react quickly to changes in the system's state. This is particularly advantageous in dynamic, non-linear systems where computational delays can degrade performance.

The tuning of the time step Δt is also critical; a smaller Δt improves accuracy and ensures stability of the predictions, but increases computational demand. Conversely, a larger Δt reduces computation time but can lead to errors in approximating the system dynamics. Balancing Δt ensures efficient computation while maintaining acceptable accuracy in predicting system behavior over the horizon.

In contrast, the Runge-Kutta methods used in `ode45` evaluate the derivative function multiple times per step, using intermediate points to improve the estimate of the next state:

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \tag{1.14}$$

where k_1, k_2, k_3, k_4 are weighted evaluations of the derivative at various points within the interval. This higher-order accuracy ensures that the numerical error per step is minimized, making it particularly suitable for systems with nonlinear dynamics. Additionally, `ode45` adaptively adjusts the step size based on the estimated error, ensuring computational efficiency without sacrificing precision.

Using the Runge-Kutta method strikes a balance between computational cost and the accuracy required for simulating the robot's kinematics, making it the preferred choice for this application. In contrast, the Forward Euler method is employed in the projection of the controller, as it is computationally efficient and suitable for scenarios where new control actions are continuously computed for each position. This approach leverages the frequent updates of the control inputs to maintain stability and performance despite the simpler numerical integration method.

1.5 Summary

This chapter has outlined the derivation of the kinematic model for a differential drive robot, detailing the roles of linear and angular velocities in determining the robot's motion. The chapter has also discussed control constraints and the performance metric used to evaluate tracking accuracy and real-time responsiveness. This model provides the foundation for designing controllers that can accurately drive the robot to desired positions and follow paths under practical constraints.

Chapter 2

Questions 1 and 2 - Controller Design for Position Tracking

In this chapter, we address the design of the controller for position tracking in the context of a mobile robot with a differential drive system. The objective of this section is to implement and evaluate controllers that guide the robot to follow a reference path or position. We will break this down into two parts corresponding to the first two questions of the assignment, focusing on achieving a target position and tracking a parametric curve.

2.1 Problem Setup

The system we are considering is a differential drive mobile robot, where the motion is determined by the robot's linear velocity v and angular velocity ω. The robot's state is defined by its position x, y, and orientation θ, while the control inputs are the linear velocity v and angular velocity ω, which influence the robot's motion.

The robot dynamics can be modeled as follows: the rate of change of the position (x, y) and orientation θ are given by:

$$\dot{x} = v\cos(\theta), \quad \dot{y} = v\sin(\theta), \quad \dot{\theta} = \omega, \tag{2.1}$$

where v is the linear velocity, defined as the average of the right and left wheel velocities:

$$v = \frac{1}{2}(v_r + v_l), \tag{2.2}$$

and ω is the angular velocity, which relates the difference in the wheel velocities to the robot's wheelbase b:

$$\omega = \frac{v_r - v_l}{b}, \tag{2.3}$$

where v_r and v_l are the velocities of the right and left wheels, respectively, and b is the distance between the two wheels (the robot's wheelbase). This model governs the robot's motion in terms of the control inputs, which we aim to optimize through the controller design.

When designing the controller, it is crucial to account for the robot's physical constraints to ensure safe and feasible operation. These constraints are applied to the control inputs, specifically the linear velocity v and angular velocity ω, as well as their rates of change, to avoid erratic or unsafe behavior.

The robot's linear velocity is constrained by a maximum allowable speed $v_{\max}$. Additionally, to prevent sudden changes in speed or direction, both linear and angular accelerations are bounded by maximum allowable values, $a_{\max}$ and $\alpha_{\max}$, respectively. These acceleration constraints ensure that the robot's motion remains smooth and within safe operational limits. Specifically, we require that:

$$-v_{\max} \leq v \leq v_{\max}; \tag{2.4}$$

$$-a_{\max} \leq \dot{v} \leq a_{\max}; \tag{2.5}$$

By respecting these constraints, the controller ensures that the robot's motion stays within the feasible operating region, preventing issues such as excessive speed or abrupt changes in direction that could affect performance or safety.

The goal of the first question is to design a controller that drives the robot from its initial position at the origin $(x, y, \theta) = (0, 0, 0)$ to the target position $(10, 10)$. The second question extends this task by requiring the robot to track a parametric curve over a specified time interval.

2.2 MPC Controller for Position Tracking

For both tasks, we use Model Predictive Control (MPC) to compute the optimal control inputs that guide the robot. The objective of MPC is to minimize the cost function over a prediction horizon while satisfying system dynamics and input constraints. The key feature of MPC is that it uses a receding horizon approach, solving an optimization problem at each time step, considering future states and inputs.

In this section, we discuss the formulation and implementation of Model Predictive Control (MPC) for position tracking of the mobile robot. MPC is a control strategy that optimizes control inputs over a finite horizon while respecting system dynamics and constraints. It is particularly well-suited for systems with constraints and is widely used for trajectory tracking tasks.

Let the robot's state at time k be represented by $x_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix}$, where x_k, y_k, and θ_k represent the robot's position and orientation at time k. The control inputs are the linear velocities of the left and right wheels, v_l and v_r, and we define the control input vector at time k as $u_k = \begin{bmatrix} v_l \\ v_r \end{bmatrix}$.

The cost function for the MPC problem is given by:

$$J = \sum_{k=1}^{N_h} \left((x_k - x_{\text{ref},k})^T Q (x_k - x_{\text{ref},k}) + u_k^T R u_k \right), \tag{2.6}$$

where $x_{\text{ref},k}$ is the reference state, Q is the state cost matrix, and R is the input cost matrix. The prediction horizon is N_h, which defines how many future time steps the MPC will optimize over. The objective is to minimize the cost function, which balances tracking the reference trajectory and minimizing control effort.

The dynamics of the system are governed by the robot's kinematic model. Specifically, the robot's position x and y, and its orientation θ, evolve according to the following equations:

$$\dot{x}_k = \cos(\theta_k) v_k, \quad \dot{y}_k = \sin(\theta_k) v_k, \quad \dot{\theta}_k = \omega_k, \tag{2.7}$$

where v_k is the linear velocity, and ω_k is the angular velocity at time k. The control inputs v_k and ω_k are related to the wheel velocities $v_{r,k}$ and $v_{l,k}$ by:

$$v_k = \frac{1}{2}(v_{r,k} + v_{l,k}), \quad \omega_k = \frac{v_{r,k} - v_{l,k}}{b}. \tag{2.8}$$

Here, b is the robot's wheelbase, which defines the distance between the two wheels.

The robot's motion is subject to several constraints. The linear velocity is bounded by the robot's maximum allowable speed:

$$-v_{\max} \leq v_k \leq v_{\max}, \tag{2.9}$$

where $v_{\max}$ is the maximum linear velocity.

The angular velocity ω_k is not independently constrained but is instead limited by the maximum linear velocity $v_{\max}$. Specifically,

$$|\omega_k| \leq \frac{v_{\max}}{r}, \tag{2.10}$$

where r is a characteristic length, such as the robot's turning radius. This ensures that the angular velocity remains consistent with the physical limits imposed by the maximum linear velocity.

Furthermore, the linear acceleration is constrained to prevent abrupt changes in speed. The acceleration must remain within a maximum allowable value $a_{\max}$, such that:

$$-a_{\max} \leq \dot{v}_k \leq a_{\max}, \tag{2.11}$$

where $a_{\max}$ is the maximum linear acceleration. This constraint ensures smooth motion and avoids excessive forces on the robot's actuators.

To solve this MPC problem, an optimization approach is employed. At each time step, the optimization problem is solved to minimize the cost function, subject to the system's dynamics and constraints. The optimization is typically performed over a finite prediction horizon N_h, and the optimal control inputs are computed for the current state of the system. This process is repeated at each time step as new measurements are received, thus updating the trajectory and control inputs.

However, increasing or decreasing the prediction horizon, N_h, in an MPC has significant effects on the results. A longer horizon provides a more global view of the trajectory, potentially leading to smoother and more optimal solutions, as it considers the system's behavior further into the future. However, it also increases the computational complexity, potentially causing delays that could affect real-time performance. On the other hand, a shorter horizon reduces computational demand and can make the system more reactive, but it might lead to suboptimal or short-sighted decisions. Balancing N_h is critical to ensure efficiency and robustness in dynamic systems like robots.

The problem is solved using numerical optimization solvers such as **Non Linear Solver fmincon** in MATLAB. The solver uses the current state and the reference trajectory to generate the control inputs that drive the robot towards the desired path while respecting the constraints on velocities and accelerations.

2.2.1 Task 1: Reaching a Target Position

In the first task, the objective is to drive the robot to the fixed position at $(10, 10)$. To solve this, we use Model Predictive Control (MPC) to compute the necessary wheel velocities at each time step. The controller takes into account the robot's current state $x_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix}$ and the target position $(10, 10)$. At each step, the MPC solves an optimization problem over a short prediction horizon, generating the optimal linear and angular velocities v_k and ω_k, that drive the robot towards the target.

The robot's dynamics within the controller are approximated using the Forward Euler method to calculate the next state $x(k+1)$. Specifically, the update is given by:

$$x(k+1) = x(k) + T_s \begin{bmatrix} \cos(x(3)) & 0 \\ \sin(x(3)) & 0 \\ 0 & 1 \end{bmatrix} u_k, \tag{2.12}$$

where $x(k)$ is the state at the current time step, T_s is the sampling time, and $u_k = [v_k, \omega_k]^T$ is the control input at time step k. This method provides an efficient way to estimate the next state for the robot based on the current state and control inputs.

The optimization problem is solved at each step using the current state and the target position, generating the optimal control inputs that guide the robot to the target. The robot's trajectory is then updated, and the next optimization step is solved until the target position is reached.

The results show that the MPC controller successfully drives the robot to the target position, minimizing the trajectory error while respecting the velocity and acceleration constraints.

2.2.2 Task 2: Tracking a Parametric Curve

In the second task, the robot is required to track a parametric curve, described by the equations for $x(t)$ and $y(t)$, over a time interval of $0 \leq t \leq 20\pi$. Unlike the first task, where the target position was fixed, this task requires the robot to follow a continuously evolving reference trajectory, adding complexity to the problem.

The MPC controller for this task computes the optimal velocities at each time step by considering the robot's current state and the reference trajectory. The reference trajectory is parameterized by time, meaning the desired positions $x(t)$ and $y(t)$ evolve over time. The robot must continuously adjust its velocities to stay on track.

The optimization problem at each step minimizes the cost function J, which balances the tracking error and the control effort. The cost function is given by:

$$J = \sum_{k=1}^{N_h} \left((x_k - x_{\text{ref},k})^T Q (x_k - x_{\text{ref},k}) + u_k^T R u_k \right), \tag{2.13}$$

where $x_{\text{ref},k}$ is the reference state (target position), Q is the state cost matrix, R is the input cost matrix, and N_h is the prediction horizon. This formulation ensures that the robot minimizes both the deviation from the desired trajectory and the control effort over the prediction horizon, while respecting the system constraints.

The optimization problem is solved at each time step, and the optimal control inputs are computed to guide the robot along the reference curve. The results demonstrate that the MPC controller effectively tracks the dynamic reference trajectory, with small deviations typical due to model inaccuracies and discretization errors. The velocity and acceleration profiles remain smooth, ensuring that the robot follows the curve without violating the system's physical limits.

The trajectory error is evaluated at each time step, and the results show that the MPC controller successfully tracks the curve with an acceptable level of deviation, demonstrating its capability to handle time-varying reference trajectories.

2.3 Simulations and Results

In this section, we present the results of the two tasks: reaching a target position and tracking a parametric curve. The simulations were carried out using a Model Predictive Control (MPC) algorithm, which optimizes the robot's wheel velocities v_l and v_r at each time step to minimize the deviation from the reference trajectory. A time step of $\Delta t = 0.1\,\text{s}$ and a prediction horizon of $N_h = 10$ were used for both tasks.

To reduce the complexity of the optimization problem, constraints were imposed on the system. These included the relationship between the linear and angular velocities of the left and right wheels, $v = \frac{v_l + v_r}{2}$, and the angular velocity constraint, $\omega = \frac{v_r - v_l}{L}$, where L is the distance between the wheels. Additionally, bounds on the velocities v_l and v_r were enforced to ensure feasible operation within the robot's physical limits. In addition to these, other restrictions allowed for a more efficient solution while maintaining the desired trajectory and control accuracy.

It was also considered that the angular acceleration, and by extension the angular velocity, could be zero when the robot is traveling in a straight line. This condition arises naturally from the kinematics of differential drive robots: when the linear velocity of both wheels is equal

($v_l = v_r$), the robot experiences no angular velocity ($\omega = 0$) and follows a linear trajectory. In such cases, the absence of angular acceleration ($\dot{\omega} = 0$) ensures that the robot's motion remains constrained to a straight path, consistent with the principle of conservation of linear momentum in the absence of rotational forces.

Additionally, constraints were imposed to limit the change in the physical position of the wheels between consecutive time steps. This is essential to ensure that the robot's motion adheres to the principles of continuity and respects its mechanical limitations. Sudden and unrealistic changes in wheel positions, such as the robot appearing to rotate 180º in 0.01 seconds, would imply infinite velocities or accelerations, which are not physically realizable.

These constraints ensure that the robot's wheels cannot "teleport" or undergo instantaneous positional changes, as such behavior would violate the fundamental laws of classical mechanics, including Newton's laws of motion. Specifically, the angular displacement of the wheels over a time step is limited by the maximum achievable velocity and acceleration, ensuring smooth and continuous motion. By enforcing these restrictions, the model maintains consistency with the robot's dynamic and kinematic properties, respecting the limitations of its actuators and the frictional interactions with the ground. This ensures that the trajectories generated by the controller remain both physically realistic and feasible to execute, avoiding situations that could compromise the accuracy or stability of the system.

For the first task, the robot started at the initial position $(0, 0)$ and was required to navigate toward a fixed target at $(10, 10)$. The objective was for the robot to reach this point while maintaining smooth motion and avoiding abrupt changes in velocity or acceleration. The MPC controller computed the necessary wheel velocities at each time step, considering the current state of the robot and the desired position.

The trajectory of the robot is shown in the phase plot in Figure 2.1, where the blue line represents the robot's actual path. From the plot, we can observe that the robot successfully converges to the target with minimal deviation from the reference path, demonstrating the effectiveness of the MPC in achieving the task. As expected, the robot gradually approaches the target, with the error becoming minimal by the end of the simulation.

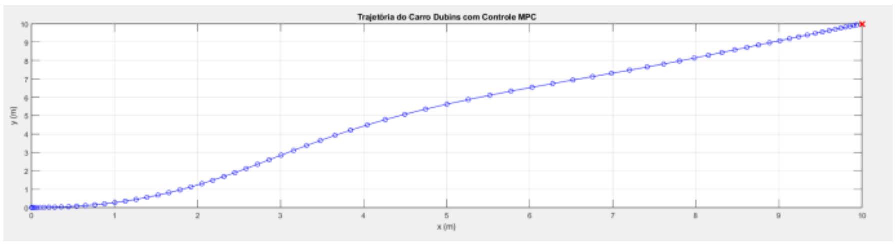


Figure 2.1: Phase plot of the robot's trajectory for Task 1. The blue line represents the robot's path.

In addition to the phase plot, we also evaluate the velocity and acceleration profiles during the execution of the first task. Figure 2.2 presents the linear velocity (v) and angular velocity (ω) over time, alongside the corresponding linear and angular accelerations. These profiles show that the MPC controller ensures smooth and gradual changes in the robot's velocity and acceleration, preventing abrupt motions and controlling the robot efficiently as it moves toward the target.

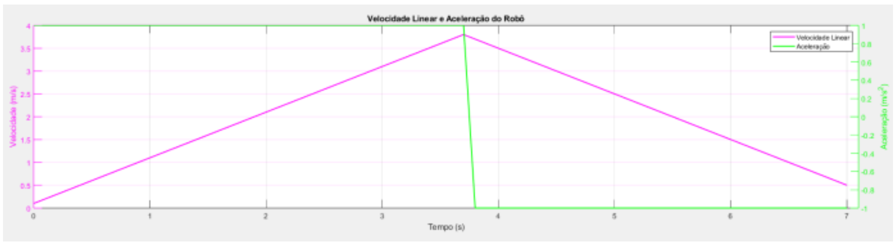


Figure 2.2: Linear velocities and accelerations of the JetBot over time for Task 1.

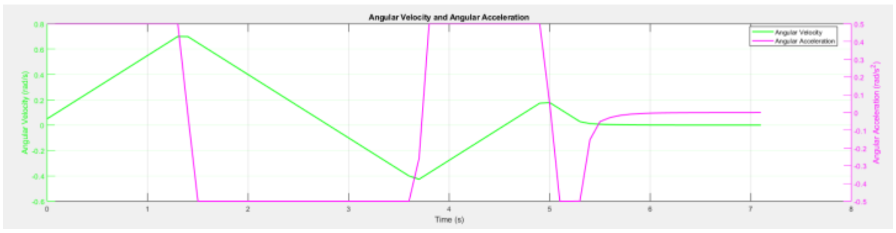


Figure 2.3: Angular velocities and accelerations over time for Task 1.

The individual velocities of the robot's wheels during the task are shown in Figure 2.4. This plot demonstrates how the left and right wheel velocities (v_l and v_r) evolve over time to enable the robot to navigate smoothly to the target. The MPC effectively balances these velocities while respecting the physical constraints of the robot.

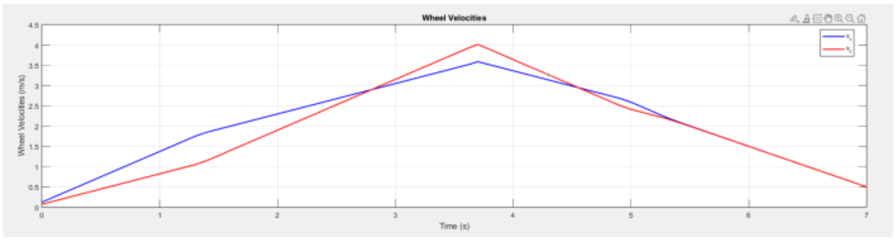


Figure 2.4: Wheel velocities (v_l and v_r) over time for Task 1.

It can be seen in the figures above that the speeds are within the limits and respect the defined restrictions.

For the second task, the robot was tasked with tracking a parametric curve defined by the equations:

$$x(t) = 5\sin\left(\frac{t}{5}\right), \quad y(t) = 5\cos\left(\frac{t}{10}\right). \tag{2.14}$$

This curve represents a dynamic reference path that the robot must follow, which is more complex than simply reaching a fixed target. The MPC controller computes the optimal control inputs at each time step to minimize the deviation from this reference trajectory.

The phase plot for Task 2 is shown in Figure 2.5, where the red line represents the reference trajectory, and the blue dots represent the robot's actual path. The plot clearly shows that the robot closely follows the reference trajectory, with small deviations that are expected in real-world systems due to modeling errors and discretization effects. Despite these small deviations, the MPC controller effectively guides the robot along the desired path.

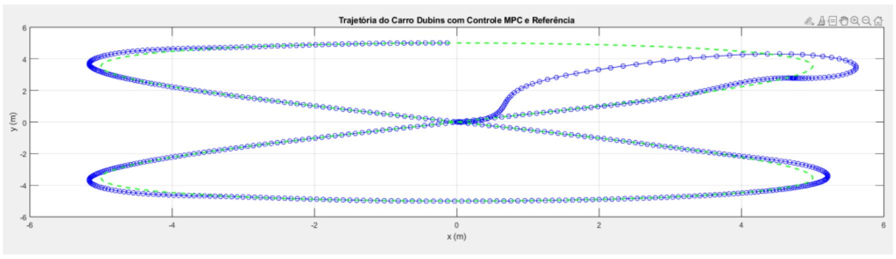


Figure 2.5: Phase plot of the robot's trajectory for Task 2. The red line represents the reference trajectory, and the blue dots represent the robot's actual path.

To further evaluate the robot's performance, we examine the linear and angular velocities and accelerations over time during Task 2. Figure 2.6 shows these profiles, highlighting the smoothness and efficiency of the motion. The linear and angular velocities are well-controlled, with the accelerations also remaining smooth. These plots demonstrate that the MPC controller handles the continuous adjustments required for tracking the dynamic reference trajectory effectively.

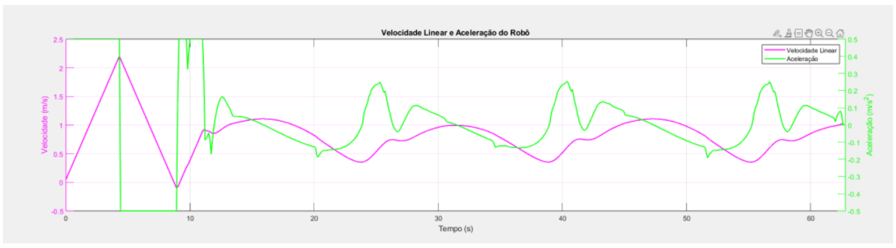


Figure 2.6: Linear velocities and accelerations of the JetBot over time for Task 2.

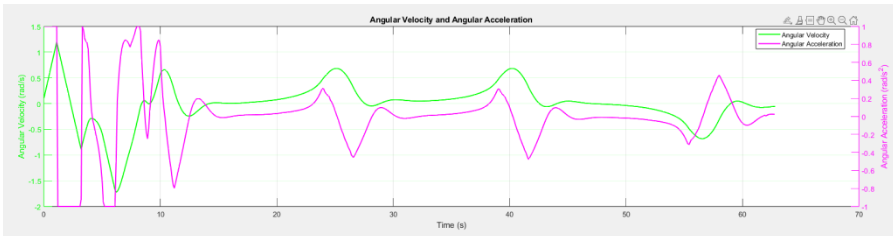


Figure 2.7: Angular velocities and accelerations over time for Task 2.

Additionally, the individual velocities of the robot's wheels during Task 2 are shown in Figure 2.8. This plot illustrates how the left and right wheel velocities (v_l and v_r) evolve over time as the robot tracks the dynamic reference trajectory. The MPC controller ensures that the wheel velocities remain within feasible limits, enabling smooth and accurate motion.

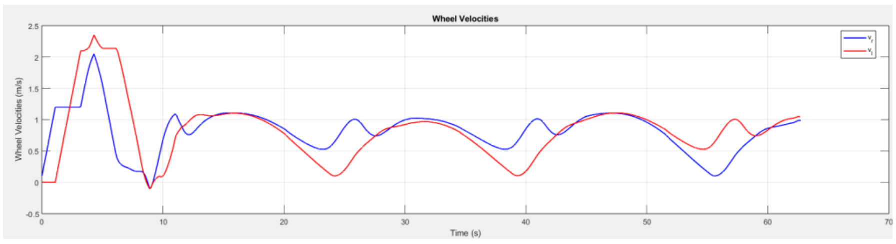


Figure 2.8: Wheel velocities (v_l and v_r) over time for Task 2.

The performance of the MPC controller in both tasks is assessed through the cost function, which penalizes deviations from the reference trajectory and excessive control efforts. The optimization problem is solved at each time step, and the optimal control inputs are computed to minimize the cost function. The results demonstrate that the MPC controller performs efficiently, ensuring that the robot follows the reference trajectories while respecting the system's constraints on velocity and acceleration. Notably, increasing the prediction horizon allows the robot to better execute curves, improving its ability to follow complex trajectories.

In summary, the simulation results show that the MPC controller successfully guided the robot in two tasks: reaching a fixed target position and tracking a dynamic parametric curve. The results highlight the effectiveness of the controller in providing smooth and efficient motion, with the robot adhering to the desired trajectories while minimizing control effort. The phase plots and velocity/acceleration profiles offer a comprehensive view of the robot's performance across both tasks.

Chapter 3

Question 3 - Advanced Multi-Robot Trajectory Coordination with Intersection Avoidance

Coordinating multiple autonomous robots in dynamic environments is a challenging yet essential task for applications such as warehouse automation, search and rescue, and collaborative exploration. In such scenarios, robots must follow predefined trajectories while avoiding collisions with each other and navigating around dynamic obstacles. This chapter addresses the problem of two autonomous robots tasked with following reference paths that intersect, while also tracking a moving target.

The motion of the robots is governed by differential drive dynamics, constrained by maximum linear velocity. To ensure safe and efficient operation, the system incorporates trajectory generation through interpolation, Model Predictive Control (MPC) for accurate path following, and an intersection avoidance algorithm that resolves potential collisions dynamically. This framework is validated using a series of simulations, highlighting both the theoretical contributions and practical applicability of the proposed method.

3.1 Model Predictive Control for Trajectory Tracking

The foundation of this approach lies in three interconnected components: trajectory generation, MPC-based trajectory tracking, and a novel dynamic intersection avoidance algorithm.

Trajectories for both robots are generated as sequences of waypoints, with each waypoint defined by its position (x, y) and orientation θ. To ensure smooth motion, interpolation is performed between consecutive waypoints. The spacing between interpolated points is determined by the sampling time T_s and the maximum allowable velocity $v_{\max}$. This guarantees that the trajectory respects the robot's kinematic constraints and provides a feasible reference for MPC.

Mathematically, the distance between consecutive interpolated points is calculated as:

$$d = v_{\max} T_s. \tag{3.1}$$

This ensures that the robots can transition smoothly between waypoints without abrupt changes in velocity or orientation.

Each robot employs an MPC controller to follow its assigned trajectory. The MPC minimizes a cost function that penalizes deviations from the reference trajectory and large control inputs. This optimization problem is constrained by the robot's differential drive dynamics:

$$\begin{aligned} \dot{x} &= v\cos(\theta), \\ \dot{y} &= v\sin(\theta), \\ \dot{\theta} &= \omega, \end{aligned} \tag{3.2}$$

where v is the linear velocity, and ω is the angular velocity. The MPC ensures that these velocities remain within predefined limits, thereby maintaining stability and avoiding unrealistic accelerations.

A key challenge in multi-robot coordination is managing intersections. When two robots approach a common waypoint, the system predicts their arrival times based on their velocities and positions. If the predicted times are too close, the system modifies one robot's trajectory to delay its arrival. This adjustment is performed dynamically, ensuring that the robots maintain a safe separation distance without deviating significantly from their original paths. The algorithm prioritizes the robot that reaches the intersection first, and the other robot is assigned a new reference point to avoid conflict.

3.2 Logic of the MATLAB Code

The MATLAB implementation for multi-robot trajectory coordination is designed to address various scenarios in which the paths of robots and a moving target may intersect. The system's primary goal is to ensure safe navigation while maintaining adaptability to dynamic environments. This section provides a detailed explanation of the code's logic, structured into specific cases and scenarios.

3.2.1 Case 1: Only Robot 1 Encounters an Intersection

In scenarios where only Robot 1 has an intersection on its path, the system evaluates three potential situations. The first situation occurs when Robot 1 reaches its predefined reference point. At this stage, the system checks the proximity of the moving target. If the target is nearby, Robot 1 transitions to following the target directly. This behavior shift requires the system to recalculate the intersections for Robot 2, ensuring that it can adjust its trajectory accordingly without creating conflicts. On the other hand, if the target is far from Robot 1's position, the robot remains stationary, waiting for the target to come within range. During this waiting period, Robot 2 continues to pursue the target, maintaining overall system functionality and avoiding interruptions.

The second situation occurs when Robot 1 encounters the target before reaching its designated intersection point. In this case, Robot 1 immediately adapts its behavior to follow the target directly. This adaptation alters the dynamics of the system, prompting a recalculation of intersections for Robot 2 to maintain safe and efficient coordination. The third situation arises when the target itself reaches the intersection before Robot 1. When this happens, the system dynamically recalculates new intersection points for both robots, ensuring they can continue to operate without collisions or unnecessary delays.

3.2.2 Case 2: Only Robot 2 Encounters an Intersection

A similar logic applies to cases where only Robot 2 has an intersection on its path. When Robot 2 reaches its selected reference point, it evaluates the distance to the target. If the target is close, Robot 2 adjusts its behavior to follow the target directly, and the system recalculates intersections for Robot 1 to adapt its trajectory. If the target is not nearby, Robot 2 pauses at its reference point and waits for the target to approach, while Robot 1 continues tracking the target. This ensures that both robots work in tandem to achieve their objectives without compromising safety.

If Robot 2 encounters the target on its way to the intersection, it dynamically adjusts its trajectory to follow the target. This adjustment triggers recalculations for Robot 1's intersections, allowing it to adapt to the new scenario seamlessly. Similarly, if the target reaches the intersection before Robot 2, the system recalculates new intersection points for both robots, enabling them to navigate safely and maintain effective coordination.

3.2.3 Case 3: Both Robots Encounter Intersections

When both robots encounter intersections, the system manages the increased complexity by dynamically prioritizing actions. Initially, each robot moves toward its respective reference point. If one robot encounters the target first, it begins following the target directly. At this point, the system reverts to managing the situation as though only one robot has an intersection, applying the logic described in the earlier cases.

If one robot encounters the target on its way to the intersection, the robot immediately adjusts its behavior to follow the target. This ensures that both robots maintain safe and efficient operation. Should the target reach one of the intersections before either of the robots, the system recalculates the affected robot's intersection point, allowing it to continue navigating without conflict or unnecessary delays. The dynamic adjustments in this scenario highlight the flexibility and robustness of the algorithm.

3.2.4 Case 4: Neither Robot Encounters an Intersection

In the simplest case, where neither robot encounters an intersection, the system operates in a straightforward manner. Both robots follow the target directly, without the need for recalculations or trajectory adjustments. This baseline behavior is effective and ensures smooth navigation when intersections are not present. The simplicity of this scenario contrasts with the complexity of the others, yet it demonstrates the adaptability of the system to operate under different conditions.

3.2.5 Generalization to Two-Dimensional Trajectories

A distinctive feature of the MATLAB implementation is its ability to handle arbitrary two-dimensional trajectories. The system is not constrained to specific orientations, such as horizontal or vertical paths, but instead dynamically computes waypoints and interpolates trajectories to accommodate any changes in direction or orientation. This flexibility is achieved through careful design of the trajectory generation and control algorithms, ensuring that the robots can smoothly adapt to varied scenarios.

This generalization enables the system to navigate complex environments, such as those found in warehouses, search and rescue operations, or collaborative exploration tasks. The ability to handle diverse trajectories significantly enhances the system's utility, making it applicable to a wide range of real-world problems. By leveraging this adaptability, the MATLAB implementation can coordinate multiple robots effectively, even in highly dynamic settings where targets move unpredictably, and obstacles are present.

The logic described in this section underscores the robustness of the MATLAB implementation. Through careful planning, dynamic recalculations, and adaptability to diverse trajectories, the system ensures that robots can navigate safely and efficiently under a variety of scenarios. Each case builds on the fundamental principles of collision avoidance and target tracking, demonstrating the versatility and practicality of the approach.

3.3 Simulation Results and Analysis

To evaluate the proposed framework, a series of simulations were conducted. These simulations demonstrate the robots' ability to track their trajectories, avoid intersections, and maintain dynamic stability. The results are presented and analyzed below.

The first simulation focuses on the positions of the two robots and the target over time. Figure 3.1 shows the trajectories of the robots as they follow their reference paths while tracking the target. The target moves along a predefined path, and the robots adapt their motion to stay within range. The figure highlights the robots' ability to synchronize their movement, even as they approach intersection points.

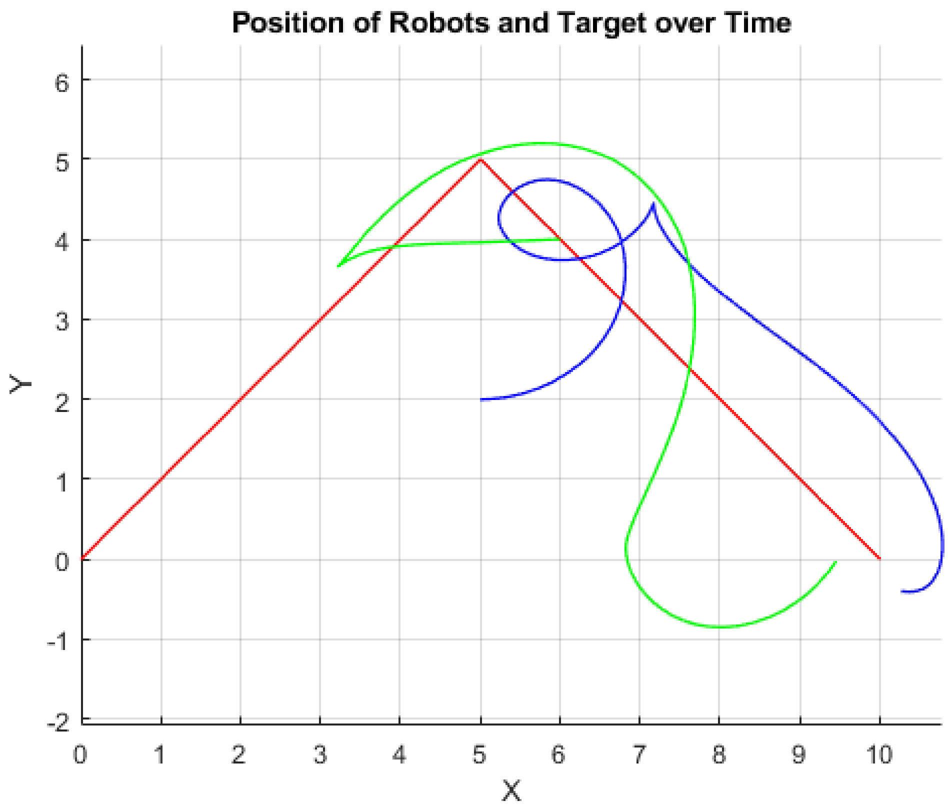


Figure 3.1: Positions of Robots and Target Over Time.

The robots dynamically adjust their trajectories to avoid intersections and track the moving target.

In this scenario, Robot 1 maintains a position closer to the target, while Robot 2 follows a secondary trajectory to avoid collisions. This behavior is a direct result of the intersection management algorithm, which prioritizes safety while maintaining tracking accuracy.

The next simulation evaluates the trajectory tracking performance of both robots. Figure 3.2 shows the trajectory of Robot 1, along with its linear velocity and acceleration profiles. The robot successfully follows its reference path, with smooth transitions in velocity and acceleration.

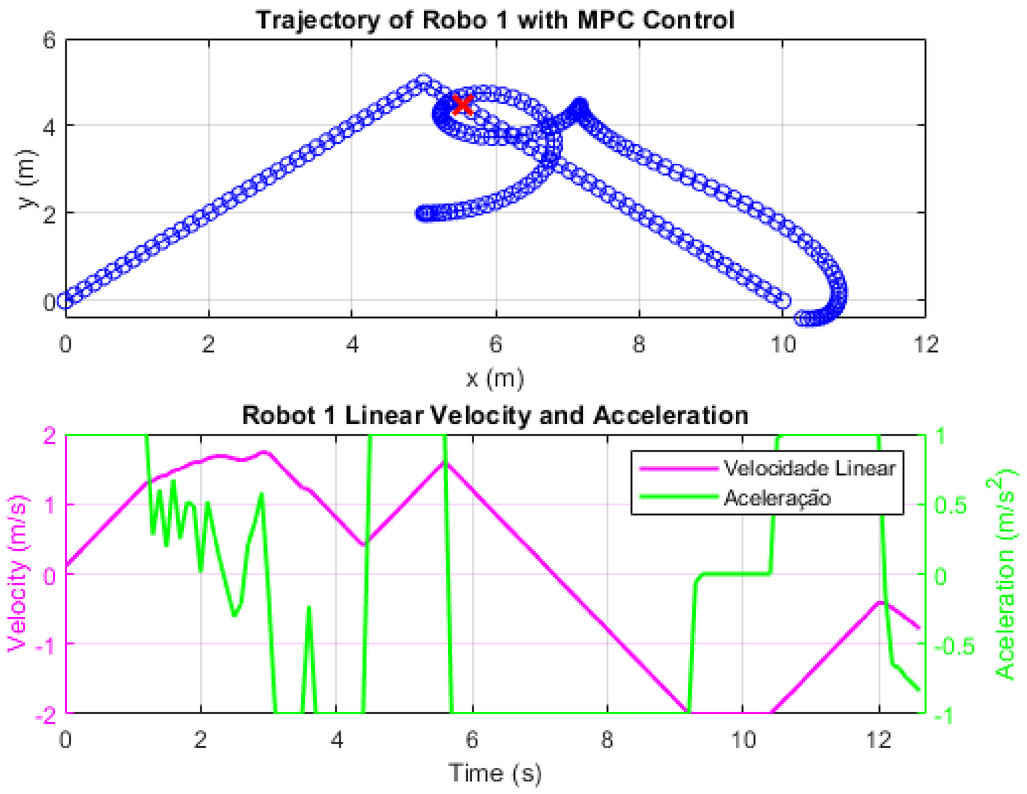


Figure 3.2: Trajectory of Robot 1 with MPC Control.

The robot exhibits smooth motion, adhering to its reference path while avoiding excessive accelerations.

Similarly, Figure 3.3 illustrates the trajectory of Robot 2. Despite modifications to its reference trajectory near the intersection, Robot 2 maintains smooth and stable motion, demonstrating the robustness of the control framework.

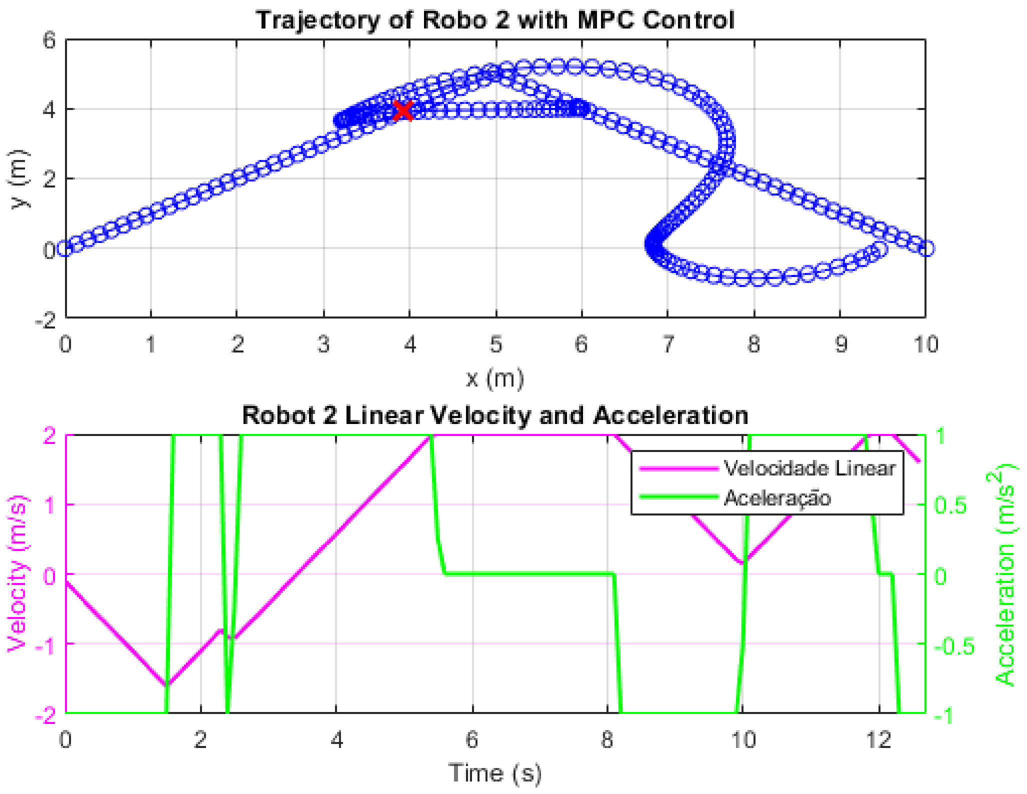


Figure 3.3: Trajectory of Robot 2 with MPC Control.

Adjustments to the trajectory near intersections are smoothly managed, preserving stability and tracking accuracy.

The performance of the MPC controller is further analyzed by examining the wheel velocities and angular dynamics of both robots. Figure 3.4 displays the wheel velocities, angular velocity, and angular acceleration of Robot 1. The results confirm that the control inputs remain within physical limits, ensuring smooth operation.

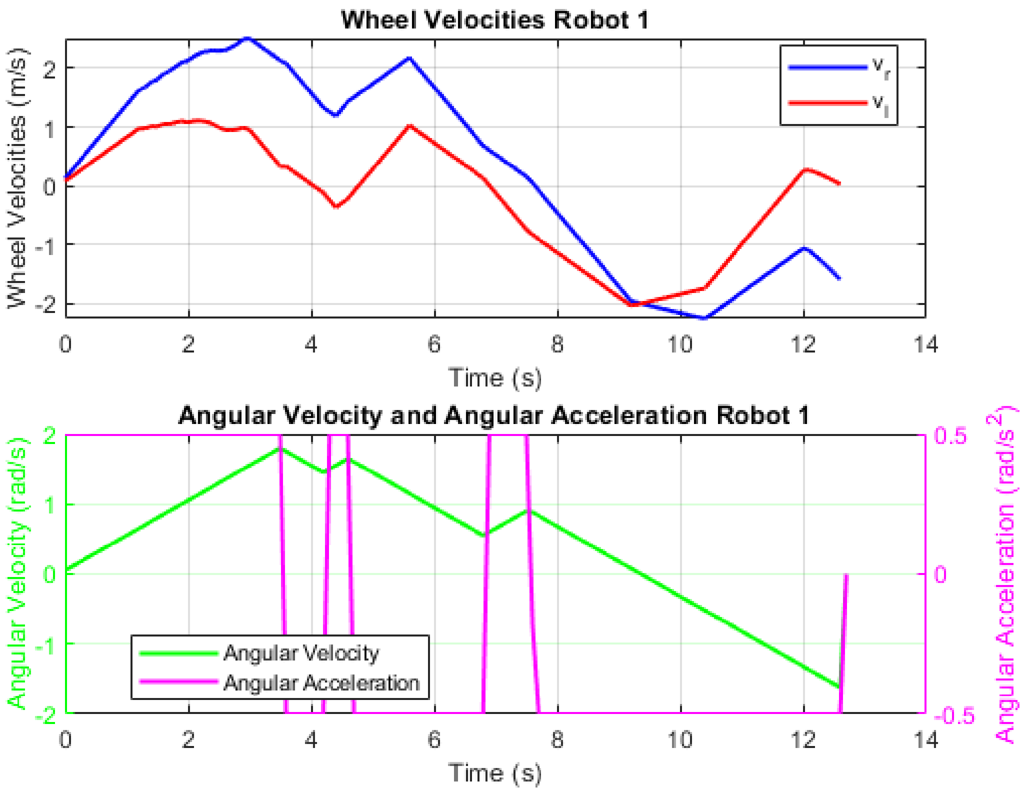


Figure 3.4: Wheel Velocities and Angular Dynamics for Robot 1.

The control inputs are well-regulated, indicating stability and feasibility.

For Robot 2, similar results are observed, as shown in Figure 3.5. The MPC effectively manages the robot's dynamics, even during trajectory adjustments near intersections.

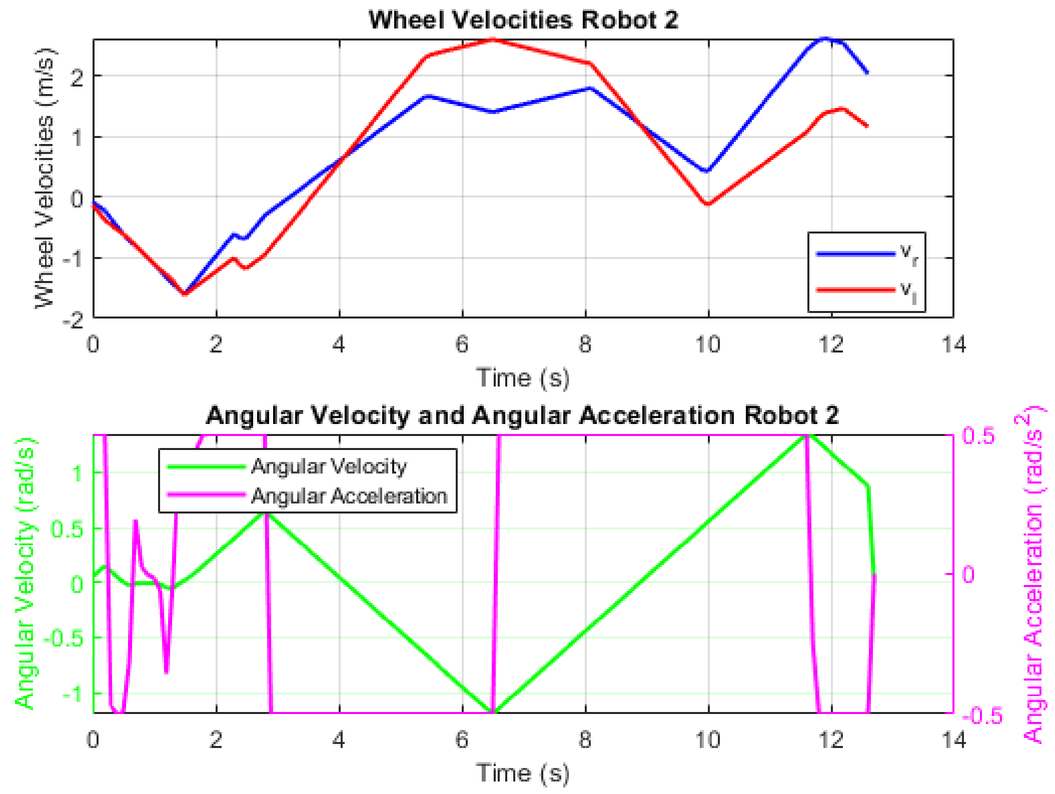


Figure 3.5: Wheel Velocities and Angular Dynamics for Robot 2.

The robot maintains smooth control inputs, highlighting the robustness of the framework.

3.4 Conclusions

This chapter presented a comprehensive framework for multi-robot trajectory coordination, incorporating trajectory generation, MPC-based tracking, and dynamic intersection avoidance. The simulations demonstrated that the proposed system is capable of achieving accurate trajectory tracking, collision-free navigation, and smooth dynamic behavior.

The results highlight the effectiveness of the intersection avoidance algorithm, which dynamically adjusts reference trajectories to prevent conflicts. This approach ensures safety without compromising performance. Future work could extend this framework to scenarios involving more robots, dynamic obstacles, and real-world implementation. Additionally, integrating more sophisticated prediction models could enhance the system's ability to handle highly dynamic environments.

Chapter 4

Question 4 - Circular Motion Control for Robot Drones with Target Tracking

This section explains the implementation and results of the `controllerMovie` function, designed to enable two robots to perform a coordinated circular filming maneuver around a moving target. The objective is for the robots to maintain a radius of 2 meters from the target, continuously orient their left wheels towards it (simulating a mounted camera), and complete one revolution every 40 seconds. The simulation is performed over a total duration of 80 seconds, allowing for two complete revolutions.

The primary goal is to ensure the robots follow circular trajectories around the target vehicle while adhering to constraints on maximum velocity and acceleration. Specifically, the robots are limited by a maximum linear velocity of $v_{\max} = 2\,\mathrm{m/s}$ and a maximum acceleration of $a_{\max} = 1\,\mathrm{m/s}^2$. The simulation is run with a sampling time of $T_s = 0.1$ seconds, which determines the frequency of control updates.

4.1 Reference Trajectory and Control Inputs

The reference trajectory for the robots is defined such that they follow circular paths around a moving center. The center of the circle moves along the horizontal axis with a velocity of $2\,\mathrm{m/s}$, and its position is given by:

$$P_{\text{centro}}(t) = \begin{bmatrix} 2t \\ 0 \end{bmatrix}. \tag{4.1}$$

Each robot must trace a circle with a radius of 2 meters around this moving center. The positions of the two robots are defined as:

$$P_1(t) = P_{\text{centro}}(t) + \begin{bmatrix} r\cos\theta(t) \\ r\sin\theta(t) \end{bmatrix}, \tag{4.2}$$

and

$$P_2(t) = P_{\text{centro}}(t) + \begin{bmatrix} r\cos(\theta(t)+\pi) \\ r\sin(\theta(t)+\pi) \end{bmatrix}. \tag{4.3}$$

Here, $\theta(t)$ is the time-dependent angle of each robot on the circle, and the term π in the second equation ensures that the two robots are 180° apart, positioning them on opposite sides of the circle.

The control system is implemented using a Nonlinear Model Predictive Control (NMPC) framework. The state of each robot is represented by $\mathbf{x} = [x, y, \theta]^\top$, where x and y represent the position, and θ is the orientation. The control inputs are $\mathbf{u} = [v, \omega]^\top$, with v denoting the linear velocity and ω the angular velocity. The dynamics of the system are described by:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \mathbf{u}. \tag{4.4}$$

The control inputs are constrained by the physical limitations of the robots. To complete one full revolution every 40 seconds, the angular velocity of the robots is fixed at $\omega = \frac{2\pi}{40} = \frac{\pi}{20}$ rad/s. Thus, the angle of each robot varies over time according to:

$$\theta(t) = \omega t = \frac{\pi}{20} t. \tag{4.5}$$

The linear velocity, v, is adjusted to maintain the robots' circular motion at a constant radius of 2 meters. The linear velocity is restricted by the maximum velocity constraint, $0 \leq v \leq v_{\max}$, where $v_{\max} = 2\,\text{m/s}$.

Additionally, the change in control inputs between time steps is limited to ensure smooth motion and to avoid sudden variations that could destabilize the system.

4.2 Simulation Results and Observations

The simulation results demonstrate the effectiveness of the NMPC controller. The figures below illustrate the performance metrics for both robots, including their trajectories, wheel velocities, and linear and angular velocity profiles.

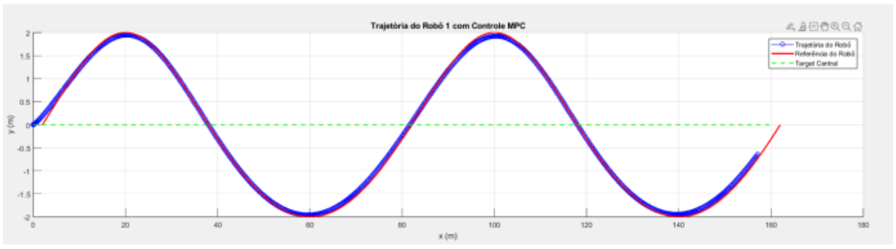


Figure 4.1: Trajectory of Robot 1 with MPC control.

Figure 4.1 shows the trajectory of Robot 1. The robot successfully maintains a circular path of 2 meters around the moving target while following its motion along the horizontal axis.

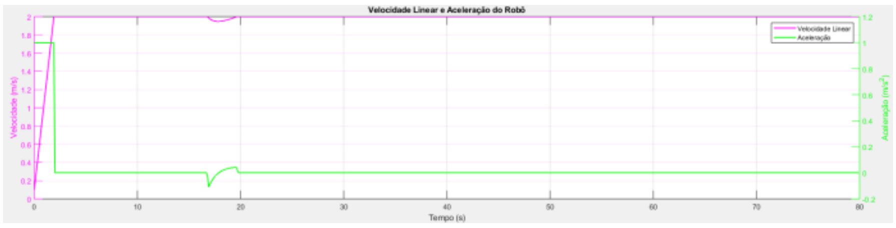


Figure 4.2: Linear velocity and acceleration of Robot 1.

Figure 4.2 presents the linear velocity and acceleration of Robot 1. The results confirm that the robot maintains smooth velocity profiles without violating the acceleration limits.

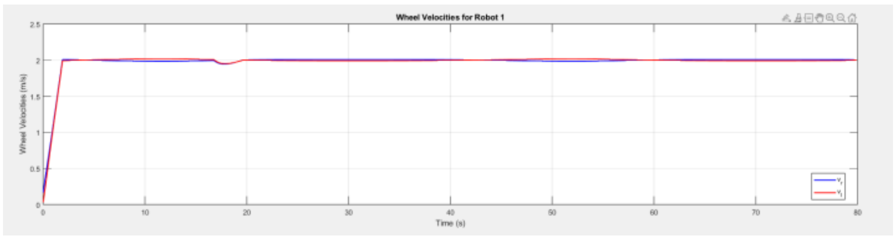


Figure 4.3: Wheel velocities of Robot 1.

Figure 4.3 shows the individual velocities of the left and right wheels of Robot 1. The MPC ensures that the velocities remain within the physical constraints while allowing for accurate trajectory tracking.

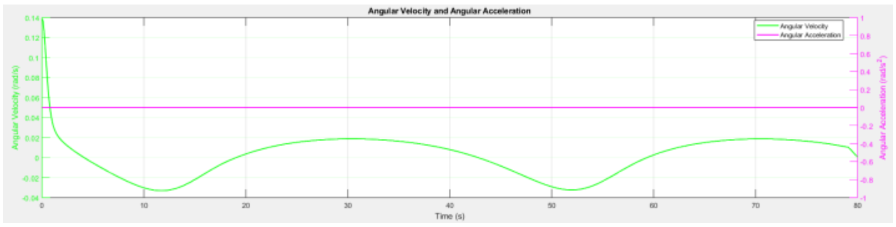


Figure 4.4: Angular velocity and acceleration of Robot 1.

Figure 4.4 illustrates the angular velocity and acceleration of Robot 1. The angular profiles confirm that the robot's orientation is continuously adjusted to maintain the desired alignment with the target.

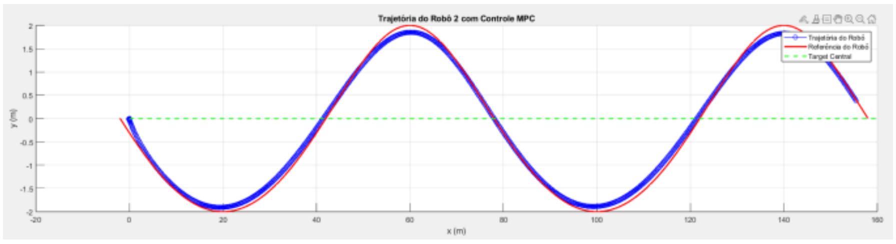


Figure 4.5: Trajectory of Robot 2 with MPC control.

Figure 4.5 shows the trajectory of Robot 2. Similar to Robot 1, Robot 2 maintains a circular path of 2 meters around the target and follows its motion accurately.

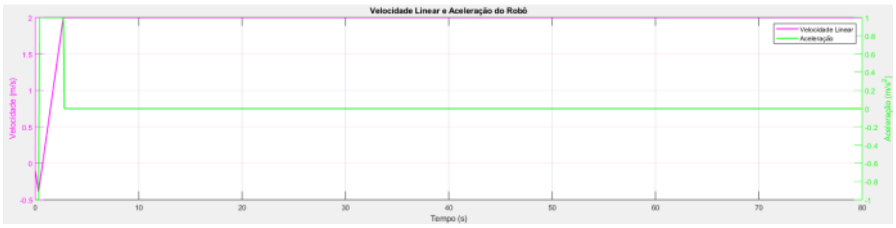


Figure 4.6: Linear velocity and acceleration of Robot 2.

Figure 4.6 presents the linear velocity and acceleration of Robot 2. The velocity profiles demonstrate smooth transitions, consistent with the imposed physical constraints.

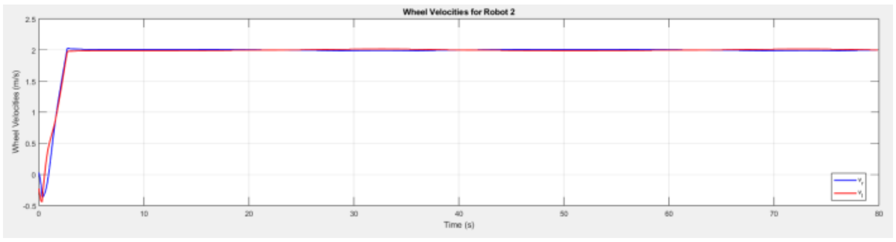


Figure 4.7: Wheel velocities of Robot 2.

Figure 4.7 shows the left and right wheel velocities for Robot 2. The velocities remain bounded and exhibit smooth adjustments over time.

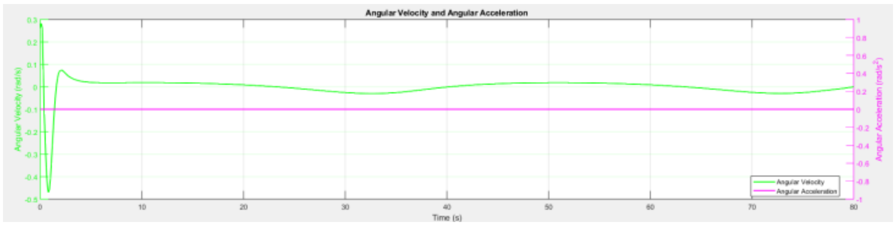


Figure 4.8: Angular velocity and acceleration of Robot 2.

Figure 4.8 illustrates the angular velocity and acceleration of Robot 2. These profiles confirm that the robot's orientation adjustments are smooth and adhere to the desired trajectory.

In conclusion, the proposed NMPC controller achieved the desired circular filming maneuver for both robots with high accuracy and compliance with real-time computational constraints. The trajectories were smooth, and the robots consistently oriented their cameras toward the target. The combination of NMPC and a well-structured cost function proved effective for this dynamic and constrained scenario. Future work could include incorporating disturbances or unknown target motions to further enhance the robustness of the control system.

Chapter 5

Part II - Background

5.1 Robot Operating System (ROS)

The Robot Operating System (ROS) is an open-source middleware framework designed to simplify the development of robotics applications. Despite its name, ROS is not an operating system but a collection of tools, libraries, and conventions that provide the infrastructure necessary for creating complex robotic behaviors. Its modular architecture allows developers to focus on high-level functionality without worrying about low-level details.

ROS offers several key features:

- **Hardware Abstraction**: ROS provides a standard interface for interacting with various hardware components, including sensors, actuators, and communication devices.
- **Communication Mechanisms**: ROS supports multiple communication paradigms, such as:
  - **Topics**: A publish-subscribe mechanism for asynchronous data exchange between nodes.
  - **Services**: A synchronous request-response mechanism for immediate data transactions.
  - **Actions**: A tool for managing long-duration tasks with real-time feedback and the ability to cancel operations.
- **Toolset**: ROS includes tools for visualization (e.g., RViz), simulation (e.g., Gazebo), and debugging, enabling efficient development and testing.

These features make ROS a versatile and widely adopted platform in both academia and industry, facilitating rapid prototyping and deployment of robotic systems.

5.2 Jetbot Platform

The NVIDIA Jetbot is a small-scale, open-source robotics platform built around the NVIDIA Jetson Nano, a compact edge AI computing device. It is designed to provide an accessible entry point for education, prototyping, and research, combining real-time AI processing capabilities with a user-friendly interface. This combination makes the Jetbot an excellent choice for implementing advanced robotics applications, including those requiring machine learning and computer vision.

At the heart of the Jetbot's capabilities lies the NVIDIA Jetson Nano, which brings powerful GPU-based computational resources to the platform. These resources allow the Jetbot to perform intensive tasks, such as object detection, real-time video processing, and other AI-driven functionalities, with remarkable efficiency. The ability to execute such tasks in real-time enhances its suitability for applications that demand immediate responses, such as autonomous navigation and obstacle avoidance.

Another critical advancement was the replacement of the YALMIP framework with CasADi for formulating and solving the optimization problem. CasADi, a versatile symbolic framework for optimization and automatic differentiation, was paired with the IPOPT solver to achieve faster and more efficient optimization. The IPOPT solver was configured with a tolerance level set to 10^{-10}, ensuring that the solutions obtained were not only computationally efficient but also met the stringent accuracy requirements of the system. This adjustment significantly reduced the computational overhead compared to the setup in Part I while maintaining the desired level of optimization precision.

The Jetbot's flexibility is another of its significant advantages. Its open-source design encourages users to customize both hardware and software, adapting the platform to various use cases and experimental setups. Whether integrating additional sensors or developing unique control algorithms, the Jetbot provides a robust foundation for innovation.

A particularly valuable feature of the Jetbot is its compatibility with ROS, which facilitates seamless integration of optimization-based control algorithms and communication with other robotic components. This compatibility enhances the Jetbot's capabilities, enabling it to operate effectively in complex robotic ecosystems.

In this project, the Jetbot is utilized within the robotics arena at FCT-UNL. The jetbot repository serves as a vital resource, offering a comprehensive guide for setting up the platform, including simulation tools and sample code for controlling the robot. By leveraging these resources, the project aims to implement an optimization-based controller that efficiently achieves the desired task, demonstrating the practical utility of the Jetbot in solving real-world robotics challenges.

Chapter 6

Implementation

6.1 Task Description

The selected task involves controlling the Jetbot to navigate towards a reference point at coordinates (10, 10) on a 2D plane. The robot employs a Model Predictive Control (MPC) framework, implemented using the CasADi library, to compute optimal control inputs that minimize tracking error while adhering to constraints on linear and angular velocities, as well as their respective accelerations.

The CasADi-based optimization model incorporates a dynamic state update using a Runge-Kutta integration method for accurate trajectory prediction over a finite time horizon. Control inputs are constrained to ensure the robot's physical feasibility, including bounds on linear speed, angular velocity, and their rates of change. The system is designed to achieve the reference target efficiently and precisely, demonstrating the efficacy of the control strategy.

During the simulation, the robot starts at an initial position of (0, 0) and iteratively computes its trajectory to approach the reference point. The simulation setup logs the robot's position, heading angle, linear velocity, and angular velocity at each time step. A visualization of the robot's trajectory and velocity profiles illustrates the robot's performance in converging to the desired target.

6.1.1 Numerical Optimization and Solver Improvements

In Part II of the project, the numerical optimization approach underwent several significant enhancements to improve the overall performance and reliability of the control system. One of the key changes was the adoption of the Runge-Kutta method of order 4 for the discretization process within the Model Predictive Control (MPC) framework. Unlike the simpler Euler discretization method used in Part I, the Runge-Kutta method provided a more accurate and stable numerical solution to the differential equations governing the system's dynamics. This improvement was crucial in scenarios requiring precise trajectory predictions over extended horizons, as the higher-order method allowed the controller to predict further into the future without compromising on accuracy. The stability offered by this approach also enabled the use of larger time steps between prediction horizons, while maintaining reliable control performance.

The values were set based on the limitations of the robot itself. This choice was made to provide the solver with a practical and reasonable starting point, which in turn expedited the convergence of the optimization process. The combination of these changes—higher-order discretization, an advanced solver framework, and thoughtfully chosen initial conditions—resulted in a more robust and efficient control system. These enhancements played a pivotal role in ensuring the Jetbots could adhere to their desired trajectories with improved responsiveness and computational efficiency, even in dynamically challenging scenarios. By leveraging these advanced techniques, the control implementation achieved a new level of precision and reliability, further cementing the benefits of optimization-based approaches in robotics control systems.

6.2 Integration of the controllerSimple with ROS

In the implementation with ROS, the code identifies two robots, Jetbot0 and Jetbot1, using the `diff_drive_controller` package. This package publishes the robots' odometry on the topic `/jetbot0/diff_drive_controller/odom` (type `nav_msgs/Odometry`) and subscribes to the topic `/jetbot0/diff_drive_controller/cmd_vel` (type `geometry_msgs/Twist`), which receives velocity commands for control.

The odometry information is extracted from `Odometry` messages sent by each robot's topics and used to compute the robot's state, represented by the position variables x, y, and orientation θ (in radians). These values are obtained from the odometry message and converted into a state vector, which is then used by the controller.

The controller, implemented in the `controllerSimple` function, is responsible for calculating the optimal linear and angular velocities for the robots, enabling them to reach a defined reference position. The calculated velocities are published to the topics `/jetbot0/diff_drive_controller/cmd_vel` and `/jetbot1/diff_drive_controller/cmd_vel`. The `diff_drive_controller` works with a `Twist` message, extracting the `linear.x` component for linear velocity and the `angular.z` component for angular velocity. The other components of the message are ignored.

The controller uses the CasADi library to solve a model predictive control (MPC) problem. The MPC solves an optimization problem, computing optimal commands for the robots while respecting constraints such as maximum linear and angular accelerations. The model of each robot considers:

- Linear velocity as the average of the right and left wheel velocities.
- Angular velocity as the difference between the wheel velocities divided by the distance between them.

Additionally, the controller applies the fourth-order Runge-Kutta (RK4) integration method to accurately update the robots' states over time. Constraints are enforced to ensure the computed commands respect the physical limits of the robots, such as maximum velocities and accelerations.

6.2.1 Simulation's Results

The results of the simulation based on this implementation are presented in Figure 6.1, which combines two subplots in a single figure. The first subplot shows the robot's position trajectory as it moves toward the target, with the x and y positions plotted on the same graph. The second subplot displays the linear and angular velocities over time. This figure validates the simpler controller's effectiveness, achieving the desired target with acceptable performance and compliance with velocity constraints.

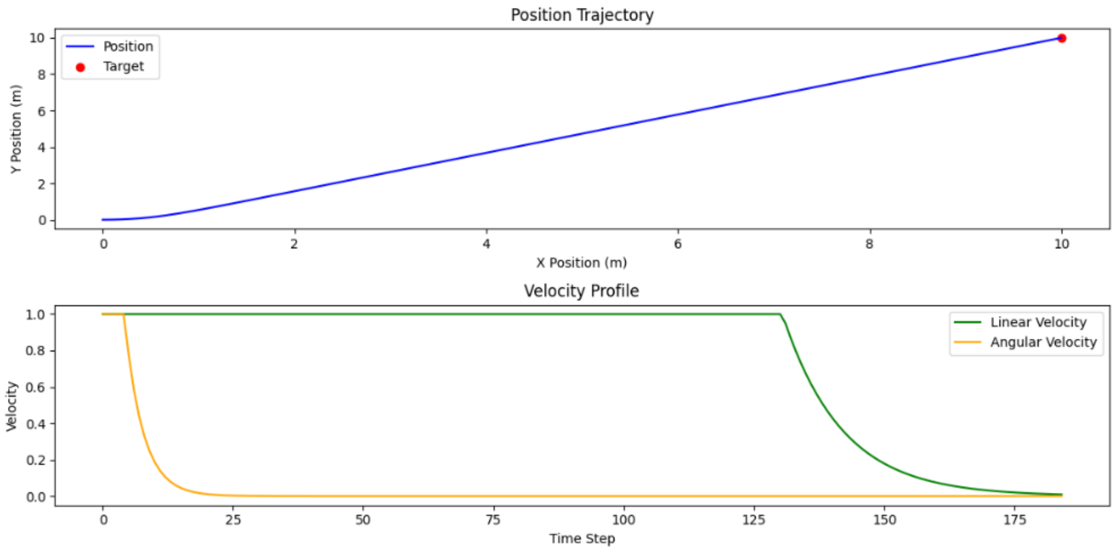


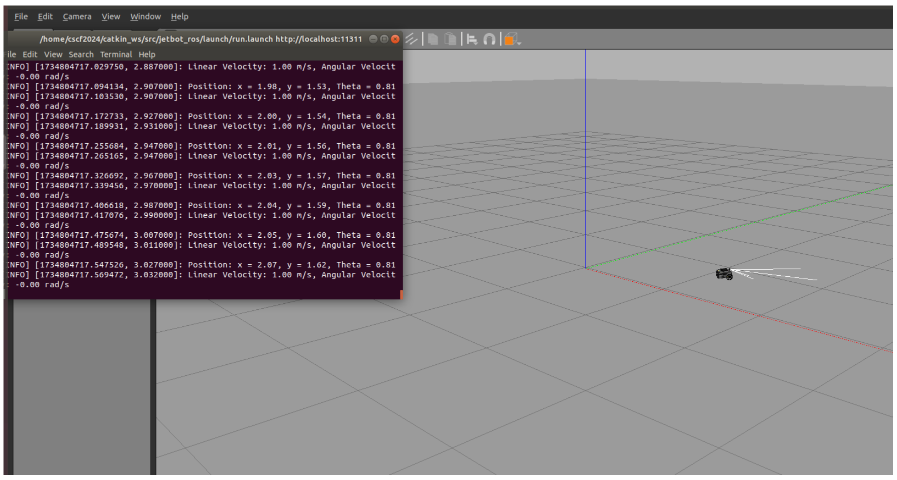
Figure 6.1: Position trajectory and velocity profile from the simpler simulation.

Overall, these results validate that the controllerSimple provides an efficient and reliable solution for robotic navigation within the ROS environment, ensuring both trajectory precision and compliance with dynamic constraints. This simulation further demonstrates that the controller is robust under the given settings, supporting potential future expansions and more complex scenarios.

6.2.2 ROS' Results (Gazebo and RViz)

The integration of the `controllerSimple` with the ROS ecosystem was validated using simulations in Gazebo and RViz. The goal was to observe the robot's behavior in a realistic environment as it navigates toward the target position of $(10, 10)$. Two key moments of the simulation are presented in Figures 6.2 and 6.3.

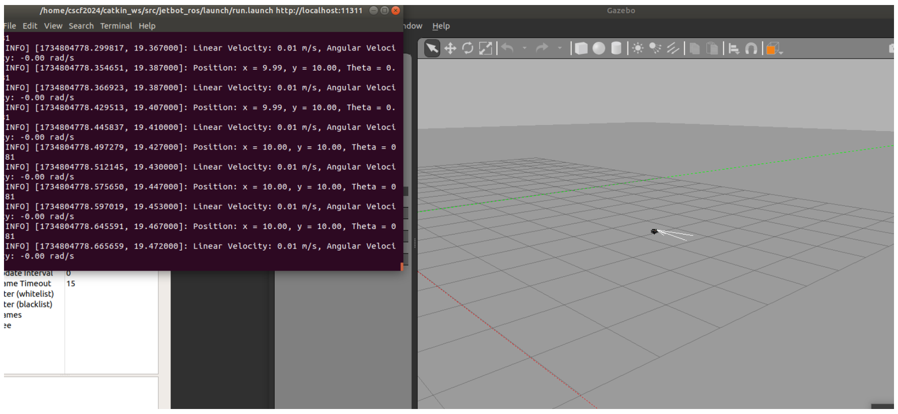
At the start of the simulation, as shown in Figure 6.2, the robot begins moving with an initial velocity of 1. This illustrates the controller's ability to generate smooth and compliant velocity commands, ensuring stable motion initiation.


File Edit Camera View Window Help
/home/cscf2024/catkin_ws/src/jetbot_ros/launch/run.launch http://localhost:11311
ile Edit View Search Terminal Help

```
INFO] [1734804717.029750, 2.887000]: Linear Velocity: 1.00 m/s, Angular Velocit
: -0.00 rad/s
INFO] [1734804717.094134, 2.907000]: Position: x = 1.98, y = 1.53, Theta = 0.81
INFO] [1734804717.103530, 2.907000]: Linear Velocity: 1.00 m/s, Angular Velocit
: -0.00 rad/s
INFO] [1734804717.172733, 2.927000]: Position: x = 2.00, y = 1.54, Theta = 0.81
INFO] [1734804717.189931, 2.931000]: Linear Velocity: 1.00 m/s, Angular Velocit
: -0.00 rad/s
INFO] [1734804717.255684, 2.947000]: Position: x = 2.01, y = 1.56, Theta = 0.81
INFO] [1734804717.265165, 2.947000]: Linear Velocity: 1.00 m/s, Angular Velocit
: -0.00 rad/s
INFO] [1734804717.326692, 2.967000]: Position: x = 2.03, y = 1.57, Theta = 0.81
INFO] [1734804717.339456, 2.970000]: Linear Velocity: 1.00 m/s, Angular Velocit
: -0.00 rad/s
INFO] [1734804717.406618, 2.987000]: Position: x = 2.04, y = 1.59, Theta = 0.81
INFO] [1734804717.417076, 2.990000]: Linear Velocity: 1.00 m/s, Angular Velocit
: -0.00 rad/s
INFO] [1734804717.475674, 3.007000]: Position: x = 2.05, y = 1.60, Theta = 0.81
INFO] [1734804717.489548, 3.011000]: Linear Velocity: 1.00 m/s, Angular Velocit
: -0.00 rad/s
INFO] [1734804717.547526, 3.027000]: Position: x = 2.07, y = 1.62, Theta = 0.81
INFO] [1734804717.569472, 3.032000]: Linear Velocity: 1.00 m/s, Angular Velocit
: -0.00 rad/s
```

Figure 6.2: Initial phase of the simulation with the robot starting at an initial velocity of 1.

By the end of the simulation, as shown in Figure 6.3, the robot successfully reaches the target position of $(10, 10)$, with its velocity reduced to 0. This confirms the controller's ability to effectively stop the robot at the desired location, meeting the final constraints while ensuring precision.



```
/home/cscf2024/catkin_ws/src/jetbot_ros/launch/run.launch http://localhost:11311
File  Edit  View  Search  Terminal  Help
1
INFO] [1734804778.299817, 19.367000]: Linear Velocity: 0.01 m/s, Angular Veloci
y: -0.00 rad/s
INFO] [1734804778.354651, 19.387000]: Position: x = 9.99, y = 10.00, Theta = 0.
1
INFO] [1734804778.366923, 19.387000]: Linear Velocity: 0.01 m/s, Angular Veloci
y: -0.00 rad/s
INFO] [1734804778.429513, 19.407000]: Position: x = 9.99, y = 10.00, Theta = 0.
1
INFO] [1734804778.445837, 19.410000]: Linear Velocity: 0.01 m/s, Angular Veloci
y: -0.00 rad/s
INFO] [1734804778.497279, 19.427000]: Position: x = 10.00, y = 10.00, Theta = 0
81
INFO] [1734804778.512145, 19.430000]: Linear Velocity: 0.01 m/s, Angular Veloci
y: -0.00 rad/s
INFO] [1734804778.575650, 19.447000]: Position: x = 10.00, y = 10.00, Theta = 0
81
INFO] [1734804778.597019, 19.453000]: Linear Velocity: 0.01 m/s, Angular Veloci
y: -0.00 rad/s
INFO] [1734804778.645591, 19.467000]: Position: x = 10.00, y = 10.00, Theta = 0
81
INFO] [1734804778.665659, 19.472000]: Linear Velocity: 0.01 m/s, Angular Veloci
y: -0.00 rad/s
```

Figure 6.3: Final phase of the simulation with the robot at the target position $(10, 10)$ and velocity reduced to 0.

These results highlight the robustness and precision of the implemented controller when tested in a realistic simulation environment. The robot's ability to achieve smooth acceleration and deceleration while maintaining accurate position tracking demonstrates the effectiveness of the `controllerSimple` in achieving its objectives within the ROS ecosystem.

6.3 Integration of the controllerMovie with ROS

The `controllerMovie`, which coordinates the two robots Jetbot0 and Jetbot1, is not yet fully ready to operate within the ROS environment. However, its design introduces significant improvements in coordinating multiple robots simultaneously.

For the `controllerMovie`, the control process is similar to that of the `controllerSimple`, with the main difference being the use of the `ApproximateTimeSynchronizer`. This synchronizer is used to combine messages from different topics approximately, ensuring that the robots' position messages are processed as if they were received at the same "time." The `ApproximateTimeSynchronizer` is particularly useful when messages have different latencies but still need to be used together for accurate calculations.

By utilizing the `ApproximateTimeSynchronizer`, the `controllerMovie` ensures that the robots' position messages are received with a minimal temporal difference, configured through the `slop` parameter. Once both messages are synchronized, the controller can calculate new actions based on the updated information accurately. This process is essential for optimizing the robots' behavior, enabling more efficient and coordinated navigation between them.

The `slop` parameter defines the maximum allowable difference between the publication times of messages for them to be considered synchronized. For example, if the `slop` is set to 0.1 seconds, the synchronizer will accept messages that have up to a 0.1-second difference in their publication times.

6.3.1 Simulation Results

To validate the potential of the `controllerMovie`, a set of simulations was conducted to analyze the behavior of both robots simultaneously. The results are presented through six key plots that provide insights into their dynamics and coordination.

Figure 6.4 shows the trajectories of both robots in the xy-plane. This phase plot highlights how the robots navigate toward their respective targets while avoiding collisions and maintaining coordinated movement. The overlapping trajectories indicate effective coordination and smooth navigation.

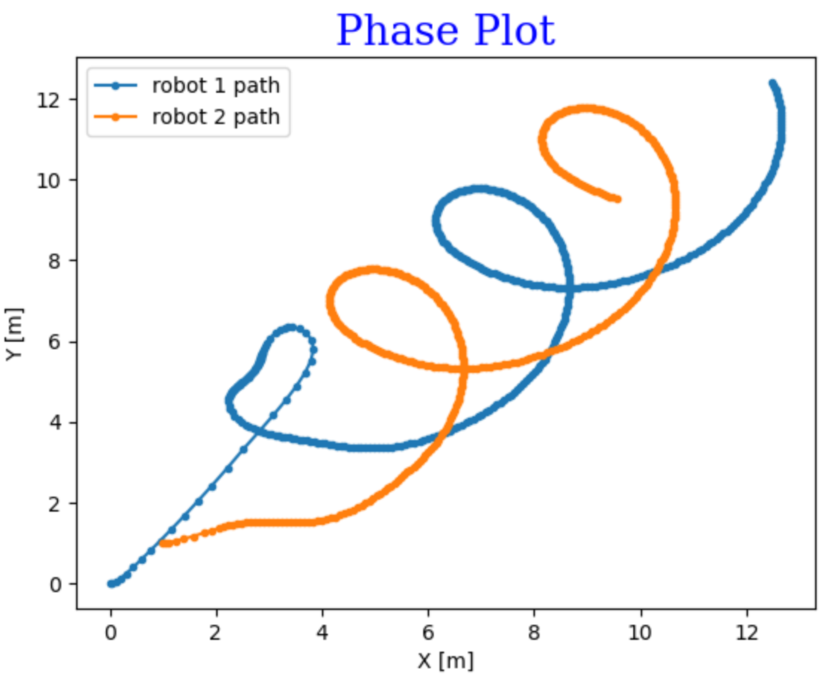


Figure 6.4: Phase plot showing the trajectories of both robots in the xy-plane.

The evolution of the x-coordinate of both robots over time is illustrated in Figure 6.5. This plot demonstrates how the robots adjust their x-positions dynamically to approach their targets. The smooth trajectories reflect consistent and stable behavior from the controller.

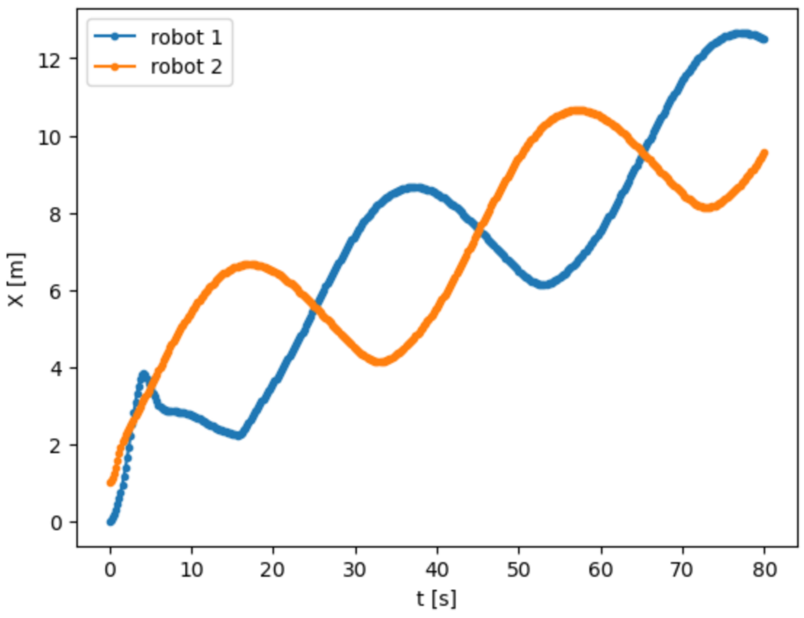


Figure 6.5: x-coordinate of both robots as a function of time.

Similarly, Figure 6.6 shows the y-coordinate evolution of both robots over time. Combined with the x-position data, this plot provides a comprehensive view of how the robots navigate the xy-plane effectively and with precision.

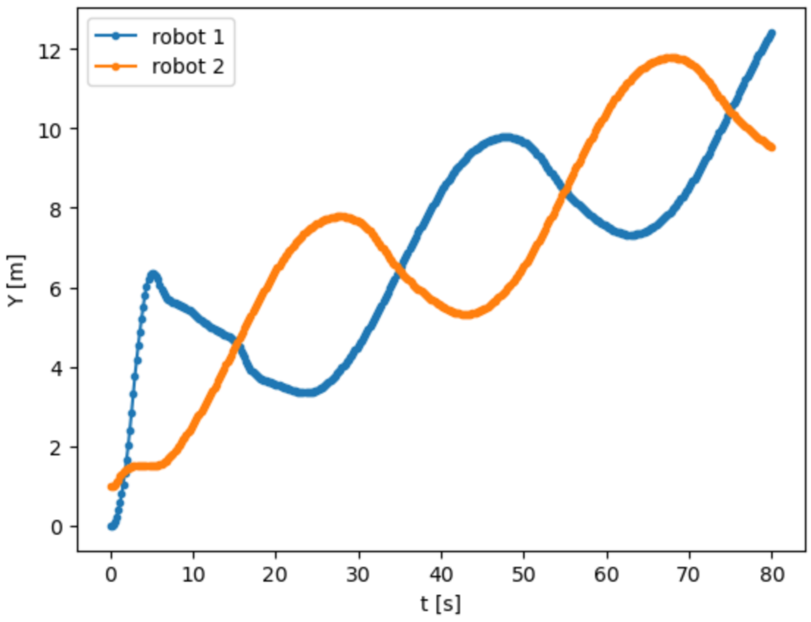


Figure 6.6: y-coordinate of both robots as a function of time.

The orientation (θ in radians) of both robots over time is depicted in Figure 6.7. This plot shows how each robot adjusts its heading direction to align with its target trajectory. The smooth changes in orientation indicate that the controller effectively manages the robots' rotational dynamics.

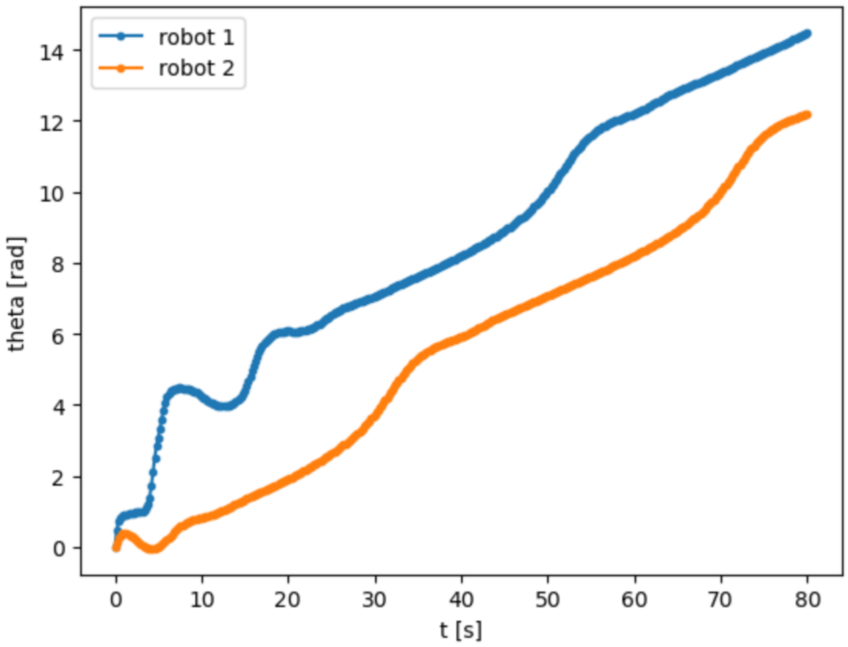


Figure 6.7: Orientation (θ) of both robots as a function of time.

The linear velocity of both robots over time is presented in Figure 6.8. This plot highlights how the robots accelerate and decelerate smoothly, adhering to the velocity constraints. The gradual changes in linear speed reflect the controller's capability to manage the robots' translational motion effectively.

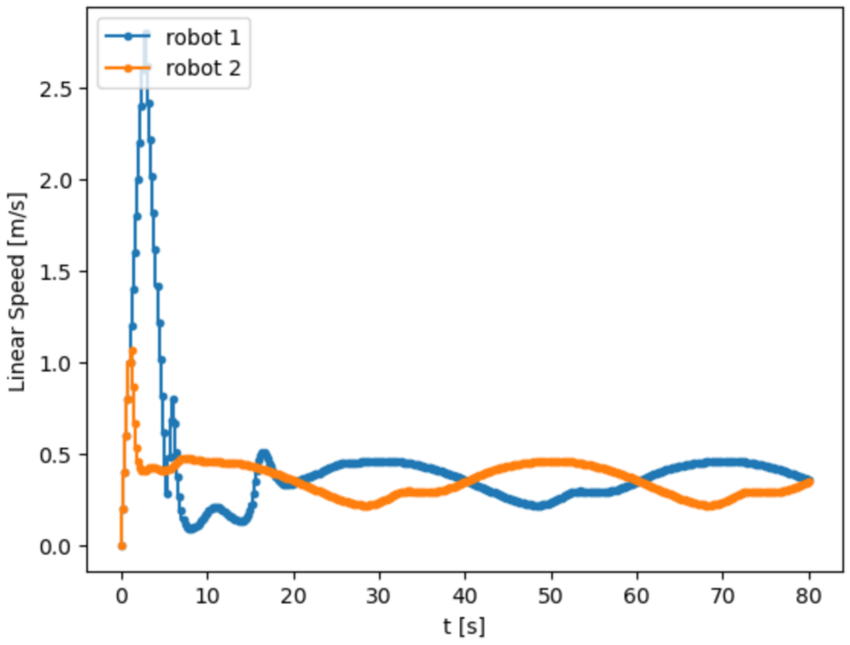


Figure 6.8: Linear speed of both robots as a function of time.

Finally, Figure 6.9 shows the angular velocity of both robots over time. This plot provides insight into how each robot adjusts its rotational speed to maintain coordinated motion. The angular speed profiles demonstrate smooth transitions and effective rotational control.

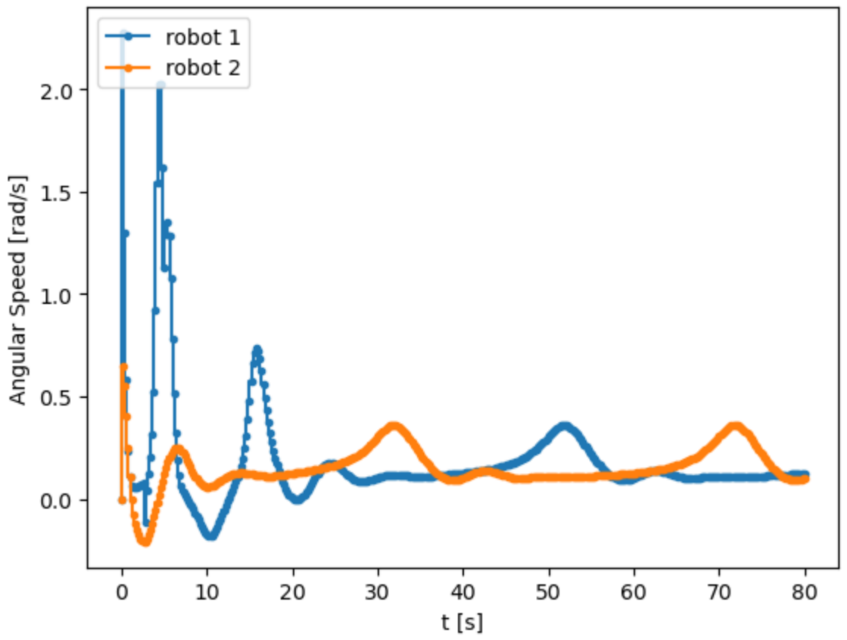


Figure 6.9: Angular speed of both robots as a function of time.

These plots collectively demonstrate the potential of the `controllerMovie` to achieve efficient and coordinated navigation for multiple robots. Despite not being fully operational in ROS, the preliminary results indicate that the synchronizer and control strategies offer promising capabilities for future development.

Conclusion

This project presented a comprehensive study of control strategies for autonomous multi-robot systems, addressing both simulation-based exploration and real-world implementation. In Part I, the focus was on developing and evaluating optimization-based controllers using MATLAB. These strategies proved effective in navigating complex trajectories and achieving precise motion objectives. Simulations validated the robustness of the controllers while highlighting trade-offs between complexity, performance, and computational efficiency.

Part II built upon these findings by transitioning to a physical robotics platform. Using ROS and the NVIDIA Jetbot, the implementation demonstrated the feasibility of real-world applications, specifically guiding robots to orbit a moving target under spatial and orientation constraints. This transition highlighted challenges such as system integration and the need for robust real-time processing, which were effectively addressed by leveraging ROS tools and the computational capabilities of the Jetson Nano processor.

The integration of Model Predictive Control (MPC) and Nonlinear Model Predictive Control (NMPC) across both parts showcased advanced coordination and trajectory tracking, enabling collision-free navigation and adaptation to dynamic environments. These methods, combined with innovative coordination algorithms, provided a solid framework for autonomous multi-robot operations.

Despite the promising results, scalability and real-world factors like sensor noise and communication delays remain areas for improvement. Future research could explore integrating reinforcement learning with model-based methods to enhance adaptability to complex, unknown environments. Additionally, deploying these strategies on larger fleets and incorporating sophisticated prediction models for dynamic obstacles are critical next steps.

In conclusion, this work demonstrates the effectiveness of advanced control strategies in achieving precise, efficient, and intelligent robotic operations. By addressing limitations and exploring extensions, it lays a robust foundation for future advancements in autonomous robotics, with potential applications in automation, search and rescue, and collaborative exploration.

References

[1] Daniel Silvestre, *Modeling and Optimal Control - Cyber-Physical Control Systems*, pt. Available at: *https://moodle.fct.unl.pt/pluginfile.php/755149/mod_resource/content/2/CPCS_Module_1 _Model_Opt.pdf.*

[2] Daniel Silvestre, *Modeling and Optimal Control - Cyber-Physical Control Systems*, pt. Available at: *https://moodle.fct.unl.pt/pluginfile.php/755145/mod_resource/content/2/CPCS_Module_2 _MPC.pdf.*

[3] Daniel Silvestre, *Control Lyapunov Functions and Control Barrier Functions - Cyber-Physical Control Systems*, pt. Available at: *https://moodle.fct.unl.pt/mod/resource/view.php?id=545774.*

[4] Daniel Silvestre, *Control Lyapunov Functions and Control Barrier Functions - Cyber-Physical Control Systems*, pt. Available at: *https://moodle.fct.unl.pt/mod/resource/view.php?id=545774.*

[5] Daniel Silvestre, *Distributed and Decentralized MPC - Cyber-Physical Control Systems*, pt. Available at: *https://moodle.fct.unl.pt/mod/resource/view.php?id=545772.*

[6] Daniel Silvestre, *Nonlinear MPC Analysis - Cyber-Physical Control Systems*, pt. Available at: *https://moodle.fct.unl.pt/mod/resource/view.php?id=545769.*

[7] CasADi - Docs. Available at: *https://web.casadi.org/docs/.*

[8] Yalmip. Available at: *https://yalmip.github.io/tutorial/installation/.*

[9] ROS. Available at: *http://wiki.ros.org/melodic/Installation/Ubuntu.*

[9] ROS. Available how to use at: *http://wiki.ros.org/ROS/Tutorials.*

Anexos

Anexo 1: Função `controllerTracking`

Listing 6.1: Função `controllerTracking` utilizada no controle MPC.

```
function [u_mpc] = controllerTracking(x0)
    % Persistent variables to retain initialization across calls
    persistent i r_extended controller u_mpc_prev horizon

    % Initialize persistent variables on first call
    if isempty(controller)

        % System definitions
        b = 1;              % Distance between wheels
        v_max = 10;         % Maximum speed (m/s)
        a_max = 0.5;        % Maximum acceleration (m/s^2)
        T_s = 0.1;          % Sampling time
        horizon = 10;       % Prediction horizon for MPC

        i = 1;  % Starting index
        % Simulation time
        t = 0:T_s:20 * pi;

        % Reference definition
        ref = [5 * sin(t / 5); 5 * cos(t / 10); zeros(1, length(t))];

        % Extend the reference
        r_extended = [ref, ref(:, end * ones(1, horizon))];

        % Initialize previous control inputs
        u_mpc_prev = [0; 0];

        % State and control variables
        nx = 3; % Number of states
        nu = 2; % Number of inputs

        x = sdpvar(repmat(nx, 1, horizon + 1), repmat(1, 1, horizon + 1));
        vr = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
        vl = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
        u = sdpvar(repmat(nu, 1, horizon), repmat(1, 1, horizon));
        a = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));

        % Use u_prev instead of v_prev and omega_prev
        u_prev = sdpvar(repmat(nu, 1, horizon + 1), repmat(1, 1, horizon + 1));
        r = sdpvar(repmat(nx, 1, horizon + 1), repmat(1, 1, horizon + 1));

        % Cost weights
        Q = diag([10, 10, 0]); % Tracking error
        R = diag([1, 1]);       % Control effort

        % Initialize objective and constraints
        objective = 0;
        constraints = [];

        for k = 1:horizon
            % Compute linear and angular velocities
            v = 0.5 * (vr{k} + vl{k});
            omega = (vr{k} - vl{k}) / b;
            u{k} = [v; omega];

            % Cost function
            objective = objective + (x{k+1} - r{k})' * Q * (x{k+1} - r{k}) + u{k}' *

            % Dynamics constraints
            x_next = x{k}(1) + T_s * v * cos(x{k}(3));
            y_next = x{k}(2) + T_s * v * sin(x{k}(3));
            theta_next = x{k}(3) + T_s * omega;

            constraints = [constraints, x{k+1}(1) == x_next];
            constraints = [constraints, x{k+1}(2) == y_next];
            constraints = [constraints, x{k+1}(3) == theta_next];

            % Control constraints
            constraints = [constraints, -v_max <= v <= v_max];

            % Acceleration constraints
            a{k} = (v - u_prev{k}(1)) / T_s;
            constraints = [constraints, -a_max <= a_max];

            % Store previous control inputs
            constraints = [constraints, u_prev{k+1} == u{k}];
        end

        % Define the optimizer
        parameters_in = {x{1}, [r{:}], u_prev{1}};
        solutions_out = {[u{:}]};
        options = sdpsettings('verbose', 1, 'debug', 1);
        controller = optimizer(constraints, objective, options, parameters_in, soluti
    end

    % Get the reference trajectory for the current horizon
    future_r = r_extended(:, i:i + horizon);

    % Solve the optimization problem
    inputs = {x0, future_r, u_mpc_prev};
    [solutions, diagnostics] = controller{inputs};

    % Check for infeasibility
    if diagnostics == 1
        error('The problem is infeasible');
    end

    % Extract control input
    u_mpc = solutions(:, 1);

    % Update previous control inputs
    u_mpc_prev = u_mpc;

    % Increment the reference index
    if i < size (r_extended-horizon,2)
    i = i + 1;
end
```

Anexo 2: Função `estimateIntersectionPoint`

Listing 6.2: Função `estimateIntersectionPoint` utilizada para estimar pontos de interseção entre os robôs.

```
function [ref_estimated, intersection_indices, has_intersection, times_target, times_
    estimateIntersectionPoint(target_path, target_pos, target_vel, follower_pos, foll
% Estimate interception position of the follower robot with the target robot
% considering the target follows a predefined trajectory.
%
% Inputs:
%   - target_path: Matrix of trajectory positions [x; y; theta] (3 x N)
%   - target_pos: Current position [x; y; theta] of the target robot (3 x 1)
%   - target_vel: Current velocity [v; omega] of the target robot (2 x 1)
%   - follower_pos: Current position [x; y; theta] of the follower robot (3 x 1)
%   - follower_vel: Current velocity [v; omega] of the follower robot (2 x 1)
%   - Ts: Sampling time
%   - max_intersections: Maximum number of intersection points to find
%
% Outputs:
%   - ref_estimated: Intersection reference positions [x; y; theta] (3 x M) for M int
%   - intersection_indices: Indices of the trajectory points corresponding to interse
%   - has_intersection: Boolean indicating if intersection exists (true/false)
%   - times_target: Times for the target to reach each point in the path
%   - times_follower: Times for the follower to reach each point in the path
%   - distances_follower: Euclidean distances between the follower and each point in
%   - distances_target: Cumulative distances traveled by the target at each point

% Unpack velocities
v_target = target_vel(1);
v_follower = follower_vel(1);

% Unpack follower position
```

```
x_follower = follower_pos(1);
y_follower = follower_pos(2);

% Initialize variables
num_points = size(target_path, 2);
ref_estimated = []; % Store all intersection points found
intersection_indices = []; % Store indices of intersections
has_intersection = false;
min_time_diff = Ts * v_target / 2; % Tolerance for intersection time

% Initialize cumulative distances and arrays to store times/distances
cumulative_distance_target = 0;
distances_target = zeros(1, num_points + 1);
times_target = zeros(1, num_points + 1);
times_follower = zeros(1, num_points + 1);
distances_follower = zeros(1, num_points + 1);

% Include the target's current position in the path
full_path = [target_pos, target_path];

% Loop through trajectory points starting at i = 2
for i = 2:num_points + 1
    % Calculate segment distance for the target's path
    dx = full_path(1, i) - full_path(1, i - 1);
    dy = full_path(2, i) - full_path(2, i - 1);
    segment_distance = sqrt(dx^2 + dy^2);
    cumulative_distance_target = cumulative_distance_target + segment_distance;

    % Store cumulative distance for the target
    distances_target(i) = cumulative_distance_target;

    % Time for the target to reach this point
    times_target(i) = cumulative_distance_target / v_target;

    % Euclidean distance for the follower to this point
    distances_follower(i) = sqrt((x_follower - full_path(1, i))^2 + (y_follower - ful

    % Time for the follower to reach this point
    times_follower(i) = distances_follower(i) / v_follower;

    % Check for intersection
    if (abs(times_follower(i) - times_target(i)) < min_time_diff && length(intersecti
            (times_follower(i) < times_target(i) && ...
             norm(full_path(1:2, i) - ref_estimated(1:2, end)) >= 2)
        % Store intersection
        ref_estimated = [ref_estimated, full_path(:, i)];
        intersection_indices = [intersection_indices, i];
        has_intersection = true;

        % Stop if maximum intersections are found
        if length(intersection_indices) >= max_intersections
            break;
        end
    end
end

% Remove the initial values (zero at i = 1) for times and distances
times_target = times_target(2:i);
times_follower = times_follower(2:i);
distances_follower = distances_follower(2:i);
distances_target = distances_target(2:i);

end
```

Anexo 3: Função `ReferenceWithPoints`

Listing 6.3: Função `ReferenceWithPoints` utilizada para interpolação de pontos para gerar uma referência.

```
function ref = ReferenceWithPoints(points, T_s, max_velocity)
    % Function to expand a vector of points by adding interpolated points
    % based on the sampling time and maximum velocity
    %
    % Inputs:
    % - points: matrix of points (each column is a point [x; y; theta])
    % - T_s: sampling time
    % - max_velocity: maximum velocity (assumed constant for interpolation)
    %
    % Output:
    % - ref: matrix with interpolated points

    % Validate input
    if size(points, 1) ~= 3
        error('The input points must be a 3xN matrix.');
    end
    if size(points, 2) < 2
        error('The points matrix must contain at least two points.');
    end
    if T_s <= 0
        error('The sampling time T_s must be positive.');
    end
    if max_velocity <= 0
        error('The maximum velocity must be positive.');
    end

    % Initialize the expanded reference
    ref = [];

    % Loop through each consecutive pair of points
    for i = 1:(size(points, 2) - 1)
        % Start and end points
        p_start = points(:, i);
        p_end = points(:, i + 1);
        % Calculate distance between the start and end points
        distance = norm(p_end(1:2) - p_start(1:2));

        % Compute the time needed to traverse the distance
        time_to_traverse = distance / max_velocity;

        % Calculate the number of interpolation steps
        num_interpolation_points = ceil(time_to_traverse / T_s);

        % Interpolate each dimension
        interpolated_points = [];
        for dim = 1:size(points, 1)
            interpolated_points = [interpolated_points; ...
                                   linspace(p_start(dim), p_end(dim), num_interpolati
        end

        % Exclude the last point to avoid duplication
        ref = [ref, interpolated_points(:, 1:end-1)];
    end

    % Add the final point
    ref = [ref, points(:, end)];
end
```

Anexo 4: Função `selectBestReferences`

Listing 6.4: Função `selectBestReferences` para selecionar as melhores referências com base nas interseções.

```
function [selected_ref_1, selected_ref_2] = selectBestReferences(...
    ref_estimated_1, has_intersection_1, intersection_indices_1, ...
    ref_estimated_2, has_intersection_2, intersection_indices_2)
% Selects the best references for two robots based on intersections.
%
% Inputs:
%   - ref_estimated_1: Estimated references for robot 1 (3 x M1)
%   - has_intersection_1: Boolean, true if robot 1 has intersections
%   - intersection_indices_1: Indices of intersections for robot 1
%   - ref_estimated_2: Estimated references for robot 2 (3 x M2)
%   - has_intersection_2: Boolean, true if robot 2 has intersections
%   - intersection_indices_2: Indices of intersections for robot 2
%
% Outputs:
%   - selected_ref_1: Selected reference for robot 1 [x; y; theta]
%   - selected_ref_2: Selected reference for robot 2 [x; y; theta]

% Default outputs
selected_ref_1 = [];
selected_ref_2 = [];
```

```
% Proceed only if both robots have intersections
if has_intersection_1 && has_intersection_2
    % Find the earliest intersection for each robot
    min_index_1 = intersection_indices_1(1); % First intersection for Robot 1
    min_index_2 = intersection_indices_2(1); % First intersection for Robot 2

    if min_index_1 < min_index_2
        % Robot 1 intersects first
        selected_ref_1 = ref_estimated_1(:, 1); % First intersection for Robot 1

        % Check for a valid second reference for Robot 2
        for i = 2:length(intersection_indices_2) % Start at the second element
            if intersection_indices_2(i) > min_index_1 && ...
               norm(ref_estimated_2(:, i) - selected_ref_1) > 2
                selected_ref_2 = ref_estimated_2(:, i); % Select farther intersection
                break; % Stop after finding a valid reference
            end
        end

        % Default to the first intersection if no valid farther reference is found
        if isempty(selected_ref_2)
            selected_ref_2 = ref_estimated_2(:, 1);
        end
    else
        % Robot 2 intersects first
        selected_ref_2 = ref_estimated_2(:, 1); % First intersection for Robot 2

        % Check for a valid second reference for Robot 1
        for i = 2:length(intersection_indices_1) % Start at the second element
            if intersection_indices_1(i) > min_index_2 && ...
               norm(ref_estimated_1(:, i) - selected_ref_2) > 2
                selected_ref_1 = ref_estimated_1(:, i); % Select farther intersection
                break; % Stop after finding a valid reference
            end
        end

        % Default to the first intersection if no valid farther reference is found
        if isempty(selected_ref_1)
            selected_ref_1 = ref_estimated_1(:, 1);
        end
    end
else
    % Handle cases where one or both robots have no intersections
    if has_intersection_1
        selected_ref_1 = ref_estimated_1(:, 1);
    end

    if has_intersection_2
        selected_ref_2 = ref_estimated_2(:, 1);
    end
end
```

```
end
```

Anexo 5: Função `controllerCooperative`

Listing 6.5: Função `controllerCooperative`

```
function [u_mpc] = controllerCooperative(x0)
    % ================================================
    % CONTROLLER COOPERATIVE FUNCTION
    % ================================================
    % Inputs:
    %   x0 - Current states of the system [follower_1; follower_2]
    % Outputs:
    %   u_mpc - Control inputs for the followers
    %
    % Persistent variables:
    %   - controller: MPC optimizer instance
    %   - has_intersection_1/2: Intersection status for followers
    %   - selected_ref_1/2: Selected references for followers
    %   - u_mpc_prev_1/2: Previous control inputs
    %   - ref: Reference trajectory
    %   - i: Current time step index
    %   - target_vel_max, follower_v_max: Speed constraints
    %   - T_s: Sampling time
    %   - max_intersections: Maximum intersections to evaluate
    % ================================================

    % Persistent variables
    persistent controller has_intersection_1 has_intersection_2 selected_ref_1 select
    persistent u_mpc_prev_1 u_mpc_prev_2 ref i target_vel_max follower_v_max T_s max

    % Initialize persistent variables
    if isempty(controller)
        % System parameters
        b = 1;                    % Distance between wheels
        follower_v_max = 2;       % Max follower speed (m/s)
        target_vel_max = 10;      % Max target speed (m/s)
        max_intersections = 4;    % Max intersections to evaluate
        T_s = 0.1;                % Sampling time
        horizon = 4;              % MPC prediction horizon
        time = 10;                % Simulation time
        i = 1;                    % Time step index
        min_dist_diff = T_s *target_vel_max ;      % Minimum distance difference to 
        v_max = 10;
        a_max = 1;

        % Initial positions
        target_initial_pos = [-5; 0; 0];          % Target initial position [x; y; th
        target_final_pos = target_initial_pos + [target_vel_max * time; 0; 0]; % Com

        % Reference trajectory
        points = [target_initial_pos, target_final_pos];
        ref = ReferenceWithPoints(points, T_s, target_vel_max);

        % Initialize previous control inputs
        u_mpc_prev_1 = [0; 0];
        u_mpc_prev_2 = [0; 0];

        % MPC setup
        nx = 3; % Number of states
        nu = 2; % Number of inputs
        Q = diag([10, 10, 0.1]);   % State weighting matrix
        R = diag([0.1, 0.01]);     % Input weighting matrix


        % Define optimization variables
        x = sdpvar(repmat(nx, 1, horizon + 1), repmat(1, 1, horizon + 1)); % States
        vr = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
% Right wheel velocity
        vl = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
% Left wheel velocity
        u = sdpvar(repmat(nu, 1, horizon), repmat(1, 1, horizon));
% Control inputs
        a = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
% Accelerations
        u_prev = sdpvar(repmat(nu, 1, horizon + 1), repmat(1, 1, horizon + 1)); % Pr
        r = sdpvar(repmat(nx,1,1), repmat(1,1,1)); % Reference position

        % Objective and constraints
        objective = 0;
        constraints = [];


    for k = 1:horizon

        % Linear and angular speed inputs
        v = 0.5 * (vr{k} + vl{k});
        omega = (vr{k} - vl{k}) / b;
        u{k} = [v; omega];

        % Tracking error cost (distance to reference position)
        objective = objective + (x{k+1} - r)' * Q * (x{k+1} - r) + u{k}' * R * u{k};

        % Discretized dynamics
        x_next = x{k}(1) + T_s * v * cos(x{k}(3));
        y_next = x{k}(2) + T_s * v * sin(x{k}(3));
        theta_next = x{k}(3) + T_s * omega;


        % System dynamics constraints
        constraints = [constraints, x{k+1}(1) == x_next];
        constraints = [constraints, x{k+1}(2) == y_next];
        constraints = [constraints, x{k+1}(3) == theta_next];
        % Store previous control inputs
        constraints = [constraints, u_prev{k+1} == u{k}];

        % Control input constraints
        constraints = [constraints, -v_max <= v <= v_max];

        % Linear acceleration constraint
        a{k} = (v - u_prev{k}(1)) / T_s;
        constraints = [constraints, -a_max <= a{k} <= a_max];
    end

    % ===============================================================
    % DEFINE THE OPTIMIZER
    % ===============================================================

    parameters_in = {x{1},r, u_prev{1}};
    solutions_out = {[u{:}]};
    options = sdpsettings('verbose', 0, 'debug', 0);
    controller = optimizer(constraints, objective, options, parameters_in, solutions_

    % ===============================================================
    % UPDATE REFERENCES AND EVALUATE INTERSECTIONS
    % ===============================================================

    [ref_estimated_1, intersection_indices_1, has_intersection_1, ~, ~, ~, ~] = ...
    estimateIntersectionPoint(ref, ref(:,1), target_vel_max, x0(1:3), follower_v_max,

    [ref_estimated_2, intersection_indices_2, has_intersection_2, ~, ~, ~, ~] = ...
    estimateIntersectionPoint(ref, ref(:,1), target_vel_max, x0(4:6), follower_v_max,

    [selected_ref_1, selected_ref_2] = selectBestReferences( ref_estimated_1, has_int
end

    follower_1_pos = x0(1:3);
    follower_2_pos = x0(4:6);
    target_pos = ref(:,i);
     % Dist ncias do target para os rob s
    dist_1_target = norm(follower_1_pos(1:2) - target_pos(1:2));
    dist_2_target = norm(follower_2_pos(1:2) - target_pos(1:2));

    % Caso 1: Apenas o Robo 1 tem uma interse  o
    if has_intersection_1 && ~has_intersection_2
        % Verificar se o Robo 1 chegou   refer ncia
        if norm(follower_1_pos(1:2) - selected_ref_1(1:2)) < min_dist_diff
            % Verificar se o target est  pr ximo (dentro do limite de 2 unidades)
            if dist_1_target >= 0 && dist_1_target <= 1
                % Caso ideal: Target est  pr ximo, mudar foco para segui-lo
                selected_ref_1 = target_pos;
                has_intersection_1 = false; % Desconsiderar interse  es

                % Recalcular interse  es para o Robo 2
                [ref_estimated_2, intersection_indices_2, has_intersection_2, ~, ~,
                estimateIntersectionPoint(ref(:, i:end), target_pos, target_vel_m
        end
        % Verificar se o Robo 1 encontra o target no caminho
    elseif dist_1_target <= 1
        selected_ref_1 = target_pos; % Seguir o target diretamente
        has_intersection_1 = false; % Desconsiderar interse   es

        % Recalcular interse   es para o Robo 2
        [ref_estimated_2, intersection_indices_2, has_intersection_2, ~, ~, ~, ~]
            estimateIntersectionPoint(ref(:, i:end), target_pos, target_vel_max,
    else
        % Caso 3: O target chegou primeiro    interse  o
        if norm(target_pos(1:2) - selected_ref_1(1:2)) < min_dist_diff

            % Recalcular interse   es para ambos os rob s
            [ref_estimated_1, intersection_indices_1, has_intersection_1, ~, ~, ~
                estimateIntersectionPoint(ref(:, i:end), target_pos, target_vel_m

            [ref_estimated_2, intersection_indices_2, has_intersection_2, ~, ~, ~
                estimateIntersectionPoint(ref(:, i:end), target_pos, target_vel_m

            % Atualizar as refer ncias se novas interse   es forem encontradas
            if has_intersection_1
                [selected_ref_1, ~] = ...
                    selectBestReferences(ref_estimated_1, has_intersection_1, int
                    ref_estimated_2, has_intersection_2, intersection_indices_2);
            else
                selected_ref_1 = target_pos; % Seguir o target diretamente
            end
        end
    end

    % Verificar se h  interse   es para o Robo 2
    if has_intersection_2
        [selected_ref_1, selected_ref_2] = ...
            selectBestReferences(ref_estimated_1, has_intersection_1, intersectio
            ref_estimated_2, has_intersection_2, intersection_indices_2);
    else
        % Caso contr rio, o Robo 2 segue diretamente o target
        selected_ref_2 = target_pos;
    end

    % Caso 2: Apenas o Robo 2 tem uma interse  o
elseif ~has_intersection_1 && has_intersection_2
    % Verificar se o Robo 2 chegou    refer ncia
    if norm(follower_2_pos(1:2) - selected_ref_2(1:2)) < min_dist_diff
        % Verificar se o target est  pr ximo (dentro do limite de 2 unidades)
        if dist_2_target >= 0 && dist_2_target <= 1
            % Caso ideal: Target est  pr ximo, mudar foco para segui-lo
            selected_ref_2 = target_pos;
            has_intersection_2 = false; % Desconsiderar interse   es
            % Recalcular interse  es para o Robo 1
            [ref_estimated_1, intersection_indices_1, has_intersection_1, ~, ~, ~
                estimateIntersectionPoint(ref(:, i:end),target_pos, target_vel_ma
        end
        % Verificar se o Robo 2 encontra o target no caminho
    elseif dist_2_target <= 1
        selected_ref_2 = target_pos; % Seguir o target diretamente
        has_intersection_2 = false; % Desconsiderar interse  es

        % Recalcular interse  es para o Robo 1
        [ref_estimated_1, intersection_indices_1, has_intersection_1, ~, ~, ~, ~]
            estimateIntersectionPoint(ref(:, i:end), target_pos, target_vel_max,
    else
        % Caso 3: O target chegou primeiro   interse  o
        if norm(target_pos(1:2) - selected_ref_2(1:2)) < min_dist_diff

            % Recalcular interse  es para ambos os rob s
            [ref_estimated_1, intersection_indices_1, has_intersection_1, ~, ~, ~
                estimateIntersectionPoint(ref(:, i:end),  target_pos, target_vel_r

            [ref_estimated_2, intersection_indices_2, has_intersection_2, ~, ~, ~
                estimateIntersectionPoint(ref(:, i:end),  target_pos, target_vel_r

            % Atualizar as refer ncias se novas interse  es forem encontradas
            if has_intersection_2
                [~, selected_ref_2] = ...
                    selectBestReferences(ref_estimated_1, has_intersection_1, int
                    ref_estimated_2, has_intersection_2, intersection_indices_2);
            else
                selected_ref_2 = target_pos; % Seguir o target diretamente
            end
        end
    end

    % Verificar se h  interse  es para o Robo 1
    if has_intersection_1
        [selected_ref_1, selected_ref_2] = ...
            selectBestReferences(ref_estimated_1, has_intersection_1, intersectio
            ref_estimated_2, has_intersection_2, intersection_indices_2);
    else
        % Caso contr rio, o Robo 1 segue diretamente o target
        selected_ref_1 = target_pos;
    end


    % Caso 3: Ambos os rob s t m pontos de interse  o
elseif has_intersection_1 && has_intersection_2
    % Robo 1 segue a refer ncia escolhida
    if norm(follower_1_pos(1:2) - selected_ref_1(1:2)) < min_dist_diff
        % Verificar se o target est  pr ximo (dentro do limite de 2 unidades)
        if dist_1_target >= 0 && dist_1_target <= 1
            % Caso ideal: Target est  pr ximo, mudar foco para segui-lo
        selected_ref_1 = target_pos;
        has_intersection_1 = false; % Desconsiderar interse  es

    end
    % Verificar se o Robo 1 encontra o target no caminho
elseif dist_1_target <= 1
    selected_ref_1 = target_pos; % Seguir o target diretamente
    has_intersection_1 = false; % Desconsiderar interse  es


else
    % Caso 3: O target chegou primeiro   interse  o
    if norm(target_pos(1:2) - selected_ref_1(1:2)) < min_dist_diff

        % Recalcular interse  es para o robo 1
        [ref_estimated_1, intersection_indices_1, has_intersection_1, ~, ~,
            estimateIntersectionPoint(ref(:, i:end), target_pos, target_vel_
        % Atualizar as refer ncias se novas interse  es forem encontradas
        if has_intersection_1
            [selected_ref_1, ~] = ...
                selectBestReferences(ref_estimated_1, has_intersection_1, int
                ref_estimated_2, has_intersection_2, intersection_indices_2);
        else
            selected_ref_1 = target_pos; % Seguir o target diretamente
        end
    end
end

% Robo 2 segue a refer ncia escolhida
if norm(follower_2_pos(1:2) - selected_ref_2(1:2)) < min_dist_diff
    % Verificar se o target est  pr ximo (dentro do limite de 2 unidades)
    if dist_2_target >= 0 && dist_2_target <= 1
        % Caso ideal: Target est  pr ximo, mudar foco para segui-lo
        selected_ref_2 = target_pos;
        has_intersection_2 = false; % Desconsiderar interse  es

    end
    % Verificar se o Robo 2 encontra o target no caminho
elseif dist_2_target <= 1
    selected_ref_2 = target_pos; % Seguir o target diretamente
    has_intersection_2 = false; % Desconsiderar interse  es

else
    % Caso 3: O target chegou primeiro   interse  o
    if norm(target_pos(1:2) - selected_ref_2(1:2)) < min_dist_diff

        % Recalcular interse  es para o robo 2

        [ref_estimated_2, intersection_indices_2, has_intersection_2,~, ~, ~,
            estimateIntersectionPoint(ref(:, i:end), target_pos, target_vel_

        % Atualizar as refer ncias se novas interse  es forem encontradas
```

```
                if has_intersection_2
                    [selected_ref_1, selected_ref_2] = ...
                        selectBestReferences(ref_estimated_1, has_intersection_1, int
                        ref_estimated_2, has_intersection_2, intersection_indices_2);

                else
                    selected_ref_2 = target_pos; % Seguir o target diretamente
                end
            end
        end
        % Caso 4: Nenhum rob  tem interse  o, ambos seguem o target
    elseif ~has_intersection_1 && ~has_intersection_2
        selected_ref_1 = target_pos;
        selected_ref_2 = target_pos;

    end




    % Entradas para o controlador dos dois rob s
    inputs_1 = {follower_1_pos, selected_ref_1,  u_mpc_prev_1 };
    inputs_2 = {follower_2_pos, selected_ref_2, u_mpc_prev_2};

    % Resolver MPC para os dois rob s
    [solutions_1, diagnostics_1] = controller{inputs_1};
    [solutions_2, diagnostics_2] = controller{inputs_2};

    % Verificar se o problema   invi vel
    if diagnostics_1 == 1 || diagnostics_2 == 1
        error('The problem is infeasible');
    end

    % Extrair entradas de controle e previs es de estados
    u_mpc_prev_1 = solutions_1(:, 1);  % Controle do rob  1
    u_mpc_prev_2 = solutions_2(:, 1);  % Controle do rob  2
    u_mpc = [u_mpc_prev_1;u_mpc_prev_2];

    i = i +1;
end
```

Anexo 6: Função `controllerMovie`

Listing 6.6: Função `controllerMovie`

```
function [u_mpc] = controllerMovie(x0)

persistent controller p1_extended p2_extended i u_mpc_prev_1 u_mpc_prev_2 horizon

 % Initialize persistent variables on first call
    if isempty(controller)
 b = 1;             % Distance between wheels
v_max = 2;        % Maximum speed (m/s)
a_max = 1;       % Maximum acceleration (m/s^2)
T_s = 0.1;         % Sampling time
horizon = 10;     % Prediction horizon for MPC
omega = pi / 20;  % Velocidade angular
r = 2;              % Raio do c rculo
v_central = 2;     % Velocidade do ve culo central

% Tempo de simula  o
t_total = 80;      % Tempo total (segundos)
t = 0:T_s:t_total; % Vetor de tempo
T =length(t);

i = 1;  % Starting index

% Movimento do ve culo central (centro do c rculo)
p_central = [v_central * t; zeros(1, length(t))];

% Movimento circular dos rob s
theta = omega * t; %  ngulo  ao longo do tempo
p1 = [p_central(1, :) + r * cos(theta);
      p_central(2, :) + r * sin(theta);
      zeros(1, T)];
p2 = [p_central(1, :) + r * cos(theta + pi);
      p_central(2, :) + r * sin(theta + pi);
      zeros(1, T)];

% Extend the reference
p1_extended = [p1, p1(:, end * ones(1, horizon))];
p2_extended = [p2, p2(:, end * ones(1, horizon))];

% Initialize previous control inputs
u_mpc_prev_1 = [0; 0];
u_mpc_prev_2 = [0; 0];


% State and control variables
nx = 3; % Number of states
nu = 2; % Number of inputs

x = sdpvar(repmat(nx, 1, horizon + 1), repmat(1, 1, horizon + 1));
vr = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
vl = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
u = sdpvar(repmat(nu, 1, horizon), repmat(1, 1, horizon));
a = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));

% Use u_prev instead of v_prev and omega_prev
u_prev = sdpvar(repmat(nu, 1, horizon + 1), repmat(1, 1, horizon + 1));

r = sdpvar(repmat(nx, 1, horizon + 1), repmat(1, 1, horizon + 1));
```

```
        % Cost weights
        Q = diag([10, 10, 0]); % Tracking error
        R = diag([1, 1]);       % Control effort

        % Initialize objective and constraints
        objective = 0;
        constraints = [];

        for k = 1:horizon
            % Compute linear and angular velocities
            v = 0.5 * (vr{k} + vl{k});
            omega = (vr{k} - vl{k}) / b;
            u{k} = [v; omega];

            % Cost function
            objective = objective + (x{k+1} - r{k})' * Q * (x{k+1} - r{k}) + u{k}' *

            % Dynamics constraints
            x_next = x{k}(1) + T_s * v * cos(x{k}(3));
            y_next = x{k}(2) + T_s * v * sin(x{k}(3));
            theta_next = x{k}(3) + T_s * omega;

            constraints = [constraints, x{k+1}(1) == x_next];
            constraints = [constraints, x{k+1}(2) == y_next];
            constraints = [constraints, x{k+1}(3) == theta_next];

            % Control constraints
            constraints = [constraints, -v_max <= v <= v_max];

            % Acceleration constraints
            a{k} = (v - u_prev{k}(1)) / T_s;
            constraints = [constraints, -a_max <= a{k} <= a_max];



            % Store previous control inputs
            constraints = [constraints, u_prev{k+1} == u{k}];
        end

        % Define the optimizer
        parameters_in = {x{1}, [r{:}], u_prev{1}};
        solutions_out = {[u{:}]};
        options = sdpsettings('verbose', 0, 'debug', 0);
        controller = optimizer(constraints, objective, options, parameters_in, soluti
    end

     follower_1_pos = x0(1:3);
     follower_2_pos = x0(4:6);


      % Get the reference trajectory for the current horizon
    future_r1 = p1_extended(:, i:i + horizon);
    % Get the reference trajectory for the current horizon
    future_r2 = p2_extended(:, i:i + horizon);

    % Solve the optimization problem
    inputs_1 = {follower_1_pos, future_r1, u_mpc_prev_1};
    inputs_2 = {follower_2_pos, future_r2, u_mpc_prev_2};


    % Resolver MPC para os dois rob s
    [solutions_1, diagnostics_1] = controller{inputs_1};
    [solutions_2, diagnostics_2] = controller{inputs_2};


    % Verificar se o problema   invi vel
    if diagnostics_1 == 1 || diagnostics_2 == 1
        error('The problem is infeasible');
    end


    % Extrair entradas de controle e previs es de estados
    u_mpc_prev_1 = solutions_1(:, 1); % Controle do rob  1
    u_mpc_prev_2 = solutions_2(:, 1); % Controle do rob  2
    u_mpc = [u_mpc_prev_1;u_mpc_prev_2];

    i = i +1;
end
```

Anexo 7: Função `controllerSimple`

Listing 6.7: Função `controllerSimple`

```
function [u_mpc] = controllerSimple(x0)
    % Persistent variables to retain initialization across calls
    persistent controller horizon b v_max a_max alpha_max T_s ref Q R u_mpc_prev

    % Check if persistent variables are already initialized
    if isempty(controller)
        % Initialize persistent variables
        b = 1;                    % Distance between wheels
        v_max = 10;               % Maximum speed (m/s)
        a_max = 1;                % Maximum acceleration (m/s^2)
        alpha_max = 4;          % Maximum angular acceleration (rad/s^2)
        T_s = 0.1;                % Sampling time
        horizon = 2;              % Prediction horizon for MPC
        ref = [10; 10; 0];        % Target reference position
        Q = diag([10, 10, 0]); % State weights
        R = diag([8, 1]);          % Control input weights

        % Initialize `u_mpc_prev` to zero
        u_mpc_prev = [0; 0];
        % Define YALMIP variables
        nx = 3; % Number of states
        nu = 2; % Number of inputs

        x  = sdpvar(repmat(nx, 1, horizon + 1), repmat(1, 1, horizon + 1)); % States
        vr = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
% Right wheel velocity
        vl = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
% Left wheel velocity
        u = sdpvar(repmat(nu, 1, horizon), repmat(1, 1, horizon));
% Control inputs
        a = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
% Linear acceleration
%          alpha = sdpvar(repmat(1, 1, horizon), repmat(1, 1, horizon));
% Angular acceleration
        u_prev = sdpvar(repmat(nu, 1, horizon+1), repmat(1, 1, horizon+1));
% Control inputs previous


        % Initialize objective and constraints
        objective = 0;
        constraints = [];

        % Define the MPC optimization problem
        for k = 1:horizon
            % Linear and angular speed inputs
            v = 0.5 * (vr{k} + vl{k});
            omega = (vr{k} - vl{k}) / b;
            u{k} = [v; omega];

            % Tracking error cost
            objective = objective + (x{k+1} - ref)' * Q * (x{k+1} - ref) + u{k}' * R

            % Discretized dynamics
            x_next = x{k}(1) + T_s * v * cos(x{k}(3));
            y_next = x{k}(2) + T_s * v * sin(x{k}(3));
            theta_next = x{k}(3) + T_s * omega;

            % System dynamics constraints
            constraints = [constraints, x{k+1}(1) == x_next];
            constraints = [constraints, x{k+1}(2) == y_next];
            constraints = [constraints, x{k+1}(3) == theta_next];
%              constraints = [constraints, u_prev{k+1}(1) == v];
%              constraints = [constraints, u_prev{k+1}(2) == omega];
            % Store previous control inputs
            constraints = [constraints, u_prev{k+1} == u{k}];

            % Control input constraints
            constraints = [constraints, -v_max <= v <= v_max];

            % Linear acceleration constraint
            a{k} = (v - u_prev{k}(1)) / T_s;
            constraints = [constraints, -a_max <= a{k} <= a_max];

            % Angular acceleration constraint
%             alpha{k} = (omega - u_prev{k}(2)) / T_s;
%             constraints = [constraints, -alpha_max <= alpha{k} <= alpha_max];
        end

        % Define the optimizer
        parameters_in = {x{1}, u_prev{1}};
        solutions_out = {[u{:}]};
        options = sdpsettings('verbose', 0);
        controller = optimizer(constraints, objective, options, parameters_in, soluti
    end

    % Solve the MPC problem
    inputs = {x0, u_mpc_prev};
    [solutions, diagnostics] = controller{inputs};

    % Check for infeasibility
    if diagnostics == 1
        error('The problem is infeasible');
    end

    % Extract control input for the current time step
    u_mpc = solutions(:, 1);

    % Update persistent variable with the current control input
    u_mpc_prev = u_mpc;
end
```

Anexo 8: controller.py

Listing 6.8: controller.py

```
import rospy
import math
import tf
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import numpy as np
from casadi import *


class RobotController:
    def __init__(self):
        # Topics
        pub_topic_name = "/jetbot0/diff_drive_controller/cmd_vel"
        sub_topic_name = "/jetbot0/diff_drive_controller/odom"

        # ROS Subscribers and Publishers
        self.velocity_pub = rospy.Publisher(pub_topic_name, Twist, queue_size=10)
        self.velocity_sub = rospy.Subscriber(sub_topic_name, Odometry, self.odometry

        # Command message
        self.velocity_msg = Twist()


        #controller
        self.controller = None
        self.u_prev = vertcat(0,0)


    def odometry_callback(self ,msg):

        # Extract position and orientation from Odometry message
        x = msg.pose.pose.position.x
        y = msg.pose.pose.position.y
        orientation_q = msg.pose.pose.orientation

        # Convert quaternion to Euler angles (roll, pitch, yaw)
        quaternion = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_
        _, _, theta = tf.transformations.euler_from_quaternion(quaternion)
        # Log the extracted values
        rospy.loginfo("Position: x = %.2f, y = %.2f, Theta = %.2f", x, y, theta)

        # State vector as a column vector (3x1)
        state_vector = np.array([[x], [y], [theta]])

        # Call the controllerSimple function
        control_output = self.controllerSimple(state_vector)
        self.u_prev = control_output[:,0]


        # Assign linear and angular speeds to the Twist message
        self.velocity_msg.linear.x = control_output[0, 0]  # Linear speed
        self.velocity_msg.angular.z = control_output[1, 0]  # Angular speed



        # Log the control actions without non-ASCII characters
        rospy.loginfo("Linear Velocity: {:.2f} m/s, Angular Velocity: {:.2f} rad/s".f

        # Publish the command
        self.velocity_pub.publish(self.velocity_msg)

    def controllerSimple(self, robot_states):
        if self.controller is None:
             # Initialize constants
            b = 0.22  # Distance between wheels
            v_max = 1  # Maximum speed (m/s)
            a_max = 3  # Maximum acceleration (m/s^2)
            omega_max = 1  # Maximum angular velocity (rad/s)
```

```
alpha_max = 3  # Maximum angular acceleration (rad/s^2)
T_s = 0.5  # Sampling time
horizon = 5  # Prediction horizon
ref = vertcat(10, 10, 0)  # Target reference position
Q = diag(DM([10, 10, 0]))  # State weights
R = diag(DM([8, 1]))  # Control input weights

# CasADi optimizer
opti = Opti()

# Optimization variables
nx = 3  # Number of states
nu = 2  # Number of inputs
x = opti.variable(nx, horizon + 1)  # States

u = opti.variable(nu, horizon)  # Control inputs

vr = opti.variable(1, horizon)  # Right wheel velocity
vl = opti.variable(1, horizon)  # Left wheel velocity

a = opti.variable(1, horizon)  # Linear acceleration
alpha = opti.variable(1, horizon)  # Angular acceleration

# Parameters
x0 = opti.parameter(nx,1)  # Initial state
u_prev = opti.parameter(nu,1)  # Previous control input

# Define state and control model
f = lambda x,u: vertcat(u[0]*cos(x[2]), u[0]*sin(x[2]), u[1]) # dx/dt = f

# Define objective function
objective = 0
for k in range(horizon):

    # Tracking error cost
    objective += mtimes([(x[:, k + 1] - ref).T, Q, (x[:, k + 1] - ref)])


    # Runge-Kutta 4 integration
    k1 = f(x[:,k],          u[:,k])
    k2 = f(x[:,k]+T_s/2*k1, u[:,k])
    k3 = f(x[:,k]+T_s/2*k2, u[:,k])
    k4 = f(x[:,k]+T_s*k3,   u[:,k])
    x_next = x[:,k] + T_s/6*(k1+2*k2+2*k3+k4)
    opti.subject_to(x[:,k+1]==x_next)

# Minimize the objective
opti.minimize(objective)

# Control input constraints
opti.subject_to(opti.bounded(-v_max, u[0,:], v_max)) # linear speed is li
opti.subject_to(opti.bounded(-omega_max, u[1,:], omega_max)) # linear spe
```

```
            opti.subject_to(x[:,0]==x0)


            # Linear acceleration constraint
            a = (u[0,:] - u_prev[0]) / T_s
            opti.subject_to(opti.bounded(-a_max, a, a_max))

            # Angular acceleration constraint
            alpha = (u[1, :] - u_prev[1]) / T_s
            opti.subject_to(opti.bounded(-alpha_max , alpha, alpha_max ))

            # Enforce relationships between u, vr, and vl
            opti.subject_to(u[0,:] == 0.5*(vl + vr))
            opti.subject_to(u[1,:] == (vr - vl)/b)


            # ---- make the solver silent ------ to see the time of computation, iter
            opts = {}
            another_opts = {}
            opts['verbose'] = False
            opts['print_time'] = False
            another_opts['print_level'] = 0

            # ---- solve NLP ------
            opti.opts = {"ipopt.tol":1e-10, "expand":True}
            opti.solver("ipopt", opts,another_opts) # set numerical backend

            self.controller = opti.to_function("controller", [u_prev, x0], [u[:,0]],

        # Call the controller to get the optimal control inputs
        return self.controller(self.u_prev, robot_states)

if __name__ == '__main__':
    node_name = "controller_node"
    rospy.init_node(node_name)
    RobotController()
    rospy.spin()
```

Anexo 9: `controller4.py`

Listing 6.9: `controller4.py`

```
import rospy
import math
import tf
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import numpy as np
import casadi
from casadi import *
import math
from message_filters import Subscriber, ApproximateTimeSynchronizer

class DualRobotController:
    def __init__(self):

       # Topics for Jetbot 0
        pub_topic_name0 = "/jetbot0/diff_drive_controller/cmd_vel"
        sub_topic_name0 = "/jetbot0/diff_drive_controller/odom"

        # Topics for Jetbot 1
        pub_topic_name1 = "/jetbot1/diff_drive_controller/cmd_vel"
        sub_topic_name1 = "/jetbot1/diff_drive_controller/odom"

        # Publishers
        self.velocity_pub0 = rospy.Publisher(pub_topic_name0, Twist, queue_size=10)
        self.velocity_pub1 = rospy.Publisher(pub_topic_name1, Twist, queue_size=10)

        # Subscribers using message_filters
        self.odom_sub0 = Subscriber(sub_topic_name0, Odometry)
        self.odom_sub1 = Subscriber(sub_topic_name1, Odometry)

        # Synchronizing the callbacks
        self.sync = ApproximateTimeSynchronizer(
            [self.odom_sub0, self.odom_sub1], queue_size=10, slop=0.1
        )
        self.sync.registerCallback(self.odometry_callback)


        # Command message
        self.velocity_msg0 = Twist()
        self.velocity_msg1 = Twist()


        #controllerMovie
        self.controller = None
        self.u_prev_1 = vertcat(0,0)
        self.u_prev_2 = vertcat(0,0)
        self.ref1 = vertcat(0,0)
        self.ref2 = vertcat(0,0)
        self.car_pos = vertcat(3,3)
        self.index = 0



    def odometry_callback(self, odom0, odom1):


        #Synchronized callback that processes odometry messages from both robots.
```

```
# Extract position and orientation from Odometry message
x0 = odom0.pose.pose.position.x
y0 = odom0.pose.pose.position.y
orientation_q0 = odom0.pose.pose.orientation

x1 = odom1.pose.pose.position.x
y1 = odom1.pose.pose.position.y
orientation_q1 = odom1.pose.pose.orientation

# Convert quaternion to Euler angles (roll, pitch, yaw)
quaternion0 = [orientation_q0.x, orientation_q0.y, orientation_q0.z, orientat
_, _, theta0 = tf.transformations.euler_from_quaternion(quaternion0)
# Log the extracted values
rospy.loginfo("ROobo 1 - Position: x = %.2f, y = %.2f, Theta = %.2f", x0, y0,


# Convert quaternion to Euler angles (roll, pitch, yaw)
quaternion1 = [orientation_q1.x, orientation_q1.y, orientation_q1.z, orientat
_, _, theta1 = tf.transformations.euler_from_quaternion(quaternion1)
# Log the extracted values
rospy.loginfo("Robo 2- Position: x = %.2f, y = %.2f, Theta = %.2f", x1, y1, t


# State vector as a column vector (6x1)
state_vector = np.array([[x0], [y0], [theta0], [x1], [y1], [theta1]])


# Call the controllerMovie function
control_output = self.controllerMovie(state_vector)
self.u_prev_1 = control_output[0:2,0]
self.u_prev_2 = control_output[2:4,0]




# Assign linear and angular speeds to the Twist message
self.velocity_msg0.linear.x = control_output[0, 0]  # Linear speed
self.velocity_msg0.angular.z = control_output[1, 0]  # Angular speed
self.velocity_msg0.linear.x = control_output[2, 0]  # Linear speed
self.velocity_msg0.angular.z = control_output[3, 0]  # Angular speed

# Log the control actions without non-ASCII characters
rospy.loginfo("Acoes de controlo para o Robo 1: Velocidade Linear={:.2f} m/s,
rospy.loginfo("Acoes de controlo para o Robo 2: Velocidade Linear={:.2f} m/s,


# Publish the command
self.velocity_pub0.publish(self.velocity_msg)
self.velocity_pub1.publish(self.velocity_msg)
```

```
def controllerMovie(self, robot_states):

    if self.controller == None:


        b = 0.1 #m    melhor ir saber agora que temos acesso aos carros
        vmax = 5 # m/s
        amax = 1 # m/s^2
        alfamax = amax * (2/b) #rad/s


        Th = 1.2 # horizon time #Aumentei o Th porque agora com runge kutta 4
        Nh = 6 # number of control intervals
        dt = Th/Nh # length of a control interval
        nx = 3 # number of states
        nu = 2 # number of inputs
        ny = 2 # number of outputs


        Ts = 0.2
        T = 80
        numberIterations = round(T/Ts)

        time_beyond_mpc= dt*Nh*0.5 #s
        iterations_beyond_mpc = round(time_beyond_mpc/dt)

        #car beeing recorded parameters
        car_speed_x = 0.1
        car_speed_y = 0.1

        car_x0 = self.car_pos[0]
        car_y0 = self.car_pos[1]

        self.car_pos[0,:] = np.array(range(0, Nh + numberIterations + iterations_b
        self.car_pos[1,:] = np.array(range(0, Nh + numberIterations + iterations_b

        robot1_states = robot_states[0:2]
        robot2_states = robot_states[2:4]

        C = DM([[1, 0, 0],[0, 1, 0]])

        tau = self.desfasamento_obtencao(robot1_states, robot2_states, vertcat(ca

        self.ref1[0,:] = carPath_x + 2*cos(range(len(carPath_x))*dt*2*pi/40 + tau
        self.ref1[1,:] = carPath_y + 2*sin(range(len(carPath_x))*dt*2*pi/40 + tau

        self.ref2[0,:] = carPath_x + 2*cos(range(len(carPath_x))*dt*2*pi/40 + tau
        self.ref2[1,:] = carPath_y + 2*sin(range(len(carPath_x))*dt*2*pi/40 + tau
opti = Opti() # Optimization problem

# ---- decision variables ---------
X = opti.variable(nx,Nh+1) # state trajectory
state_x   = X[0,:] # position x
state_y = X[1,:] # position y
state_theta   = X[2,:] # orientation
U = opti.variable(nu,Nh)   # control trajectory (linear and angular speed
input_linear_vel = U[0,:]
input_angular_vel = U[1,:]
Vl = opti.variable(1,Nh)
Vr = opti.variable(1,Nh)
Ref = opti.variable(ny,Nh+1)
ref_x = Ref[0,:]
ref_y = Ref[1,:]
OldU = opti.variable(nu,1)
PosVehicle = opti.variable(2,Nh+1)
Pos_x_Vehicle = PosVehicle[0,:]
Pos_y_Vehicle = PosVehicle[1,:]
X0_parameter = opti.parameter(nx,1)
OldU_parameter = opti.parameter(nu,1)
Ref_x_parameter = opti.parameter(1,Nh+1)
Ref_y_parameter = opti.parameter(1,Nh+1)
Pos_x_Vehicle_parameter = opti.parameter(1,Nh+1)
Pos_y_Vehicle_parameter = opti.parameter(1,Nh+1)

# ---- objective          ---------
Q = 7
R = 1
Y = mtimes(C,X) # para multiplicar matrizes
difyr = Y - Ref

# ---- dynamic constraints ---------
f = lambda x,u: vertcat(u[0]*cos(x[2]), u[0]*sin(x[2]), u[1]) # dx/dt = f
objective = 0
for k in range(Nh): # loop over control intervals

   relativePos = Y[:,k] - PosVehicle[:,k]
   crossPro = self.MX_skew(vertcat(f(X[:,k], U[:,k])[0:2],0))*vertcat(rel
   objective = objective + sumsqr(U[:,k])*R + sumsqr(difyr[:,k])*Q + cros
   # Runge-Kutta 4 integration
   k1 = f(X[:,k],          U[:,k])
   k2 = f(X[:,k]+dt/2*k1, U[:,k])
   k3 = f(X[:,k]+dt/2*k2, U[:,k])
   k4 = f(X[:,k]+dt*k3,    U[:,k])
   x_next = X[:,k] + dt/6*(k1+2*k2+2*k3+k4)
   opti.subject_to(X[:,k+1]==x_next) # close the gaps
objective = objective + sumsqr(difyr[:,k+1])*Q
opti.minimize(objective)
# ---- path constraints -----------
```

```
    #assume-se que ele s   anda no sentido anti_hor rio
    opti.subject_to(opti.bounded(0,input_linear_vel,vmax)) # linear speed is
    ##########################################################################
    # ---- boundary conditions --------
    opti.subject_to(state_x[0]==X0_parameter[0])
    opti.subject_to(state_y[0]==X0_parameter[1])
    opti.subject_to(state_theta[0]==X0_parameter[2])
    opti.subject_to(ref_x[:]==Ref_x_parameter)
    opti.subject_to(ref_y[:]==Ref_y_parameter)
    opti.subject_to(OldU[0]==OldU_parameter[0])
    opti.subject_to(OldU[1]==OldU_parameter[1])
    opti.subject_to(Pos_x_Vehicle[:]==Pos_x_Vehicle_parameter)
    opti.subject_to(Pos_y_Vehicle[:]==Pos_y_Vehicle_parameter)
    # ---- misc. constraints ----------
    Us = horzcat(OldU,U)
    UsDiff = casadi.diff(Us,1,1)
    opti.subject_to(opti.bounded(-amax*dt,UsDiff[0,:],amax*dt)) # linear spee
    opti.subject_to(opti.bounded(-alfamax*dt,UsDiff[1,:],alfamax*dt)) # linea
    opti.subject_to(input_linear_vel == 0.5*(Vl + Vr))
    opti.subject_to(input_linear_vel == (Vr - Vl)/b)
    # ---- initial values for solver ---
    opti.set_initial(input_linear_vel, 1) #este valor tamb m deveria dar par
    opti.set_initial(input_angular_vel, 1)
    # ---- make the solver silent ------ to see the time of computation, iter
    opts = {}
    another_opts = {}
    opts['verbose'] = False
    opts['print_time'] = False
    another_opts['print_level'] = 0
    # ---- solve NLP ------
    opti.opts = {"ipopt.tol":1e-10, "expand":True}
    opti.solver("ipopt", opts,another_opts) # set numerical backend
    controller = opti.to_function("controller", [OldU_parameter, X0_parameter

if norm_2(mtimes(C, robot_states[0:2])- self.ref1[:,self.index]) > 3:
    k1 = self.index + iterations_beyond_mpc
else:
    k1 = self.index
if norm_2(mtimes(C, robot_states[2:4])- self.ref2[:,self.index]) > 3:
    k2 = self.index + iterations_beyond_mpc
else:
    k2 = self.index

aux1 = np.array(range(k1,k1 + Nh +1))
aux2 = np.array(range(k2,k2 + Nh +1))
u1 = self.controller(self.u_prev_1, robot_states[0:2], self.ref1[0,aux1], sel
u2 = self.controller(self.u_prev_2, robot_states[2:4], self.ref2[0,aux2], sel
self.index += 1
return vertcat(u1, u2)
```

```
    def desfasamento_obtencao(x01,x02, pos_i, C):
        tau = math.atan((x01[1]-x02[1])/(x01[0]-x02[0]))
        aux1 = norm_2(mtimes(C,x01) - 2*vertcat(cos(tau), sin(tau)) - pos_i)
        aux2 = norm_2(mtimes(C,x02) - 2*vertcat(cos(tau+pi), sin(tau+pi)) - pos_i)
        aux3 = norm_2(mtimes(C,x01) - 2*vertcat(cos(tau+pi), sin(tau+pi)) - pos_i)
        aux4 = norm_2(mtimes(C,x02) - 2*vertcat(cos(tau), sin(tau)) - pos_i)

        if aux1 + aux2 < aux3 + aux4:
            return tau
        else :
            return tau+pi

    def MX_skew(x):
        # Obtain skew-symmetric matrix from vector
        n = MX.size(x)[0]
        if n == 3:
            X = vertcat(    horzcat(0,       -x[(2)],     x[(1)]), horzcat(x[(2)],
0,       -x[(0)]),  horzcat(-x[(1)],     x[(0)],     0)      );
        elif n == 1:
            X = vertcat(    horzcat(0,       -x[(0)]), horzcat(x[(0)],
0)      );
        else:
        # Use rospy.logerr for error handling
            rospy.logerr("SKEW function not implemented for input dimensions other th
            return None  # Or raise an error depending on your preference
        return X

if __name__ == '__main__':
    node_name = "dual_robot_controller"
    rospy.init_node(node_name)
    DualRobotController()
    rospy.spin()
```