ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Artificial Intelligence in Industry

PUSHING CARS' LIMITS: EXPLORING AUTONOMOUS TECHNOLOGIES IN THE FORMULA SAE DRIVERLESS COMPETITION

CANDIDATE
Edoardo Fusa

SUPERVISOR
Prof. Michele Lombardi

Academic year 2024-2025

Session 1st

To my future self.

Abstract

Questa tesi illustra il progetto e l'implementazione di uno stack software completo per un veicolo elettrico Formula SAE Driverless di prima generazione, stabilendo un'architettura di base per la navigazione su tracciati sconosciuti delimitati da coni. L'architettura del sistema si fonda su classici algoritmi di robotica, esplorando al contempo l'integrazione di moderne tecniche di machine learning.

La pipeline di percezione utilizza una stereocamera con un rilevatore di oggetti YOLOv11n addestrato su misura e un algoritmo di stereo matching potenziato da ORB per ricostruire l'ambiente 3D del tracciato. Per il controllo, è stato formulato un modello dinamico del veicolo che funge da nucleo per un framework di controllo predittivo basato su modello non lineare (NMPC) progettato con il toolkit `acados`.

Un contributo chiave di questo lavoro è l'esplorazione di una strategia di controllo ibrida basata sul Reinforcement Learning. Un agente Proximal Policy Optimization (PPO) è stato addestrato per eseguire il tuning online degli iperparametri di un controllore PD di tracciamento della traiettoria, combinando il machine learning con un framework classico e interpretabile.

La validazione sperimentale ne evidenzia l'efficacia potenziale, dimostrando che l'agente PPO apprende con successo il compito di ottimizzazione e mettendo al contempo in luce i limiti di stabilità del controller classico sottostante. Questo lavoro stabilisce una solida base software per le corse autonome e fornisce indicazioni chiare e basate sui dati per lo sviluppo futuro di percezione e controllo ad alte prestazioni.

Abstract

This thesis details the design and implementation of a complete software stack for a first-generation Formula SAE Driverless electric vehicle, establishing a foundational architecture for navigating unknown, cone-delineated tracks. The system's architecture is built upon a foundation of classical robotics algorithms while exploring the integration of modern machine learning techniques.

The perception pipeline utilizes a stereocamera with a custom-trained YOLOv11n object detector and an ORB-enhanced stereo matching algorithm to reconstruct the 3D track environment. For control, a dynamic vehicle model was formulated to serve as the predictive core for a Non-linear Model Predictive Control (NMPC) framework designed with the `acados` toolkit.

A key contribution of this work is the exploration of a hybrid control strategy using Reinforcement Learning. A Proximal Policy Optimization (PPO) agent was trained to perform online hyperparameter tuning of a PD path-following controller, combining machine learning with a classical, interpretable framework.

Experimental validation shows the potential efficacy of the training setup and demonstrates that the PPO agent successfully learns its optimization task, while also highlighting the stability limits of the underlying classical controller. This work establishes a complete software foundation for autonomous racing and provides clear, data-driven insights for future development in high-performance perception and control.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Autonomous driving has emerged as one of the most transformative technologies of the 21st century, promising to revolutionize transportation by enhancing safety, efficiency, and accessibility. The development of a fully autonomous vehicle is an immensely complex undertaking, encompassing a wide array of research fields from robotics and computer vision to control theory and artificial intelligence. To accelerate innovation in this domain, a number of high-profile autonomous racing competitions have been established. These events provide a challenging yet constrained environment, pushing the boundaries of perception, planning, and control under extreme conditions and serving as an ideal testbed for new technologies.

This thesis addresses the multifaceted challenge of developing a complete autonomous driving system within the specific context of the Formula SAE (FSAE) Driverless competition. The core objective of this competition is to design, build, and program a race car capable of navigating an unknown track, delineated by colored traffic cones, at the highest possible speed. Successfully achieving this goal requires a tightly integrated software stack that can reliably perform the fundamental tasks of an autonomous system: it must perceive the environment to build a map of the track, plan an optimal trajectory through that map, and execute that trajectory with high precision by controlling the vehicle's actuators.

The work presented in this thesis details the design, implementation, and validation of such a software stack for our team's first electric driverless vehicle. Our approach is built upon a foundation of robust, well-understood algorithms, while also exploring the integration of modern machine learning techniques to enhance performance. For perception, we developed a complete stereocamera pipeline that uses a custom-trained YOLOv11n object detector to identify track cones and a multi-stage matching algorithm, refined with ORB features, to reconstruct their 3D positions. For control, we formulated a high-fidelity dynamic bicycle model, validated against professional simulation software, to serve as the predictive core for a Non-linear Model Predictive Controller (NMPC) implemented using the `acados` toolkit.

Beyond these classical methods, this thesis also presents a novel investigation into combining traditional control with modern AI. We explore a framework where Reinforcement Learning, specifically the Proximal Policy Optimization (PPO) algorithm, is used not for direct end-to-end control, but to perform online hyperparameter tuning of a classical PD path-following controller. This hybrid approach aims to leverage the optimization power of RL without sacrificing the safety and interpretability of a well-defined control system.

The primary contributions of this work are the creation of a complete, functional software architecture for an autonomous race car, including a robust system management framework; the development and validation of a stereocamera-based perception system that demonstrates the value of feature-based matching for 3D accuracy; the formulation of a dynamic vehicle model and an NMPC framework suitable for high-performance driving; and a practical exploration of using Reinforcement Learning to enhance classical controllers, providing valuable insights into the opportunities and challenges of this emerging paradigm.

This thesis is structured as follows. Chapter 2 details the context of the problem, describing the FSAE Driverless competition, its dynamic events,

and the core software requirements. Chapter 3 provides the necessary theoretical background, covering the key hardware and software principles, including LiDAR, stereocameras, vehicle modeling, and the fundamentals of MPC and Reinforcement Learning. Chapter 4 reviews the state-of-the-art by examining related work from other autonomous racing competitions and in the specific field of combining MPC with reinforcement learning. Chapter 5 presents the core technical contributions of this work, detailing the design, implementation, and experimental validation of our perception pipeline, vehicle models, and control strategies. Finally, Chapter 6 concludes the thesis by summarizing the work, acknowledging its limitations, and outlining promising directions for future research.

Chapter 2

Context

Autonomous driving represents a broad and rapidly evolving field, encompassing a diverse range of complex challenges. This thesis addresses these challenges within the specific context of the Formula SAE (FSAE) Driverless competition. In this international competition, university teams design, build, and develop both the hardware and software for a prototype race car. The vehicle's performance is evaluated across four distinct dynamic events, or *missions*, which are conducted as time trials without direct competitors on the track simultaneously [9].

The competition environment is defined by colored traffic cones that the vehicle must perceive to navigate the course. A critical challenge across most events is that the exact track layouts are unknown beforehand, requiring the autonomous system to perform robustly in unseen environments.

The work presented in this thesis was developed as part of the UniBo Motorsport team, specifically for its Driverless Division. The software stack[1] is implemented on our prototype, named Athena, which, in accordance with competition regulations, is built upon a newly designed chassis for the current season, with the exception for teams that develop a Driverless car for the first time (which is our case) [9]. The primary objective is to develop a system

[1] Throughout this work, terms such as *codebase*, *stack*, *software*, *solution*, and *project*, unless otherwise specified, refer to the Formula SAE Driverless software developed by the UniBo Motorsport team.

capable of achieving the best possible performance in the competition.

Figure 2.1: Our prototype Athena.

2.1 The FSAE Driverless Dynamic Events

The competition comprises four dynamic events, each designed to test specific aspects of the vehicle's performance. A common rule for all events is that the maximum distance between two consecutive cones of the same color on a track boundary is five meters.

1. **Acceleration:** This event evaluates the vehicle's maximum longitudinal acceleration from a standstill. The track is a 75-meter straight line marked by cones for visual guidance. The vehicle must accelerate from a designated starting position, cross the finish line as quickly as possible, and come to a complete stop within a defined braking zone. The track layout is predefined and known, allowing for offline optimization of acceleration and braking profiles.

2. **Skidpad:** This event measures the vehicle's quasi-steady-state cornering capability at maximum lateral acceleration. The track consists of two pairs of concentric circles forming a figure-eight pattern with precise, known dimensions. The vehicle must navigate a 3-meter wide corridor delineated by inner and outer cones, completing one timed lap on the right-hand circle and one on the left-hand circle. Like Acceleration, the Skidpad layout is fixed and known in advance.

3. **Autocross:** This event assesses the vehicle's overall handling and agility on a short, complex circuit. The Autocross is a single-lap event on a closed-loop track whose layout is unknown to teams before the competition. The track is defined by a series of blue cones marking the left boundary and yellow cones marking the right boundary. The layout typically features a combination of short straights, chicanes, hairpin turns, and slaloms.

4. **Trackdrive:** This is the main endurance event, designed to test the reliability, consistency, and sustained performance of the autonomous system. The event consists of ten consecutive laps on a closed circuit similar in style to the Autocross track. The layout, with a typical lap length between 200 m and 500 m, is also unknown beforehand and is delimited by blue and yellow cones.

2.2 Generic Autonomous System Architecture

Any autonomous vehicle, including the FSAE Driverless prototype, is built around a core set of interacting components that operate in a continuous feedback loop.

First, the vehicle must be equipped with a suite of **sensors** to perceive its environment. Common examples include LiDARs, cameras, RADARs, Global Navigation Satellite Systems (GNSS), Inertial Measurement Units (IMUs),

and wheel encoders.

The data from these sensors is processed by an onboard **computing unit**, which makes informed decisions about the vehicle's next actions. Unlike in conventional computing, the physical characteristics of this unit—its weight, dimensions, and power consumption—are critical design constraints. These factors directly impact the vehicle's battery life, center of mass, handling dynamics, and even the necessity of processing the data coming from the IMU.

The decisions from the computing unit are formulated as **control commands**, typically target values for speed and steering angle. These commands are sent to the vehicle's **actuators** (e.g., steering motor, throttle controller), which physically execute the desired actions.

This entire process operates as a continuous **feedback loop**: perception informs planning, planning generates commands, actuation executes them, and the resulting change in the vehicle's state is captured by the sensors in the next cycle, thereby closing the loop. The loop continues until a stop condition is met.

2.3 Core Software Requirements

To successfully operate within this feedback loop, the software algorithms must adhere to several critical requirements.

- **Real-time:** The system must operate in real-time. This implies that the entire sense-plan-act cycle must complete within a strict time budget, typically dictated by the refresh rate of the primary sensors (e.g., a LiDAR operating at 20 Hz imposes a 50 ms budget). Exceeding this budget leads to processing lag, where decisions are based on outdated information, potentially causing control instability. This constraint, combined with actuator latency, necessitates that all critical code is written in a high-performance, compiled language such as C++ or Rust.

- **Readable:** High code readability is essential for effective debugging and rapid troubleshooting, which is critical during on-track testing and competition events. Understandable code allows team members to quickly identify and resolve malfunctions.

- **Maintainable:** The codebase must be maintainable to facilitate iterative development, adaptation to new hardware (e.g., sensor upgrades), and reuse across different vehicle prototypes or competition seasons.

- **Reliable:** Reliability is paramount for competitive success. A reliable system exhibits consistent and deterministic behavior across repeated trials and in varying, unseen environments. Achieving this is often the most significant challenge.

2.4 Development and Validation Challenges

Developing and validating an autonomous system that meets these requirements is a complex, multidisciplinary endeavor, requiring expertise in fields ranging from low-level programming and control theory to machine learning and vehicle dynamics.

While Deep Learning (DL) has shown remarkable success in specific domains like object detection and Reinforcement Learning, it is not a panacea for all autonomous driving challenges. *For robust deployment, it is crucial that such technologies are not only reliable but also interpretable or explainable.* Before being trusted in a real-world scenario, all algorithms, whether based on DL or classical methods, must undergo rigorous testing and validation.

The validation process for this problem is non-trivial, but several methods can aid development.

- **Simulation:** A high-fidelity simulation environment is an indispensable tool. It allows for the early detection of logical and architectural bugs in a safe, cost-effective, and repeatable manner. If the simulation

accurately models vehicle physics, it can also be used to test and tune control algorithms with greater confidence.

- **Data Replay:** This technique involves recording real-time sensor data from the physical vehicle (e.g., LiDAR point clouds and camera images during human driving) and ”replaying” it to the software stack. This provides a perfect simulation for perception algorithms that rely only on sensor data. However, it is an open-loop test; the control outputs generated by the algorithm cannot influence the vehicle's trajectory in the playback, as that is dictated by the original recording.
- **On-Vehicle Testing:** The ultimate validation is deploying the solution on the real prototype. While successful operation is a strong indicator of a robust system, this method presents challenges for quantitative evaluation due to the difficulty in establishing a ground truth. For instance:
  - To measure the accuracy of map generation, one would need to precisely survey the position of every cone on the track and compare it against the map generated by the system. This process is prohibitively time-consuming to repeat for every new track layout.
  - To measure the accuracy of the vehicle's localization, an external, high-precision motion capture system would be required.

2.5 Macro-Areas of Any Autonomous Stack

The software stack for an autonomous race car is typically organized into three interconnected macro-areas.

2.5.1 Software Infrastructure

This area encompasses the foundational elements that support the entire software stack. It includes the choice of development frameworks (e.g., Robot

Operating System 2 - ROS 2 [16]), version control systems (e.g., Git), and continuous integration (CI) pipelines. While some elements like CI do not run on the vehicle, they are vital to the development process. An example of an infrastructure component on Athena is the state machine responsible for managing the startup, execution, and shutdown of all necessary processes for a selected mission, including anomaly detection and failure recovery. Handling sensor drivers and framework updates also falls under this area.

2.5.2 Perception

The perception system is responsible for interpreting raw sensor data to build a coherent understanding of the environment. Its primary tasks are to create a map of the track and to continuously estimate the vehicle's position and orientation (localization) within that map. An example in our system is the LiDAR pipeline, which detects the 3D position of traffic cones. This is achieved by leveraging a priori knowledge of cone dimensions and by applying simple techniques to mitigate point cloud distortion caused by vehicle motion at high speeds.

2.5.3 Motion Planning

Motion planning utilizes the world model from the perception system to compute a safe and efficient path for the vehicle. This process can generate:

- **Local Trajectories:** When a complete map is not yet available, the system can use the positions of nearby visible cones to generate a short-term trajectory that keeps the vehicle within the track boundaries.
- **Global Trajectories:** Once a complete map of the circuit is built, the system can compute a global trajectory, such as the track centerline or a lap-time-optimized racing line.

These planned trajectories are then translated into executable commands by a controller. Controllers can be broadly categorized as:

- **Geometric Controllers:** These algorithms use the vehicle's kinematic properties to determine a target steering angle (δ) to follow a path. They typically do not compute throttle or torque commands.
- **Physics-based Controllers:** These more advanced controllers also consider the vehicle's dynamic properties (e.g., mass, inertia, tire forces). This allows for more stable, precise, and efficient control, especially near the limits of handling.

Chapter 3

Background

This chapter provides an overview of some of the key hardware and software components that form the foundation of our autonomous system.

3.1 Autonomous System Unit (ASU)

The Autonomous System Unit (ASU) is the central onboard computer that runs the high-level software stack, including perception, mapping, and motion planning. For our system, we chose to build the ASU using commercial components rather than a specialized embedded platform like an NVIDIA Jetson.

This decision was driven by several key factors. Commercial components provide complete control over the software environment, allowing us to install a standard Linux distribution (Ubuntu 24.04) and manage dependencies freely. This approach also benefits from extensive community support and documentation, which simplifies troubleshooting. Furthermore, a desktop-class CPU offers significantly higher single-core performance compared to typical embedded ARM processors, which is advantageous for many of our algorithms.

The primary trade-offs for this increased performance and flexibility are greater physical dimensions, higher weight, and increased power consumption. A critical design challenge was thermal management. Since the vehicle

must be capable of operating in the rain, traditional air cooling with open vents was not a viable option due to the risk of water infiltration. To solve this, we designed a custom, water-resistant plexiglass case for the ASU, which necessitates the use of a dedicated water-cooling system.

The components of the ASU and its cooling system are listed below.

ASU Components:

- **Motherboard:** ASUS Z790-I
- **CPU:** Intel Core i7-12700K
- **RAM:** G.SKILL 16GB DDR5 7200MHz
- **GPU:** ASUS DUAL-RTX4060TI-O8G
- **SSD:** Samsung MZ-V9P1T0BW 1TB NVMe
- **Power Supply Unit:** HDPLEX 500W HiFi DC-ATX

ASU Cooling System:

- **Radiator:** Alphacool 14172
- **Water Pump:** WCP D5-VARIO
- **Reservoir:** Stealkey UNI 80

3.2 Vehicle Control Unit (VCU)

The Vehicle Control Unit (VCU) is a real-time, programmable microcontroller that serves as the low-level interface between the ASU and the car's physical hardware. Our vehicle uses the Miracle², a specialized automotive control platform provided by Alma Automotive.

The core of the Miracle² is a National Instruments System-on-Module (SOM), which combines a dual-core ARM processor for running logic with a Xilinx Artix-7 FPGA for handling high-speed, parallel input/output tasks. This architecture makes it ideal for the demands of real-time control and data acquisition. The unit is programmed using the NI LabVIEW graphical toolchain.

In our system, the VCU acts as the central communication gateway. It

connects directly to critical vehicle components via standard protocols like CAN and Ethernet, managing devices such as:

- The motor inverters
- The GPS and steering angle sensors
- The steering wheel dashboard
- The autonomous steering and braking actuators

The communication between the ASU and the VCU is handled by a robust, duplex TCP connection over Ethernet. This link creates a clear hierarchy of control:

- The **ASU sends high-level commands** (e.g., ”set steering angle to 0.1 rad,” ”set rear torque to 50 Nm”) to the VCU.
- The **VCU translates these commands into low-level electrical signals** for the actuators and simultaneously **sends essential feedback signals** (e.g., current steering angle, GPS data) back to the ASU.

This separation of tasks allows the ASU to focus on complex decision-making, while the VCU handles the safety-critical, real-time control of the vehicle’s hardware.

3.3 LiDAR Sensors

LiDAR, which stands for Light Detection and Ranging, is a crucial active sensor technology used for environment perception in autonomous systems. Unlike passive sensors like cameras that rely on ambient light, LiDAR systems actively illuminate the environment with their own light source, making them highly effective in a wide range of lighting conditions.

3.3.1 Principle of Operation: 2D LiDAR

The fundamental working principle of a LiDAR sensor is based on the concept of Time-of-Flight (ToF). As illustrated in Figure 3.1, the process is as follows:

1. The LiDAR unit emits a brief, focused pulse of laser light.
2. This pulse travels through the air at the speed of light (c) until it strikes an object.
3. A portion of the light is reflected off the object's surface and travels back to the LiDAR's detector.
4. The sensor measures the total time (t) taken for this round trip.

The distance to the object can then be calculated using a simple formula. Since the measured time accounts for the journey to the object and back, the one-way distance is half of the total distance traveled:

$$\text{distance} = \frac{c \times t}{2} \tag{3.1}$$

A 2D LiDAR sensor performs this measurement thousands of times per second while rotating around a central axis. This action sweeps the laser across a single horizontal plane, generating a 360-degree "slice" of the surrounding environment as a series of distance measurements. Classic examples of such sensors, often used in robotics, include devices made by SICK and Hokuyo.

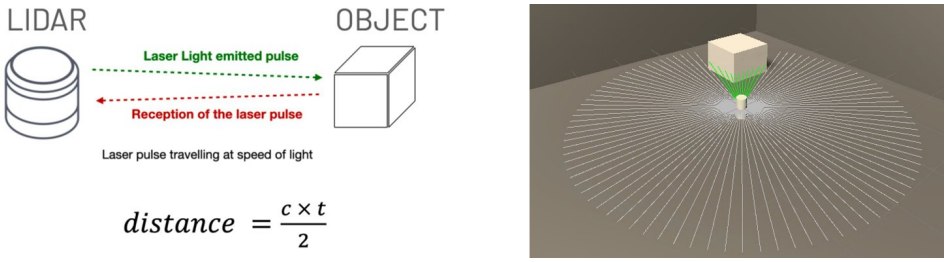


Figure 3.1: The Time-of-Flight (ToF) principle of a LiDAR sensor (left) and the resulting single plane of 2D scan data (right).

3.3.2 Extension to 3D LiDAR

The principle behind 3D LiDAR is a direct extension of its 2D counterpart. The key difference, as shown in Figure 3.2, is that a 3D LiDAR does not have

just one laser emitter but an array of them. Each laser in the array is aimed at a slightly different, fixed vertical angle.

As this entire array rotates, it creates not just one, but a stack of horizontal scanning planes. This process rapidly builds a full three-dimensional map of the environment. The output is no longer a simple 2D scan but a rich **point cloud**, where each point represents a single laser reflection and has its own (x, y, z) coordinate in space.

This 3D information is invaluable for autonomous racing. While a 2D LiDAR could detect the presence of a cone on its scanning plane, a 3D LiDAR captures the cone's full shape and height. This allows algorithms to more reliably distinguish cones from other objects and to accurately determine their position on the track, which is fundamental for navigation and mapping.

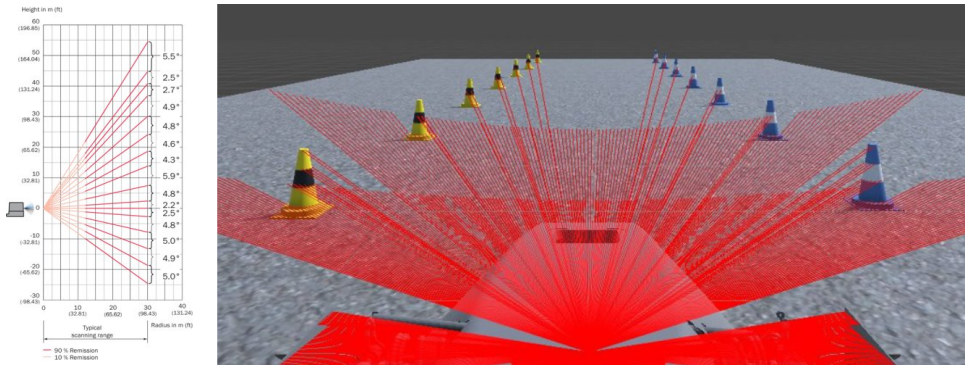


Figure 3.2: A diagram showing the multiple vertical layers of a 3D LiDAR (left) and a visualization of how these layers scan the track environment to create a 3D point cloud (right).

3.4 Stereocameras

A stereocamera is a type of sensor that uses two or more lenses to simulate human binocular vision. By capturing two separate images of the same scene from slightly different viewpoints, the system can perceive depth and reconstruct the 3D structure of its environment.

3.4.1 Principle of Operation: Triangulation and Disparity

The principle behind stereo vision is analogous to a simple human experience: if you hold a finger in front of your face and view it by closing one eye and then the other, the finger appears to shift against the background. This apparent shift is known as parallax.

In computer vision, this horizontal shift in an object's pixel position between the left and right images is called *disparity*. As illustrated in Figure 3.3, the magnitude of this disparity is inversely proportional to the object's distance from the cameras:

- **Nearby objects** exhibit a **large** disparity.
- **Distant objects** exhibit a **small** disparity.

Once the camera system is calibrated, the depth (or distance) to any point in the scene can be calculated through triangulation. This relationship is captured by the following formula:

$$\text{Depth} = \frac{f \cdot \text{baseline}}{\text{disparity}} \tag{3.2}$$

where:

- **Focal Length (f):** An intrinsic property of the camera lenses, known from calibration.
- **Baseline:** The fixed, precisely known distance between the centers of the two camera lenses.
- **Disparity:** The measured difference (in pixels) of an object's horizontal position between the left and right images.

To calculate depth, a stereo system must first solve the **correspondence problem**: finding the same point in both the left and right images to measure

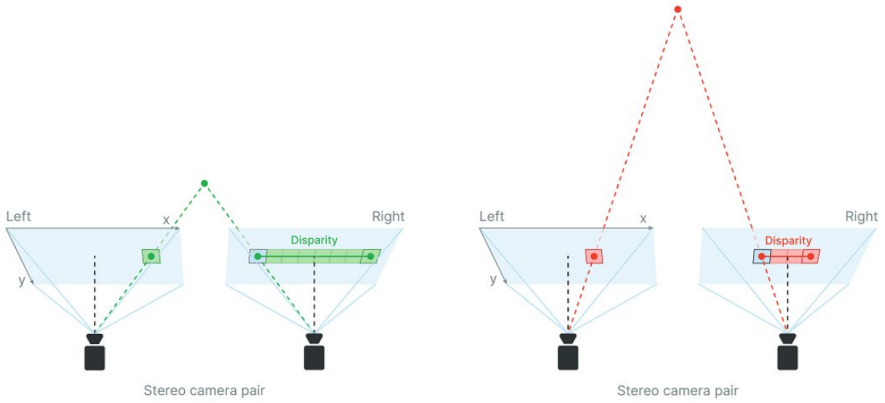


Figure 3.3: The principle of stereo vision. The disparity, or shift in an object's pixel position, is larger for closer objects (green) and smaller for farther objects (red).

its disparity. This matching process is a computationally intensive task. Unlike active sensors like LiDAR, stereocameras are passive and rely on ambient lighting, making them less effective in low-light or uniform environments where distinct features are scarce. However, their primary advantage is that they provide rich color and texture information in addition to depth, which is invaluable for tasks like object classification. Our system uses a ZED 2i, a popular example of a pre-calibrated, industrial stereocamera.

3.5 ROS 2

ROS (Robot Operating System) 2 [16] is a widely used, open-source framework in robotics that provides a standardized way for different software components to communicate. It operates on a system of *nodes*, which are individual processes performing specific tasks (e.g., processing LiDAR data or calculating control commands).

These nodes exchange information through a publish-subscribe messaging system organized around named channels called **topics**. A node can act as a *publisher* by sending data to a specific topic, or as a *subscriber* by listening for data on that topic. The units of data exchanged on these topics are called **messages**, which are strongly-typed data structures. For instance, a perception

node might publish a message containing an array of detected cone positions to a */cone_map* topic, while the controller node subscribes to this topic to receive the map data.

In addition to this continuous data streaming model, ROS 2 provides a communication mechanism for remote procedure calls called **services**. Services operate on a synchronous request-response model, similar to a client-server architecture. A node acting as a *service server* can offer a specific function, which another node, the *service client*, can call. When a client calls a service, it sends a request message and waits for the server to process it and return a response message. This two-way communication is ideal for tasks that require a direct confirmation, such as asking a state management node to switch the car into ”autonomous mode” and waiting for a success or failure response before proceeding.

While ROS 2 provides a rich library of standard message types (e.g., for point clouds, images, and odometry), a key feature is the ability for developers to define **custom messages**. This allows for the creation of tailored data structures that precisely fit the application’s needs, such as a custom message to convey a complete vehicle state or a specific control command. This modular, message-based architecture is exceptionally well-suited for the high-frequency, asynchronous data exchange required for real-time autonomous control.

3.6 Stereo Matching

After establishing the principles of stereo vision, we now focus on the core computational task required to extract depth information: **stereo matching**. The goal of any stereo matching algorithm is to solve the correspondence problem: that is, for a given pixel in the left image, to find the exact same corresponding pixel in the right image. Once this correspondence is found, the disparity can be calculated, and from that, the depth can be determined.

3.6.1 Image Rectification

Before a matching algorithm can be run efficiently, the pair of stereo images must undergo a crucial preprocessing step called **rectification**. In a raw, uncalibrated pair of images, a point visible in the left image could lie anywhere along a diagonal line (known as an epipolar line) in the right image. Searching along this diagonal line for every pixel would be computationally very slow.

Image rectification is a transformation process that warps both images such that all epipolar lines become perfectly horizontal and aligned. This means that a point appearing at pixel coordinates (x, y) in the rectified left image is guaranteed to appear on the same horizontal scanline, y, in the rectified right image. This powerful simplification reduces the search for a corresponding point from a two-dimensional problem to a much faster one-dimensional one.

3.6.2 Matching Algorithms

Once the images are rectified, the algorithm can proceed to find correspondences along each horizontal scanline. Modern matching algorithms typically do not match individual pixels, as a single pixel provides too little information and can be highly ambiguous. Instead, they operate on small windows or ”patches” of pixels (e.g., a 7x7 square) around a point of interest.

To determine the best match for a patch from the left image, the algorithm slides a candidate patch along the corresponding scanline in the right image and computes a similarity score at each position. A common and efficient metric for this is the **Sum of Absolute Differences (SAD)**. This metric is calculated as:

$$\text{SAD} = \sum_{i,j} |I_L(i, j) - I_R(i, j)| \tag{3.3}$$

where I_L and I_R are the intensity values of the pixels within the left and right patches, respectively. The position that yields the lowest SAD score is considered the best match.

3.6.3 The Disparity Map

The final output of the stereo matching process is a **disparity map**. This is a new, single-channel image where the value of each pixel corresponds to the calculated disparity for that location. High-intensity pixels represent large disparities, indicating objects that are close to the camera. Conversely, low-intensity pixels represent small disparities, indicating objects that are far away. This disparity map serves as a direct, dense representation of the scene's 3D structure and can be converted into a depth map using the formula described in Section 3.4.

3.6.4 Common Challenges

While powerful, stereo matching algorithms face several inherent challenges that can affect the quality of the resulting disparity map:

- **Textureless Regions:** On surfaces with uniform color and no texture, like a blank wall or a smooth patch of asphalt, many patches will have a similarly low matching score, making it difficult to find a unique, correct correspondence.
- **Occlusions:** Some parts of the scene may be visible to one camera but hidden from the view of the other. These occluded regions have no possible correspondence, resulting in gaps or errors in the disparity map.
- **Repetitive Patterns:** Highly repetitive textures, such as a brick wall or a chain-link fence, can cause the algorithm to find multiple "good" matches, leading to incorrect disparity estimates.

3.7 Feature Matching

While dense stereo matching algorithms aim to calculate a depth value for every pixel in an image, another important technique in computer vision is **feature matching**. Instead of processing the entire image, feature matching

focuses on identifying and matching a sparse set of salient, distinctive points between two images. These points, often called keypoints or features, typically correspond to corners, blobs, or unique textural patterns in the scene.

This sparse approach offers several advantages over dense methods. It is significantly more computationally efficient, as it only processes a few hundred keypoints instead of millions of pixels. Furthermore, the algorithms used are often designed to be robust to changes in image scale, rotation, and lighting, making feature matching an ideal tool for tasks like object tracking, visual odometry, and Simultaneous Localization and Mapping (SLAM).

3.7.1 The Feature Matching Process

The process of matching features between two images can be broken down into three main steps, as illustrated in Figure 5.8.

1. Feature Detection

The first step is to identify keypoints in each image. A good keypoint is one that can be reliably detected even if the image is transformed (e.g., rotated or viewed from a different angle). Algorithms like Harris Corner Detector or FAST (Features from Accelerated Segment Test) are designed to find such stable points by analyzing local pixel intensity patterns. The output of this stage is a list of pixel coordinates for all detected keypoints in each image.

2. Feature Description

Once a keypoint is detected, a numerical ”fingerprint” called a **descriptor** is computed to represent the image patch surrounding it. This descriptor must be distinctive enough to differentiate one keypoint from another, yet robust enough to be consistent across different viewing conditions. Descriptors can range from high-dimensional floating-point vectors (like in SIFT) to compact binary strings (like in ORB).

3. Descriptor Matching

With sets of descriptors from both images, the final step is to find correspondences. The most straightforward method is **Brute-Force Matching**. For each descriptor in the first image, the algorithm compares it to every descriptor in the second image to find the one with the smallest ”distance” (e.g., Euclidean distance for float vectors or Hamming distance for binary strings). The pair with the minimum distance is considered a match. More sophisticated filtering techniques, like the ratio test, are often used to discard ambiguous or weak matches.

3.7.2 ORB: An Efficient Feature Algorithm

For real-time applications such as autonomous racing, computational efficiency is paramount. One of the most popular algorithms that balances performance and accuracy is **ORB** (Oriented FAST and Rotated BRIEF) [23]. ORB is widely used in robotics because it is effective and free from patents. As its name suggests, it combines two key components:

- **Oriented FAST for Detection:** ORB uses the FAST algorithm to quickly detect corner-like keypoints. It then improves upon standard FAST by calculating an orientation for each keypoint based on the intensity of its local neighborhood. By knowing the orientation, the system can achieve robustness to in-plane rotation.
- **Rotated BRIEF for Description:** To describe the keypoint, ORB uses a modified version of the BRIEF (Binary Robust Independent Elementary Features) descriptor. BRIEF creates a compact binary string by performing a series of simple intensity comparisons between pairs of pixels in the patch around the keypoint. This binary format is extremely fast to compute and even faster to match using the Hamming distance. ORB enhances BRIEF by ”steering” the pixel comparison pattern according

to the keypoint's orientation calculated in the first step. This makes the resulting descriptor rotation-invariant.

The combination of a fast detector and a compact, efficient descriptor makes ORB an excellent choice for our perception pipeline, where it is used to find stable correspondences for robust 3D triangulation.

3.8 Convolutional Neural Networks

Convolutional Neural Networks (CNNs or ConvNets) are a class of deep neural networks that have become the de-facto standard for tasks in computer vision. Their design is inspired by the organization of the animal visual cortex. Unlike traditional machine learning models that require hand-engineered features, the key strength of a CNN is its ability to automatically and hierarchically learn relevant features directly from raw pixel data.

3.8.1 Core Components

A CNN is constructed from a sequence of specialized layers. The most fundamental of these are the convolutional, activation, and pooling layers.

The Convolutional Layer

The convolutional layer is the core building block of a CNN. It operates by sliding a small matrix of weights, known as a **filter** or **kernel**, over the input image. At each position, the filter performs a convolution operation: an element-wise multiplication with the underlying patch of the image, followed by a summation of the results. This process produces a single value.

By sliding the filter across the entire image, a two-dimensional **feature map** is created. This feature map indicates the presence of a specific feature (e.g., a vertical edge, a corner, or a particular color) at different locations in the image. During the training process, the network learns the optimal values

for these filters, effectively teaching itself which features are important for the given task. A typical CNN will have many such filters in each convolutional layer, each one learning to detect a different feature.

The Activation Function (ReLU)

After each convolution, an activation function is applied to introduce non-linearity into the model. Without non-linearity, a deep stack of layers would behave like a single, simple layer, and would be unable to model the complex patterns found in real-world data.

The most commonly used activation function is the **Rectified Linear Unit (ReLU)**. Its function is very simple: it replaces all negative pixel values in a feature map with zero, while leaving positive values unchanged.

$$f(x) = \max(0, x) \tag{3.4}$$

ReLU is popular because it is computationally very efficient and helps mitigate some common issues that can occur during the training of deep networks.

The Pooling Layer

The pooling layer is used to progressively reduce the spatial dimensions (width and height) of the feature maps. This process, also known as down-sampling, serves two main purposes: it reduces the number of parameters and computations in the network, and it makes the learned features more robust to small variations in their position.

The most common form of pooling is **Max Pooling**. It involves sliding a small window (e.g., 2x2) over the feature map and, for each window, outputting only the maximum value. This effectively summarizes the features present in a neighborhood.

3.8.2 Typical CNN Architecture

A typical CNN architecture for image classification consists of several stacked blocks of ‘Convolution -> ReLU -> Pooling‘ layers. The initial layers learn simple, low-level features like edges and colors. Subsequent layers combine these to learn more complex features, such as textures, patterns, and eventually, object parts.

After these convolutional blocks, the final feature maps are ”flattened” into a one-dimensional vector. This vector is then fed into one or more standard **fully connected (or dense) layers**, which perform the final classification by mapping the learned high-level features to the output classes.

3.8.3 Application in Object Detection: YOLO

While the architecture described above is for classifying an entire image, object detection is a more complex task that involves both classifying *and* localizing objects with bounding boxes. One of the most influential real-time object detection models based on CNNs is **YOLO (You Only Look Once)** [21].

Unlike older, two-stage detectors that were slow, YOLO is a ”single-shot” detector. It treats object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. The core idea is to divide the input image into a grid. For each grid cell, a CNN simultaneously predicts:

- A set of bounding boxes anchored to that cell.
- A confidence score for each box, indicating the likelihood that it contains an object.
- The class probabilities for the object within each box.

This single-pass architecture makes YOLO and its subsequent versions extremely fast, enabling its use in real-time applications where high frame rates are critical. For this reason, it forms the foundation of the cone detector used

in our perception pipeline.

3.9 Vehicle Modeling

In autonomous driving, a mathematical model of the vehicle is a fundamental tool. It allows algorithms to predict how the car will respond to control inputs, which is essential for planning safe and efficient trajectories. The complexity of these models can vary greatly, but they generally fall into two main categories: kinematic models and dynamic models. The choice between them represents a critical trade-off between computational simplicity and physical accuracy.

3.9.1 Kinematic Models

Kinematic models describe the vehicle's motion based purely on geometry and velocity, without considering the forces (like tire forces or inertia) that cause the motion. They answer the question, "If the vehicle is moving at a certain speed and the wheels are turned to a certain angle, where will it be a moment later?"

The most common kinematic representation is the **kinematic bicycle model**. This model simplifies the car by collapsing the two front wheels into a single wheel at the front axle and the two rear wheels into a single wheel at the rear axle, as if it were a bicycle. Its motion is described by a simple set of equations:

$$\dot{x} = v\cos(\theta) \tag{3.5}$$

$$\dot{y} = v\sin(\theta) \tag{3.6}$$

$$\dot{\theta} = \frac{v}{L}\tan(\delta) \tag{3.7}$$

where (x, y) is the position of the vehicle's center of mass, θ is its yaw angle (heading), v is its longitudinal velocity, L is the wheelbase (the distance

between the front and rear axles), and δ is the steering angle.

The core assumption of kinematic models is the **no-slip condition**, which presumes that the wheels always move perfectly in the direction they are pointing. This assumption holds reasonably well at low speeds and during gentle maneuvers.

- **Advantages:** Kinematic models are simple, require very few parameters, and are extremely fast to compute.
- **Disadvantages:** They become inaccurate at higher speeds or during aggressive driving, where tire slip becomes significant. They cannot predict effects like understeer or oversteer.
- **Use Cases:** They are well-suited for low-speed path tracking, parking maneuvers, and as a simplified model for high-level trajectory planning.

3.9.2 Dynamic Models

Dynamic models, also called physical models, provide a more accurate representation by incorporating the principles of physics, namely Newton's second law ($F = ma$). These models consider the vehicle's mass, its moment of inertia, and the various forces acting upon it.

The **dynamic bicycle model** is a common extension of its kinematic counterpart. It uses the same simplified geometry but adds the effect of forces. A dynamic model can predict how the vehicle will accelerate, decelerate, and turn in response to forces generated by the powertrain, brakes, and, most importantly, the tires.

The key improvement is the modeling of **lateral tire forces**. Unlike the kinematic model's no-slip assumption, a dynamic model accounts for tire slip—the difference between the direction a tire is pointed and its actual direction of travel. This is crucial because it is the slip that generates the lateral forces needed for cornering. These forces are typically described by complex, nonlinear tire models, such as the Pacejka Magic Formula, which capture how

grip changes with factors like vertical load and slip angle.

- **Advantages:** Dynamic models are much more accurate, especially at high speeds and near the vehicle's handling limits. They can predict complex phenomena like weight transfer and loss of grip.
- **Disadvantages:** They are significantly more complex, require the identification of many more physical parameters (e.g., mass, inertia, tire stiffness coefficients), and are computationally more expensive.
- **Use Cases:** They are essential for high-performance controllers like Model Predictive Control (MPC) and for creating high-fidelity simulations used for testing and validation.

3.10 Path Following with a PD Controller

Once a desired path is known, a controller is needed to generate the steering commands that keep the vehicle on that path. One of the most fundamental and effective approaches for this task is a path-following controller. This section describes the Proportional-Derivative (PD) controller used in our system, which is a variation of the classic Pure Pursuit algorithm.

3.10.1 The PID Controller Framework

To understand our controller, it is first useful to understand the general Proportional-Integral-Derivative (PID) control framework. A PID controller is a ubiquitous feedback mechanism used in countless industrial control systems. Its goal is to minimize the error between a measured process variable and a desired setpoint by calculating a corrective output. It does this by combining three distinct terms:

- **Proportional (P) Term:** This term produces an output that is directly proportional to the current error. A larger error results in a larger corrective action. It provides the primary response to the error.

- **Integral (I) Term:** This term considers past errors by accumulating them over time. Its purpose is to eliminate any residual steady-state error that the proportional term alone cannot correct. However, in fast-acting systems like vehicle steering, this term can cause overshoot and instability. For this reason, it is omitted.

- **Derivative (D) Term:** This term acts on the rate of change of the error. It has a dampening effect, reducing overshoot and oscillations by ”predicting” the future error and tempering the control response accordingly.

The combined output is a weighted sum of these three terms. Since the integral term is often not needed for vehicle steering control, a simpler **PD controller** is frequently used, providing a balance of responsiveness and stability.

3.10.2 The Race Line as a Reference

The ”desired setpoint” for a racing context is not a single value but a continuous path known as the **race line**. A race line is an optimized trajectory that represents the fastest path around a given circuit. It is typically defined as a sequence of discrete points, where each point contains not only a 2D position (x, y) but also other critical information such as the path's curvature and a target velocity profile. This race line serves as the reference trajectory that the path-following controller must track. A good implementation for optimizing the race line is present in [27].

3.10.3 Our PD Path-Following Implementation

Our controller continuously calculates the necessary steering angle to follow the race line. The process is repeated at each time step and can be broken down into two main stages.

1. Finding the Target Point

Instead of trying to steer towards the closest point on the path (which can be unstable), the controller looks ahead to a ”target point” further along the race line. This is the core idea behind Pure Pursuit. The process is as follows:

1. The vehicle's current position is obtained from the localization system. Our implementation uses the position of the rear axle as the reference point for the car.

2. A target point is selected on the race line at a specific ”lookahead distance”, d, from the vehicle's current position.

3. This lookahead distance is not fixed; it is dynamically adjusted based on the vehicle's current speed. A longer lookahead distance is used at higher speeds to encourage smoother steering inputs. This is governed by two tunable parameters: a minimum lookahead distance ($d_{\text{la,min}}$) and a velocity-dependent gain (g_{la}). The final formula is:

$$d = v \cdot g_{\text{la}} + d_{\text{la,min}}$$

This target point effectively becomes the short-term goal for the controller.

2. Computing the Steering Angle

Once the target point is identified, the controller computes the steering angle (δ) required to direct the vehicle towards it.

1. The **desired yaw angle**, θ_{target}, is calculated. This is the angle of the vector pointing from the car's current position to the target point.

2. The **heading error**, e, is computed as the difference between this desired yaw angle and the car's current yaw angle, θ.

$$e = \theta_{\text{target}} - \theta$$

3. This error is then fed into a PD controller to calculate the final steering angle command:

$$\delta_{\text{target}} = k_p \cdot e + k_d \cdot \frac{de}{dt}$$

The proportional term ($k_p \cdot e$) provides the primary steering action, turning the wheels proportionally to the heading error. The derivative term ($k_d \cdot de/dt$) acts as a damper, smoothing the steering response and preventing the rapid oscillations that could arise from using a purely proportional controller. The gains k_p and k_d are the core tunable parameters that define the controller's responsiveness and stability.

3.11 Model Predictive Control (MPC)

While controllers like PD are reactive—meaning they correct errors based on the current state—Model Predictive Control (MPC) is a more advanced, proactive control strategy. Instead of just reacting to the present, MPC uses a model of the vehicle to predict its future behavior and plans an entire sequence of optimal actions over a short time horizon. This ability to "look ahead" allows it to make much more intelligent decisions, especially in complex and dynamic environments.

3.11.1 The Core Concept: Receding Horizon Control

The fundamental principle behind MPC is known as **receding horizon control**. It works in a cyclical fashion, constantly re-evaluating its plan at each time step. Imagine a human driver navigating a series of corners: they don't just look at the road immediately in front of them; they look ahead to anticipate the best line. MPC mimics this process digitally.

The process can be broken down into a few key steps:

1. **Get Current State:** The controller first measures the current state of the vehicle (e.g., its position, velocity, and heading).

2. **Predict and Optimize:** Using the vehicle's dynamic or kinematic model, the controller simulates many different possible sequences of control inputs (e.g., steering and throttle commands) over a predefined ”prediction horizon” (e.g., the next 2-3 seconds). It evaluates each resulting trajectory against a cost function to find the one optimal sequence that best achieves its goals (e.g., follows the race line, minimizes control effort, etc.) while respecting all constraints.

3. **Apply First Action:** Although the controller has planned an entire sequence of actions, it only applies the *very first* action from that optimal plan.

4. **Repeat:** At the next time step, the entire process is repeated. The horizon ”recedes” or moves forward, a new optimal plan is calculated based on the new current state, and the first step of that new plan is applied.

This constant re-planning makes MPC highly robust to disturbances and changes in the environment, as it is always correcting its plan based on the most recent information.

3.11.2 Key Components of an MPC Formulation

To implement an MPC, three core components must be defined:

- **Prediction Model:** This is the mathematical model of the system, as discussed in Section 3.9. The accuracy of the MPC's predictions is directly dependent on the fidelity of this model.

- **Cost Function:** This function mathematically defines the ”goal” of the controller. It assigns a numerical score (a cost) to a predicted trajectory, which the controller then tries to minimize. A typical cost function might penalize deviation from a reference path, aggressive control inputs, or low speeds.

- **Constraints:** These are the ”rules” that the predicted trajectory must obey. Constraints can represent physical limitations of the vehicle (e.g., maximum steering angle, maximum tire grip) or environmental rules (e.g., staying within the track boundaries).

3.11.3 Linear vs. Non-linear MPC (NMPC)

The type of prediction model used distinguishes between two main families of MPC.

- **Linear MPC (LMPC):** This approach uses a simplified, *linearized* version of the vehicle model. The main advantage is computational speed. The resulting optimization problem is a Quadratic Program (QP), which can be solved extremely quickly, even on less powerful hardware. However, since a linear model is only an approximation, LMPC can become inaccurate during aggressive maneuvers where the vehicle's non-linear dynamics (like tire slip) are significant.

- **Non-linear MPC (NMPC):** This is the more advanced approach, which uses the full, *non-linear* dynamic model directly for its predictions. This provides much higher accuracy, as the controller can reason about complex physical effects like tire saturation. The trade-off is a significant increase in computational complexity. Solving a non-linear optimization problem in real-time is a much harder task and requires specialized, highly efficient software solvers, such as `acados` [32].

The choice between LMPC and NMPC depends on the specific application. For low-speed or simple path-tracking tasks, LMPC is often sufficient. For high-performance autonomous racing, where operating near the physical limits is necessary, the superior accuracy of NMPC is required.

3.12 Markov Decision Process (MDP)

The majority of the following topics are based on the foundational text by Sutton and Barto [26]. Reinforcement Learning provides a formal framework for tackling sequential decision problems, where the outcome depends not on a single action, but on a sequence of actions taken over time. In this paradigm, a learning **agent** interacts with an **environment**. The agent must choose actions, and the environment responds to those actions by presenting new situations, or **states**, and giving rewards.

A state provides an unambiguous description of a situation within the environment. From any given state, a set of actions is available to the agent. In many real-world scenarios, the outcome of an action can be uncertain. To model this, we use a **transition function**, which defines a probability distribution over the possible next states.

Let's formalize these concepts[1]:

- S: A finite set of states.
- A: A finite set of actions.
- $T : S \times A \to S$: The transition function.
- $P(s'|s, a)$: The probability of transitioning to state s' after taking action a in state s. This is drawn from a known probability distribution over the state space.

The final component is the reward, which guides the agent's learning process. We assume a deterministic **reward function**:

- $R : S \times A \times S \to \mathbb{R}$: The reward function.
- $r_{s,a,s'} = R(s, a, s')$: The immediate reward received after transitioning from state s to state s' by taking action a.

[1] To simplify notation, we assume that the set of available actions is the same for all states and that the probability distribution over next states is discrete.

A **Markov Decision Process (MDP)** is formally defined as a tuple $\langle S, A, T, R \rangle$.

Figure 3.4 shows a classic example of an MDP known as a grid world. The agent's current state is $s_{(3,1)}$. The set of states S includes all white squares, while the gray squares are unreachable. The star represents a terminal state, which ends an episode. The available actions are $A = \{\textit{Up, Down, Left, Right}\}$. This environment could be either stochastic (e.g., a 20% chance of moving right when the agent chooses to move up) or deterministic (the chosen action is always executed perfectly). The dashed line represents the **optimal policy** from state $s_{(3,1)}$, as it is the path that yields the maximum possible cumulative reward.

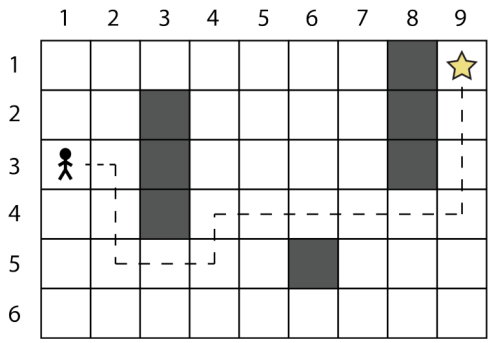


Figure 3.4: An example of an MDP: the grid world environment. Taken from [17].

3.12.1 MDP Variants

Deterministic MDP

The stochastic MDP described above is a generalization of the simpler **deterministic MDP**. In a deterministic MDP, the transition function is no longer probabilistic:

- $T : S \times A \to S$: The deterministic transition function.
- $s' = T(s, a)$: From state s, taking action a leads to a single, uniquely determined next state s'.

Continuous MDP

In a **continuous MDP**, the problem becomes more complex as one or more of its components are defined over continuous spaces:

- The state space S can be continuous (e.g., vehicle position and velocity).
- The action space A can be continuous (e.g., steering angle and acceleration).
- Both the state and action spaces can be continuous.

These variants are significantly more challenging to solve than their discrete counterparts.

3.12.2 Policies and Value Functions

The agent's decision-making logic is encapsulated in a **policy**, denoted by π. A policy is a function that maps states to actions, specifying which action the agent should take in any given state:

$$\pi : S \to A$$

The central goal of reinforcement learning is to find the optimal policy, π^*. To do this, we must first define what makes one policy better than another. This is achieved by using *value functions*, which estimate the "goodness" of being in a state or taking an action.

The **state-value function**, $V^\pi(s)$, represents the expected total reward an agent can collect starting from state s and following policy π thereafter. To handle infinite-horizon problems, a **discount factor**, $\gamma \in [0, 1)$, is introduced to give more weight to immediate rewards over future ones. The state-value function is defined by the **Bellman equation**:

$$V^\pi(s) = \sum_{s'} P(s'|s, \pi(s)) \left[r_{s,\pi(s),s'} + \gamma V^\pi(s') \right]$$

The optimal state-value function, $V^*(s)$, is simply the maximum value achievable from state s over all possible policies: $V^*(s) = \max_\pi V^\pi(s)$.

Similarly, the **action-value function**, $Q^\pi(s, a)$, is the expected total reward after taking a specific action a in state s and then following policy π afterwards. The optimal action-value function is $Q^*(s, a) = \max_\pi Q^\pi(s, a)$.

These two value functions are closely related. The value of being in a state is equal to the value of taking the best possible action from that state:

$$V^*(s) = \max_{a \in A} Q^*(s, a)$$

Once the optimal action-value function $Q^*(s, a)$ is known, the optimal policy π^* can be determined by simply choosing the action that maximizes this value in any given state:

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} Q^*(s, a)$$

3.12.3 Solving an MDP

Solving an MDP, especially a continuous one, can be a computationally demanding task. For discrete MDPs where the full model (i.e., the transition and reward functions) is known, classical dynamic programming algorithms can be used to find the optimal policy.

The two main algorithms are **Value Iteration** and **Policy Iteration**. Both methods repeatedly apply the Bellman equation to iteratively update the value function estimates until they converge to the optimal values, from which the optimal policy can be derived [19]. However, these classical methods are not used in this thesis, as they are not applicable to problems where the model is unknown or the state-action spaces are continuous.

3.13 Proximal Policy Optimization (PPO)

The classical methods for solving MDPs, like Value Iteration, are not practical for problems with continuous state or action spaces. For these more complex scenarios, modern RL algorithms use function approximation, typically with deep neural networks, to learn a policy. Proximal Policy Optimization (PPO) [24] is one of the most popular and effective algorithms in this category.

3.13.1 Policy Gradient Methods

PPO belongs to a family of algorithms known as **policy gradient methods**. Unlike value-based methods (like Q-learning) which first learn an action-value function and then derive a policy from it, policy gradient methods learn a parameterized policy directly.

The policy, $\pi_\theta(\mathbf{a}|\mathbf{s})$, is represented by a neural network with parameters θ. The goal is to find the optimal parameters θ^* that maximize the expected total reward, an objective function denoted as $J(\pi_\theta)$, which represents the overall performance of the policy averaged across all possible starting states. This is achieved by performing gradient ascent on the objective function. At each step, the algorithm adjusts the policy parameters in the direction of the "policy gradient", $\nabla_\theta J(\pi_\theta)$, which points in the direction of steepest increase in performance. The basic update rule is:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)$$

where α is the learning rate.

A major challenge with this basic approach is its instability. A single update step that is too large can drastically change the policy, leading to a collapse in performance from which it may not recover. PPO was specifically designed to address this stability issue.

3.13.2 The PPO Clipped Objective Function

The key innovation of PPO is its use of a **clipped surrogate objective function**. The goal of this function is to prevent the new, updated policy (π_θ) from moving too far away from the old policy ($\pi_{\theta_{\text{old}}}$) in a single update step.

This is achieved by first calculating the probability ratio between the new and old policies:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

This ratio, $r_t(\theta)$, measures how likely the new policy is to select a certain action compared to the old one. If $r_t > 1$, the action is more likely under the new policy; if $r_t < 1$, it is less likely.

Instead of directly optimizing the standard policy gradient objective, PPO optimizes a clipped version. The simplified objective function is:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

Let's break down this formula:

- $\hat{A}_t$: This is the **Advantage function**, which estimates how much better an action was compared to the average action from that state. A positive advantage means the action was good; a negative one means it was bad.
- $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$: This function constrains the probability ratio $r_t(\theta)$ to stay within the range $[1-\epsilon, 1+\epsilon]$. The hyperparameter ϵ is a small value (e.g., 0.2) that defines the size of this "trust region."
- $\min(\dots,\dots)$: The use of the minimum operator is the crucial part. It takes the smaller of two values: the normal policy objective and the clipped one. This has the effect of creating a pessimistic bound on the update. If an action was good ($\hat{A}_t > 0$), the objective is capped, preventing the policy from becoming "too greedy" and increasing the probability of that action too much. If an action was bad ($\hat{A}_t < 0$), the penalty

is also limited.

By taking this minimum, PPO ensures that the policy updates are conservative and small, which dramatically improves learning stability.

3.13.3 The Actor-Critic Architecture

In practice, PPO is almost always implemented using an **Actor-Critic** architecture, which involves two separate neural networks.

- **The Actor:** This network represents the policy, $\pi_\theta(\mathbf{a}|\mathbf{s})$. It takes the current state as input and outputs the action (or a probability distribution over actions) for the agent to take.
- **The Critic:** This network represents a state-value function, $V_\phi(\mathbf{s})$. It takes the current state as input and outputs a single value that estimates the expected total future reward from that state.

The two networks work together. The Actor is responsible for acting, while the Critic is responsible for evaluating those actions. The Critic's value estimate is used to compute the Advantage function ($\hat{A}_t$), which in turn is used to train the Actor via the PPO clipped objective. This setup is more stable and data-efficient than using the raw sum of rewards to train the policy.

Chapter 4

Related Work

This chapter provides an overview of existing research and projects in the field of autonomous racing, as well as in the specific domain of combining Model Predictive Control with Reinforcement Learning. This review serves to position our thesis within the broader context of the state-of-the-art.

4.1 Autonomous Racing Platforms

Academic and industrial research in autonomous driving has greatly benefited from competitions that push the boundaries of technology. We will review the literature from four key tiers of autonomous racing.

4.1.1 FSAE Driverless

The Formula Student Driverless competition serves as a foundational proving ground for university teams to develop full autonomous driving stacks from scratch. The primary challenge—navigating unknown, cone-defined tracks—has produced a wealth of public research on system architectures.

A common approach, detailed by multiple teams including KA-RaceIng from Karlsruhe Institute of Technology [2], involves a modular software stack built on ROS. These systems typically feature a LiDAR-based perception pipeline

for cone detection, a SLAM algorithm for mapping, a distinct trajectory planning module, and a controller for execution. The work by the AMZ team from ETH Zurich [11] is particularly influential, presenting a highly successful software stack that uses a MPC for control. Their work emphasizes the importance of an accurate vehicle model and demonstrates that a well-formulated MPC can achieve state-of-the-art performance in path tracking at the limits of handling.

These papers highlight a consensus on the general architecture for the competition, focusing on robust perception and state estimation as the bedrock for reliable planning and predictive control.

4.1.2 RoboRacer

The RoboRacer (formerly known as F1TENTH) competition is a 1/10th scale autonomous racing series that provides an accessible, low-cost platform for research in agile robotics. The small scale and high relative speeds make it an ideal testbed for advanced control algorithms.

Much of the research in RoboRacer focuses on comparing different control strategies. The classic path-following algorithm, Pure Pursuit, is often used as a baseline due to its simplicity and effectiveness. More advanced approaches, such as the use of Model Predictive Control, are also widely explored. Very influential work from ForzaETH is [3]. Papers like [12], show that MPC, combined with RL, can provide superior performance by explicitly accounting for vehicle dynamics and optimizing over a future horizon, leading to faster and more stable lap times. The RoboRacer community has also produced a wide range of works on perception, using both 2D LiDAR and cameras to navigate the track and avoid static or dynamic obstacles.

Also the UniBo Motorsport team has been working for the last three years on this challenge. Latest results are a 3rd place at CDC 2024 Milan, and a 1st place at ICRA 2025 Atlanta, both against the strongest teams, including

ForzaETH.

4.1.3 Indy Autonomous Challenge (IAC)

The Indy Autonomous Challenge (IAC) represents a significant step up in complexity from FSAE, featuring full-scale race cars competing head-to-head at speeds exceeding 270 km/h. The research from this competition necessarily focuses on challenges unique to high-speed and multi-agent scenarios.

Teams like PoliMOVE from Politecnico di Milano [7] and TUM Autonomous Motorsport [4] emphasize the need for maximum sensor detection range and reliable handling of multi-vehicle situations. Their work highlights the use of robust MPC for motion control under uncertainty and fast, low-level feedback loops to handle the vehicle's dynamics at the limit.

4.1.4 Abu Dhabi Autonomous Racing League (A2RL)

The A2RL is the newest and most ambitious competition in this domain, utilizing modified Super Formula cars, which are recognized as the fastest single-seaters outside of Formula 1. The first official race was held in April 2024, and as such, the body of peer-reviewed academic literature is still emerging.

Initial technical reports and presentations from participating teams [8] describe the league's software challenges, which build directly upon the foundations laid by the IAC. Teams must develop software for a common vehicle platform to handle time trials, head-to-head overtaking, and multi-car races. The key focus remains on robust, high-speed perception and predictive control capable of handling extreme vehicle dynamics. Given its recency, A2RL represents the current frontier of autonomous racing research, and its future development will likely produce significant advancements in the field.

4.2 Combining MPC and Reinforcement Learning

The integration of traditional control theory with modern machine learning is a highly active area of research. Our work on using RL to tune controller hyperparameters is inspired by an emerging paradigm that seeks to combine the strengths of MPC and RL.

This approach, applied to drones, is detailed in [22]. In this methodology, the MPC controller is not a static component but a parameterized policy that can be adapted online. The MPC acts as both the policy provider, computing the optimal action for the environment, and as a function approximator for the RL agent's value function. Concurrently, an RL algorithm is used to tune the parameters of the MPC—such as the weights of its cost function or the parameters of its internal model—to continuously improve the controller's performance based on experience.

This synergy allows the system to leverage the key benefits of both approaches. From MPC, the ability to explicitly handle system constraints (e.g., actuator limits, friction ellipse) and to guarantee stability and safety through its model-based predictions. While from RL, the ability to learn and adapt from data, optimizing complex, non-linear objectives without requiring a perfect model of the environment.

This framework provides a structured and safe way to apply learning-based methods to real-world control problems. Our thesis applies a similar philosophy, using PPO to tune the hyperparameters of a classical controller, thereby leveraging RL's optimization power within a safe and well-understood control architecture.

Chapter 5

Technical Contributions

5.1 Athena's Autonomous Stack

In the following section we are going to give a general explanation of the whole Driverless stack running on our prototype. This will allow to understand how the more detailed contributions in the following sections are placed within the project. Our imposed time budget for the whole pipeline is 20 ms (50 Hz).

5.1.1 Software Infrastructure

The software stack running on the Autonomous System Unit (ASU) is built on ROS 2 [16]. Because in ROS each node is a separate program, team members can develop and test different parts of the system independently. This modularity was crucial for the efficient development of our software stack. The stack is primarily written in C++17 to take advantage of its performance, which is essential for low-latency perception and control tasks. For rapid prototyping of complex algorithms, such as path planning or state estimation, we initially use Python. Once an algorithm is validated, it is rewritten in C++ for optimal performance on the vehicle.

For version control and collaboration, we use Git and host our repositories on GitHub. Our development process is issue-driven: every new feature or

bug fix begins as an issue. A corresponding branch is created for development, and changes are submitted via a Pull Request (PR). Each PR is reviewed by team members and must pass automated checks before being merged into the main branch.

We use GitHub Actions for Continuous Integration (CI). This automated pipeline builds and pushes updated Docker images whenever the configuration changes, ensuring that our deployments are consistent. Additionally, a linter is automatically run on every PR to enforce a common coding style and maintain code readability. This structured workflow ensures code quality and simplifies testing.

The VCU is responsible for executing low-level commands and interfacing with the vehicle's actuators. The ASU communicates with the VCU over a TCP socket, which provides a reliable, low-latency connection. High-level commands—such as target throttle, brake pressure, and steering angle—are computed by a control node on the ASU and sent to the VCU.

The ASU Manager: A Supervisory Node

As already mentioned in Section 2.5.1, our software stack includes a central supervisory node responsible for managing the lifecycle of all other processes required to complete a mission. This component, named the ASU Manager node, acts as a master controller and state machine for the entire autonomous system.

It is implemented as a ROS 2 node but is managed at the operating system level for maximum robustness. Upon boot of the ASU, which runs Ubuntu 24.04, the manager is automatically launched as a background process (or *daemon*) by a system service. This service ensures its reliability: if the manager process terminates for any reason, the service automatically restarts it after ensuring all previously launched child processes are terminated. This is handled by a Linux kernel feature known as a control group (*CGroup*), which allows a set of related processes to be managed and terminated as a single unit.

Initialization and Startup Upon launch, the manager first reads a configuration file in YAML format. This file defines which software components are considered core "startup" processes, essential for basic system operation. The manager then uses the *fork()* and *execvp()* system calls to spawn these core ROS 2 nodes as child processes. After launching them, it performs a crucial readiness check. It communicates with each critical node via dedicated ROS 2 services (e.g., */<node_name>/is_ready*) to confirm that the node has initialized successfully and is ready to operate. The system will not proceed until all essential startup nodes report a ready state, ensuring a stable foundation before any mission-specific logic is initiated.

Mission Management and State Transitions The manager's primary role during operation is to act as a state machine, responding to commands from the VCU and internal system events. It subscribes to a */vcu/data* topic to receive status updates, including the selected mission (e.g., Acceleration, Trackdrive) and the state of the Autonomous System Master Switch (ASMS), a physical key placed on the car's chassis that is turned on when we want to activate the autonomous system. The system is designed to wait for two conditions to be met: the selection of a mission and the activation of the ASMS, both actions performed by the human. Once both are true, the manager proceeds to launch the specific set of ROS 2 nodes required for that mission, again referencing the YAML configuration file to determine which processes to start. Once these mission-specific nodes are launched and report their readiness, the manager signals to the VCU that the ASU is fully ready to begin the mission.

Process Supervision and Fault Tolerance A key function of the manager is process supervision. Its main loop continuously monitors the status of all its child processes using a non-blocking *waitpid()* call. If any node terminates unexpectedly, whether due to a crash or a normal exit, the manager interprets this as a critical system event that requires a full system reset. It immediately

initiates a teardown sequence.

During teardown, the manager sends a *SIGKILL* signal to all remaining child processes to ensure a clean and immediate stop. It then exits, which in turn triggers the operating system service to restart the entire stack from a clean slate. This same robust teardown procedure is also triggered by custom ROS 2 diagnostic messages, such as an emergency message sent by any node or a mission-finished message indicating a successful run, ensuring a consistent and safe shutdown pathway for all scenarios.

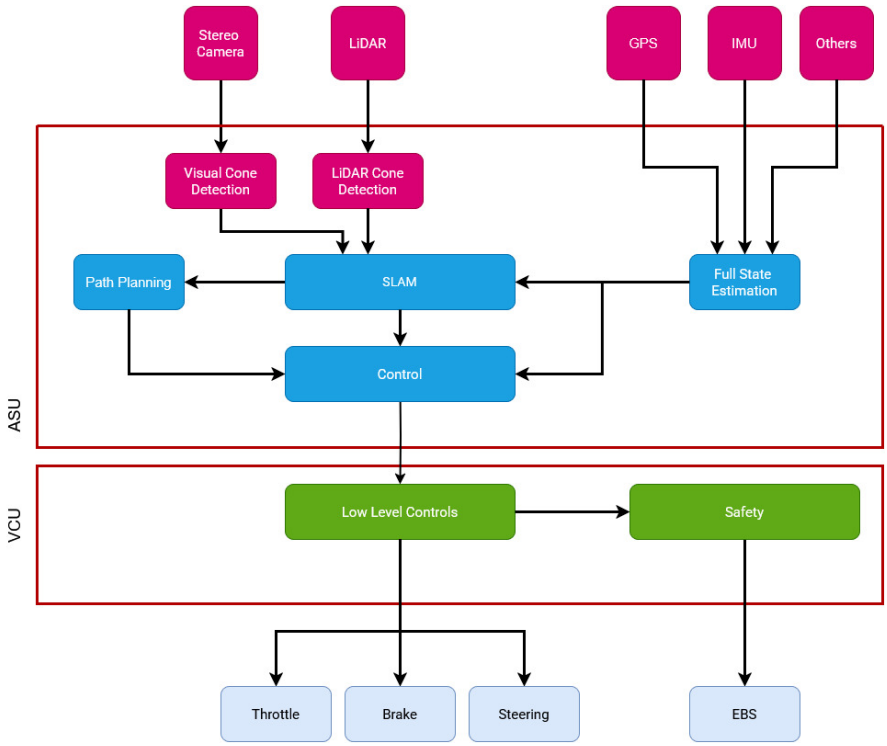


Figure 5.1: A high-level overview of the software stack architecture.

5.1.2 Perception

The prototype is equipped with a sensor suite designed for robust perception and localization. This suite includes a StereoLabs ZED 2i stereo camera, a SICK multiScan136 3D LiDAR, a GNSS/IMU unit, rear wheel encoders, and a steering angle sensor. The main goal of the perception stack is to detect

cones, estimate vehicle motion (odometry), build a map of the environment, and localize the car within it.

The LiDAR pipeline processes raw point cloud data received from the sensor over an Ethernet connection. The incoming points are first clustered, and ground points are filtered out. To remove remaining false positives, we apply a filter based on a point density estimation formula [11]. A Least Squares regression is then used to find the center of each valid cone cluster. These 3D coordinates are then passed to the SLAM module.

The stereo camera pipeline is inspired by the work presented in [11]. The stereo image is split into left and right frames, and a YOLO-based object detector identifies cones in each frame, producing a set of bounding boxes. To estimate the distance to a cone, a stereo matching algorithm is used. For each bounding box in the left image, the algorithm searches for a corresponding box in the right image along the epipolar line. Once a match is found, triangulation is used to compute the 3D coordinates of the cone, which are then published for use by the SLAM system.

All perception data is fed into a Simultaneous Localization and Mapping (SLAM) system. SLAM is a core algorithm in robotics that allows a vehicle to build a map of an unknown environment while tracking its own position within that map. Our SLAM implementation uses the detected cone positions from both LiDAR and the camera as landmarks.

To track its position between landmark sightings, the vehicle uses a motion model that fuses measurements from the IMU, wheel encoders, and steering sensor using an Extended Kalman Filter (EKF). This provides an estimate of the vehicle's current state, but this estimate accumulates error (drifts) over time. The SLAM system corrects this drift by comparing the predicted position with the actual observations of known cones, which maintains an accurate estimate of both the vehicle's pose and the map. Over the course of a lap, the map becomes more complete, and at the end of the lap, a loop closure step aligns the start and end points to create a globally consistent map.

5.1.3 Motion Planning

The planning module uses the cone map from SLAM to generate a race-optimal trajectory. The first step is to estimate the track's centerline. We do this by applying Delaunay triangulation to the cone positions. This algorithm creates a mesh of triangles connecting the cones, and the centerline is then calculated from the midpoints of the triangle edges that span the track.

Once the centerline is determined, we compute the optimal raceline using Model Predictive Control (MPC). MPC is an advanced control method that solves an optimization problem at each time step to find the best possible sequence of actions. It uses a dynamic bicycle model of the vehicle to predict its future behavior. The objective is to minimize lap time while respecting constraints such as track boundaries and the vehicle's physical limits.

For real-time vehicle control during a lap, we also use MPC. The framework is nearly identical to the one used for raceline generation, but the objective is changed from minimizing lap time to accurately following the pre-computed optimal raceline. The dynamic constraints remain the same.

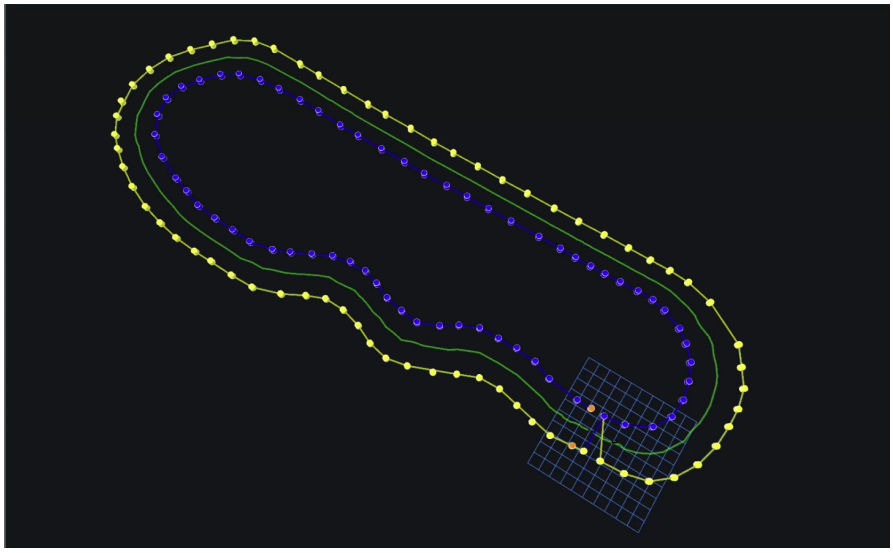


Figure 5.2: Delaunay triangulation of detected cones used to calculate the track centerline.

5.1.4 Simulator

A key component of our development workflow is a custom simulator built with the Unity game engine [30]. We chose Unity for its user-friendly interface, powerful physics engine, and fast deployment capabilities. The main purpose of the simulator is to provide a controlled environment for offline testing and debugging of the entire software stack.

Developing and diagnosing issues on the physical car is often difficult and time-consuming. The simulator offers a fast and safe feedback loop, allowing developers to test code changes immediately without needing access to the track or car. All simulator logic is implemented in C#, and we use Unity's built-in physics engine to handle collisions and object interactions. The vehicle model in the simulation is the same dynamic bicycle model used for planning and control. While simulated dynamics do not perfectly match the real world, using the same model ensures that control logic tested in simulation behaves predictably on the real vehicle.

The latest version of our simulator can also emulate LiDAR data, generating 3D point clouds similar to those from the real sensor. This feature allows us to develop and test our perception and SLAM algorithms entirely within the simulation.

5.1.5 Actuator Systems

The autonomous software stack's decisions are translated into physical motion by two critical hardware components: the steering and braking actuators. From a computer science perspective, these systems are the final output layer of the control loop, converting digital commands into mechanical force and displacement.

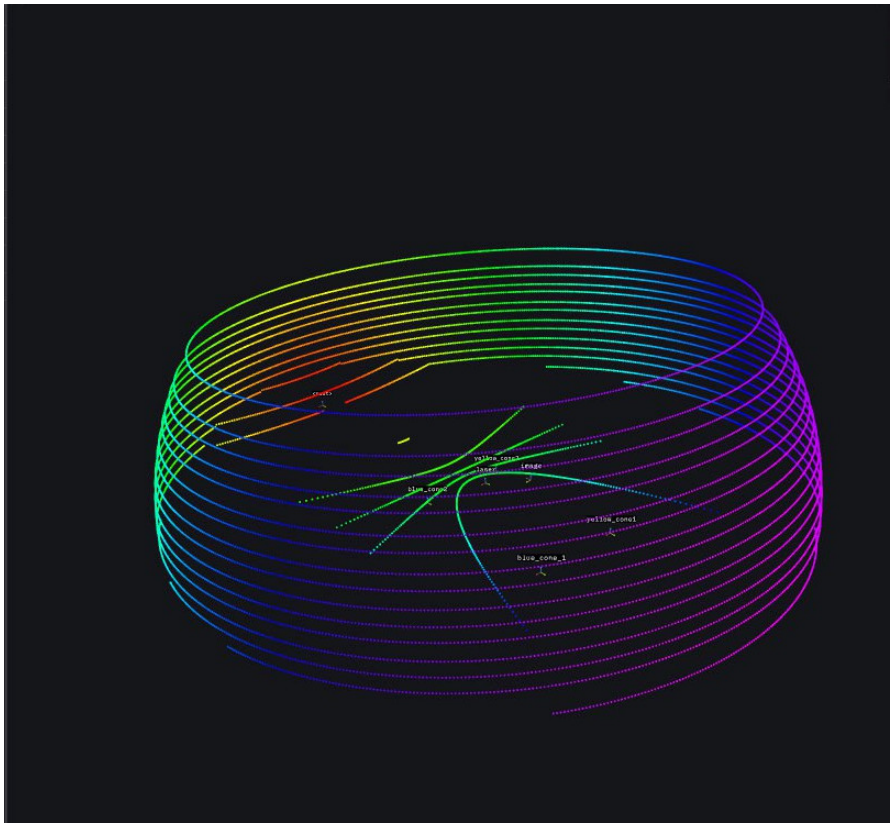


Figure 5.3: A view of the simulated LiDAR point cloud within the Unity environment.

Steering Actuator

The steering actuator is an electro-mechanical system responsible for translating a computed steering command from the motion planning software into a physical angle of the front wheels.

The core of the system is a high-torque electric motor coupled with a ball screw mechanism. This configuration efficiently converts the motor's rotational motion into the high linear force required to actuate the vehicle's steering rack. The design was constrained by the available electrical power and physical packaging space within the chassis. The resulting assembly, shown in Figure 5.4, provides precise and rapid control over the vehicle's heading, which is fundamental for accurate path tracking. The control software interfaces with this system by sending a target position or velocity command to the motor controller.

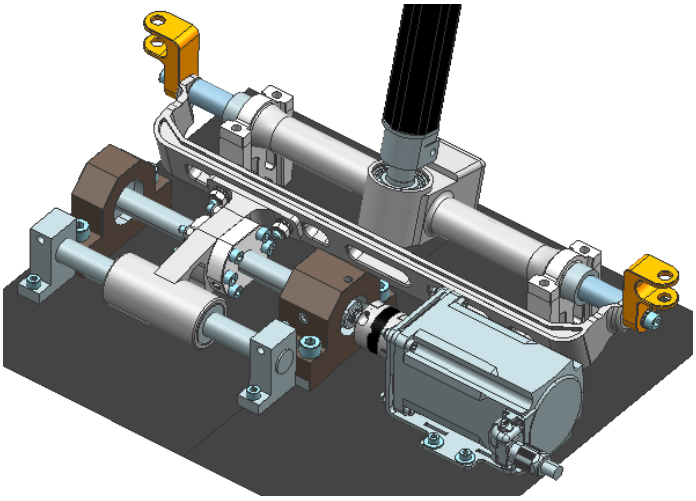


Figure 5.4: The integrated electric motor and ball screw assembly for steering actuation.

Braking System

The vehicle is equipped with a dual-actuation braking system, designed to be independent of the driver's brake pedal. This architecture provides both fine-grained control for performance driving and a robust fail-safe for emergency situations. The two systems are:

- **Autonomous System Brake (ASB):** This is the primary service brake used for normal autonomous operation. It consists of a **linear electric actuator** that provides precise, proportional control over the braking force. This allows the motion planning and control algorithms to modulate deceleration for tasks like corner entry and stopping at a target location.

- **Emergency Brake System (EBS):** This is a safety-critical, **pneumatic system** designed for maximum, non-proportional braking. It is triggered by the main vehicle safety circuit in the event of a critical system failure or an emergency stop command. Its function is to bring the vehicle to a halt as quickly as possible, serving as a crucial safety override.

As shown in Figure 5.5, both the electric ASB actuator and the pneumatic EBS cylinders are integrated into a single mechanical unit. They act on the same master cylinders, which transmit pressure to the hydraulic brake calipers. The hydraulic circuits are managed by **shuttle valves**, which ensure that whichever system generates the higher pressure—be it the ASB, the EBS, or the manual driver's pedal—is the one that actuates the brakes, preventing interference between the independent inputs.

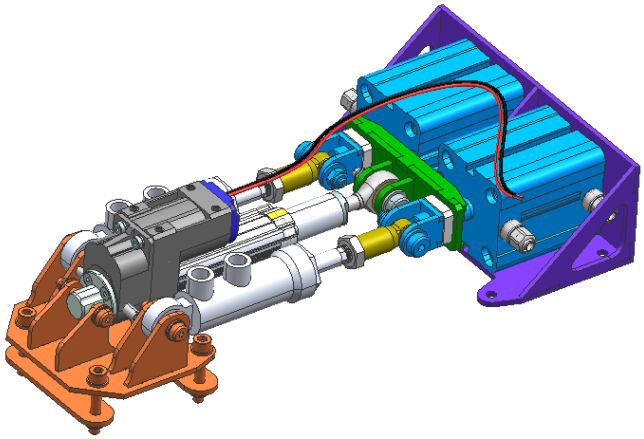


Figure 5.5: The combined assembly housing the electric Autonomous System Brake (ASB) actuator and the pneumatic Emergency Brake System (EBS) cylinders.

5.2 Stereocamera Pipeline

The perception system relies on a stereocamera to detect the track layout and reconstruct the 3D positions of the cones. This process begins with acquiring images from the sensor and concludes with a list of 3D landmarks, complete with color information, in the vehicle's coordinate system.

5.2.1 Sensor Characteristics and Operational Limits

Our system utilizes a ZED 2i stereocamera, fitted with polarizing filters to mitigate glare. The camera is configured with the following operational parameters:

- **Resolution & Framerate:** 2x (1280x720) at 60 fps
- **Focal Length:** 4 mm
- **Baseline:** 12 cm
- **Field of View (FOV):** 72° (H) x 44° (V) x 81° (D)
- **RGB Sensors:** Dual 1/3" 4MP CMOS, 2µm pixel size, rolling shutter
- **Motion Sensors:** Integrated Accelerometer and Gyroscope. The magnetometer cannot be used, as its readings are heavily compromised by interference from the vehicle's nearby metallic main hoop[1].
- **Physical Specs:** 175.3 x 30.3 x 43.1 mm, 229 g
- **Power:** 380 mA at 5V, supplied via USB-C

The manufacturer's specifications on depth perception provide critical insight into the system's operational limits. The ideal depth range is stated as 1.5 m to 20 m, with depth accuracy degrading significantly with distance. At 2 m, the accuracy is better than 0.4% (<8 mm), but at 20 m, it falls to less than 7%, which corresponds to a potential error of 1.4 meters. This is a significant margin of error that implies that the performance of our perception algorithms will inherently be less reliable for cones detected at longer distances.

[1] The main hoop is a mandatory element that each car needs to have [9].

5.2.2 Image Acquisition and Preprocessing Node

The pipeline's first stage is a custom ROS 2 node that wraps an open-source driver [15] to interface with the camera. This node is responsible for acquiring the raw image data and performing initial validation and processing.

A significant operational challenge identified during testing is the camera's default automatic exposure setting. When faced with a bright sky, the camera often underexposes the lower portion of the image, making the traffic cones difficult to discern. To address this, future work will focus on implementing manual control over the camera's exposure settings.

This node's primary responsibility, however, is to handle a data corruption issue observed in practice. At a non-negligible rate, the camera driver outputs a "shifted" frame, where the left and right images are not correctly placed side-by-side (Figure 5.6).

Figure 5.6: Example of a corrupted output from the camera driver. Instead of a single, sharp discontinuity at the center, multiple misaligned seams are present.

Processing such a frame would lead to erroneous depth calculations. To prevent this, a data validation algorithm is implemented as a sanitation check. It exploits the fact that a correctly formed stereo image has a sharp visual discontinuity along the central vertical seam. The algorithm operates as follows:

1. It isolates a narrow vertical strip at the horizontal center of the incoming stereo frame.

2. It calculates the column-wise Sum of Absolute Differences (SAD) in pixel intensity directly on the central seam, as well as on the columns

immediately to its left and right.

3. It computes a ratio of these sums to determine if the visual difference at the seam is significantly larger than in the adjacent, continuous parts of the image.

If the difference at the seam is not pronounced, the algorithm identifies the frame as corrupted, and it is subsequently discarded. Testing of this validation algorithm showed a perfect F1 score, both on-track and off-track. Once a frame is validated, it is rectified using the precise manufacturer-provided calibration matrices for our specific camera unit. The resulting pair of rectified images, along with the corresponding calibration data, are then published to the ROS local network for consumption by the downstream perception nodes.

5.2.3 Cone Detection with YOLO

Once the synchronized left and right image frames are received, the first step in the perception pipeline is to detect the traffic cones within each image. This task is performed by a custom-trained You Only Look Once (YOLO) deep neural network. The network version used is *yolov11n* [10]. To achieve the real-time performance required for autonomous racing, the inference process is accelerated using hardware-specific optimization frameworks. The system is designed to leverage either the NVIDIA TensorRT engine [29] for NVIDIA GPUs or the OpenVINO toolkit [28] for Intel hardware[2]. The selection of the framework has to be done before the pipeline compilation.

The detection process for each stereo pair follows these steps:

1. **Image Preprocessing:** The raw left and right images are first preprocessed to match the fixed input dimensions required by the YOLO model (i.e., 640x640 pixels). To avoid distorting the objects, the original aspect ratio is maintained. Each image is resized to fit within the model's

[2]OpenVINO has been chosen also as a way to debug the pipeline if only an Intel CPU was available. Though, on the ASU, TensorRT is the main framework.

input dimensions, and the remaining space is filled with black padding.

2. **Batched Inference:** A key performance optimization is the use of batched inference. The preprocessed left and right images are stacked into a single batch of size two. This batch is then fed to the inference engine (TensorRT or OpenVINO) in a single pass, which is significantly more efficient than processing the two images sequentially.

3. **Post-processing and Filtering:** The raw output from the neural network is a list of potential object detections with associated confidence scores. This output is post-processed to yield the final set of bounding boxes. First, all detections with a confidence score below a predefined threshold are discarded. Then, Non-Maximum Suppression (NMS) is applied. NMS is a crucial algorithm that eliminates redundant, overlapping bounding boxes for the same object, ensuring that only the single best bounding box for each detected cone is retained.

Figure 5.7: YOLO output after NMS. The numbers represent the confidence (or probability) of the detections.

The output of this stage consists of two independent lists of 2D bounding boxes: one for the cones detected in the left image and one for the cones detected in the right image. Each bounding box contains the pixel coordinates, dimensions, and class ID (e.g., blue cone, yellow cone) of a detected cone.

5.2.4 Stereo Matching of Detections

After detecting cones in both images, the next critical step is to solve the data association problem: determining which bounding box in the left image corresponds to which bounding box in the right image. This stereo matching process is fundamental for 3D pose estimation. Our pipeline employs a multi-stage approach to find robust and accurate correspondences.

1. **Geometric Candidate Filtering:** For each bounding box in the left image, a search is performed to find potential matching candidates in the right image. This search is heavily constrained by geometric rules derived from the stereo camera setup. A right-image bounding box is considered a valid candidate only if it satisfies several conditions relative to its left-image counterpart:

    - The bounding boxes' vertical centers must be very similar, in accordance with the epipolar constraint of a rectified stereo system.
    - The height of the bounding boxes must be comparable.
    - The horizontal center of the right-image box must be to the left of the left-image box's center, and the difference in their positions (the disparity) must fall within a plausible range. This range is dynamically adjusted based on the object's vertical position in the image, allowing for larger disparities for closer objects.
    - The algorithm includes special logic to handle cones that are partially visible at the edges of the frame, which improves robustness.

2. **Template Matching for Refinement:** If one or more candidates are found, the matching is refined using template matching. The image patch defined by the left bounding box is used as a template. This template is then searched for within a constrained horizontal region of the right image, defined by the positions of the candidate boxes. Template matching is used to find the location of the best match. To achieve

higher precision, the disparity is refined to sub-pixel accuracy by fitting a parabola to the template matching scores around the peak response.

3. **Feature-Based Correspondence:** Inspired by [11], to further improve the robustness of the 3D position estimate, an optional feature-matching step can be enabled. For each matched pair of bounding boxes, an ORB (Oriented FAST and Rotated BRIEF) [23] feature detector is used to find multiple stable keypoints within each box's image patch. These keypoints are then matched between the left and right patches. This provides a set of multiple, spatially distributed correspondence points for a single cone, which is more resilient to minor inaccuracies in the bounding box itself. If this step is disabled or fails to find matches, the system falls back to using the sub-pixel-refined center point of the bounding boxes as the single correspondence point.

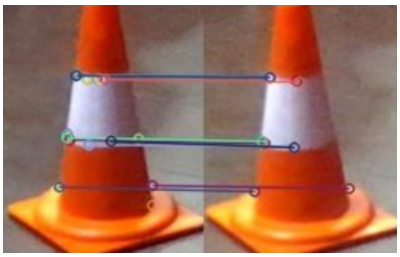


Figure 5.8: Example of ORB feature matching on a pair of left-right bounding boxes.

The result of this stage is a list of matched cone pairs. Each pair contains a set of corresponding 2D points (one or more) from the left and right images that belong to the same physical cone.

5.2.5 3D Landmark Triangulation

The final step in the pipeline is to use the 2D matched points to reconstruct the 3D position of each cone relative to the camera. This process is known as triangulation.

For each matched cone, the set of corresponding 2D keypoint pairs and the camera's pre-calibrated projection matrices are passed to the standard OpenCV [6] triangulation (*cv::triangulatePoints*) function. This function uses a linear triangulation method to compute the 3D coordinates for each keypoint pair. This results in a small 3D point cloud for each cone.

To obtain a single, robust 3D position estimate from this point cloud, the **median** of the X, Y, and Z coordinates is calculated. Using the median is a highly effective technique for rejecting outlier points that may have resulted from incorrect feature matches, leading to a more stable final position estimate[3].

Furthermore, the system also estimates the uncertainty of this 3D position. The variance of the position is calculated based on the estimated depth and the known uncertainty of the disparity measurement in pixels. This provides a covariance matrix for each landmark, which is essential information for downstream probabilistic filters, such as the SLAM system.

Finally, the calculated 3D point, now in the camera's coordinate system, is transformed into the vehicle's reference frame using the known extrinsic calibration (the position and orientation of the camera on the car's chassis). The final output of the stereocamera pipeline is a list of 3D landmarks, each with a position and an associated covariance, ready to be used for mapping and navigation.

5.2.6 Experiments and Results

Custom YOLO training

The cone detection model was trained and evaluated using the publicly available FSOCO (Formula Student Objects in Context) dataset [33]. The model was trained for 300 epochs, and its final performance was measured on a test split, which comprises 1,968 images containing a total of 36,123 annotated

[3] A likely more accurate way to do this would be to compute the geometric median, at the cost of solving an optimization problem.

cone instances. The dataset composition is shown in Table 5.1, while the performance metrics are summarized in Table 5.2.

The primary metrics are defined as follows:

- **Precision:** Measures the accuracy of the detections. Of all the predictions made by the model, this is the fraction that were correct. A high precision indicates a low rate of false positives.
- **Recall:** Measures the model's ability to find all relevant objects. Of all the actual cones present in the images, this is the fraction that the model successfully detected. A high recall indicates a low rate of false negatives.
- **mAP50:** The mean Average Precision calculated at an Intersection over Union (IoU) threshold of 0.50. This metric evaluates how well the model can both classify an object and localize it with a bounding box that overlaps at least 50% with the ground truth box.
- **mAP50-95:** The mAP averaged over a range of IoU thresholds from 0.50 to 0.95. This is a much stricter metric, as it requires highly accurate bounding box placement to score well.

As shown in the *All Classes* summary row, the model achieves an overall mAP50 of 0.824, indicating strong general performance. The precision of 0.849 suggests that when the model detects a cone, it is correct about 85% of the time, while the recall of 0.765 means it successfully identifies roughly 77% of all cones present in the dataset.

The per-class breakdown reveals further insights. The model performs exceptionally well on the primary cone classes: *Blue*, *Yellow*, and *Orange*. The precision for these classes is very high (above 0.92), meaning there are very few incorrect classifications. The mAP50 scores for these classes are also excellent, nearing 0.90, which confirms the model's ability to reliably detect and localize the main track boundary markers.

The *Large Orange Cone* class achieves the highest mAP scores (0.908 for mAP50 and 0.710 for mAP50-95). This is likely because these cones are larger and more visually distinct, making them an easier target for the detector.

Conversely, the *Unknown Cone* class exhibits significantly lower performance across all metrics (mAP50 of 0.547). This result is expected, as this class serves as a catch-all for heavily occluded, blurry, or poorly illuminated cones that are difficult to classify even for a human annotator. The model's struggle with this ambiguous class is acceptable, as its primary function is to accurately identify the well-defined track boundaries. Overall, the results confirm that the custom-trained model is highly effective for its intended purpose.

Table 5.1: Composition of the FSOCO test split used for model evaluation.

Class	**Images**	**Instances**
All Classes	1968	36123
Blue Cone	1416	12720
Yellow Cone	1638	15605
Orange Cone	850	5462
Large Orange Cone	428	1263
Unknown Cone	178	1073

Table 5.2: Performance metrics of the trained cone detector on the FSOCO test split.

Class	**Precision**	**Recall**	**mAP50**	**mAP50-95**
All Classes	0.849	0.765	0.824	0.570
Blue Cone	0.922	0.806	0.896	0.615
Yellow Cone	0.926	0.794	0.892	0.608
Orange Cone	0.925	0.787	0.877	0.603
Large Orange Cone	0.868	0.873	0.908	0.710
Unknown Cone	0.603	0.566	0.547	0.315

Performance and Timing Analysis

To evaluate the real-world performance of the stereocamera pipeline, its timing characteristics were benchmarked using data recorded during a manually

driven lap at the Rioveggio Karting Circuit. The test was executed on a high-performance machine with hardware comparable to the vehicle's ASU, featuring an AMD Ryzen 9 6900HX CPU and an NVIDIA GeForce RTX 3080 Mobile GPU.

The pipeline was benchmarked across four distinct configurations to assess the impact of both the inference engine (TensorRT vs. OpenVINO) and the feature matching strategy (sub-pixel refined center point vs. ORB features). The timing statistics, averaged over 1870 frames, are presented in Table 5.3 and Table 5.4.

Table 5.3: Mean and standard deviation of processing times (in milliseconds) for the TensorRT configuration, running on an NVIDIA RTX 3080 GPU. In parenthesis the standard deviation.

Processing Stage	**Center Point**	**ORB Features**
Preprocessing	0.34 (0.08)	0.34 (0.07)
Inference	6.78 (2.45)	6.78 (2.05)
Postprocessing	0.38 (0.08)	0.38 (0.08)
BBox Matching	0.38 (0.27)	0.40 (0.28)
Feature Matching	0.00 (0.00)	1.41 (4.27)
Triangulation	0.02 (0.01)	0.11 (0.07)
Sending	0.04 (0.02)	0.04 (0.01)
Total	**7.95 (2.57)**	**9.46 (6.35)**

Table 5.4: Mean and standard deviation of processing times (in milliseconds) for the OpenVINO configuration, running on an AMD Ryzen 9 CPU. In parenthesis the standard deviation.

Processing Stage	**Center Point**	**ORB Features**
Preprocessing	1.31 (0.19)	0.37 (0.08)
Inference	24.72 (1.61)	24.30 (1.25)
Postprocessing	0.59 (0.07)	0.60 (0.06)
BBox Matching	0.50 (0.36)	0.49 (0.35)
Feature Matching	0.00 (0.00)	1.76 (2.69)
Triangulation	0.03 (0.01)	0.15 (0.08)
Sending	0.05 (0.01)	0.05 (0.01)
Total	**27.20 (1.74)**	**27.72 (3.50)**

The results highlight several key aspects of the pipeline's performance:

- **Inference Engine Impact:** The most significant factor influencing total processing time is the inference engine. The GPU-accelerated TensorRT configuration is approximately **3.6 times faster** at the core inference task than the CPU-based OpenVINO configuration (6.78 ms vs. 24.72 ms). This demonstrates the critical importance of GPU hardware acceleration for running deep learning models in a real-time context.

- **Feature Matching Overhead:** The choice of matching strategy introduces a clear trade-off between performance and robustness. Using ORB features adds a noticeable overhead, increasing the total processing time by about 1.41 ms in the TensorRT case and 1.76 ms in the OpenVINO case. The *Feature Matching* and *Triangulation* stages take significantly longer, as they must process multiple keypoints per cone instead of a single center point. This additional processing time is the cost of achieving a potentially more stable 3D position estimate by relying on multiple geometric correspondences. It is also worth noting the high standard deviation of the feature matching stage, which is expected as the computation time is directly proportional to the number of times ORB is executed[4], a value that varies significantly from frame to frame.

- **Real-Time Capability:** All four configurations operate well within the time budget required for our 60 fps camera, which is approximately 16.7 ms per frame. The TensorRT configurations are fast, with the full pipeline completing in under 10 ms. This leaves a margin (even though not too comfortable) for all other processes running on the ASU, such as SLAM and MPC. The OpenVINO configuration, while slower, still operates at an acceptable speed (27 ms, or 37 Hz), making it a viable fallback option if dedicated GPU hardware is unavailable.

[4]ORB is executed sequentially for each pair of bounding boxes. Future work could focus on executing ORB on GPU if it is a non-negligible advantange.

3D Position Estimation Validation

To validate the 3D accuracy of the perception pipeline, a series of static tests were conducted outdoors on an asphalt surface. With the stereocamera fixed in place, a standard yellow competition cone was positioned at four known ground truth distances along the camera's forward-facing Z-axis: 5 m, 10 m, 15 m, and 20 m. For each position, 17 consecutive measurements were recorded to evaluate the stability and accuracy of the depth estimation for both the center point and ORB features methods.

The results of this experiment, showing the mean and standard deviation of the estimated Z-distance, are summarized in Table 5.5.

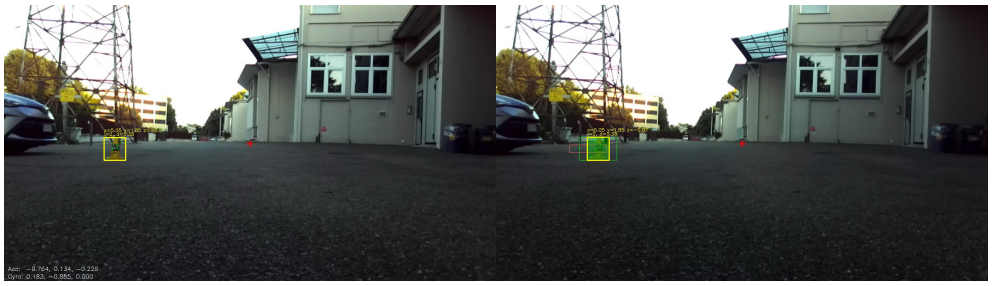


Figure 5.9: Yellow cone placed at 5 m in the experiment.

Figure 5.10: Yellow cone placed at 10 m in the experiment.

The analysis of these results provides several important insights into the pipeline's real-world performance:

- **Accuracy and Precision Decrease with Distance:** For both methods, the accuracy (how close the mean is to the ground truth) and the precision (the consistency of the measurements, indicated by the standard

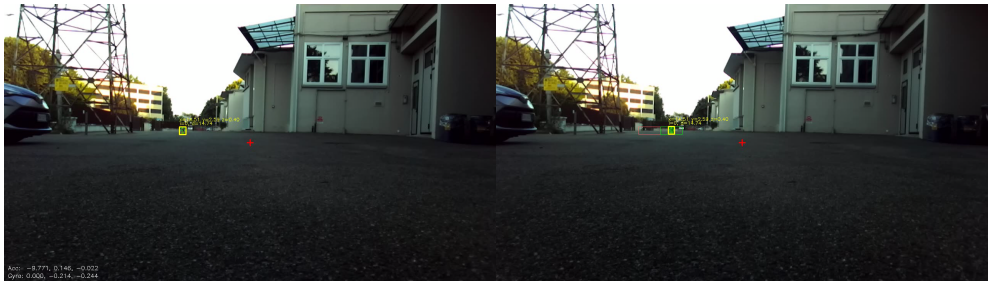


Figure 5.11: Yellow cone placed at 15 m in the experiment.

Figure 5.12: Yellow cone placed at 20 m in the experiment.

deviation) degrade as the distance to the cone increases. At 5 m, the estimates are highly accurate and stable. However, at 20 m, the mean estimation error approaches 2 meters, and the standard deviation exceeds 1 meter, indicating significant uncertainty and noise in the predictions.

- **Consistency of ORB vs. Center Point:** The key difference between the two methods lies in their consistency. At every tested distance, the standard deviation of the ORB-based method is significantly lower than that of the center point method. For instance, at 15 m, the ORB method's standard deviation (0.68 m) is nearly half that of the center point method (1.16 m). This demonstrates that using multiple feature points and taking the median provides a more stable and robust estimate, as it is less susceptible to noise and minor errors in the bounding box detection.

- **Confirmation of Sensor Limits:** The observed errors are consistent with the camera manufacturer's specifications, which state a depth accuracy of less than 7% at 20 m (an error of up to 1.4 m). Our results,

Table 5.5: Mean and standard deviation (in meters) of the estimated Z-distance for a cone placed at known ground truth distances. Results are shown for both the Center Point and ORB feature matching methods.

Ground Truth	Estimated Z-Distance: Mean (Std. Dev.)	
Distance	Center Point	ORB
5 m	5.07 (0.19)	5.08 (0.11)
10 m	9.56 (0.46)	9.87 (0.28)
15 m	14.25 (1.16)	13.95 (0.68)
20 m	18.04 (1.37)	18.35 (1.13)

with mean errors around 1.7-2.0 m at 20 m, align with this physical limitation. This confirms that for long-range perception tasks, the inherent accuracy of the sensor itself is a primary constraint on the system's performance.

In conclusion, while both methods provide comparable accuracy on average, the ORB feature-based approach offers a clear advantage in terms of measurement stability and reliability, especially at medium to long distances.

5.3 The Stack's Vehicle Dynamics Model

An accurate vehicle dynamics model is essential for developing both simulation environments and advanced control algorithms like Model Predictive Control (MPC). This section presents the single-track (or *bicycle*) model used in our software stack. The model is heavily inspired by the work of ETH Zurich's AMZ racing team [31, 25] but has been adapted to fit the specific characteristics of our vehicle, Athena.

5.3.1 Model Formulation

The model describes the vehicle's motion using a state-space representation. It operates in a vehicle-centric coordinate frame, with the X-axis pointing forward, the Y-axis to the left, and the Z-axis upward, originating at the vehicle's

center of mass.

The **state vector** $\mathbf{x}$ defines the condition of the vehicle at any given moment and consists of eleven variables:

$$\mathbf{x} = [s, n, \mu, v_x, v_y, r, F_{l,F}, F_{r,F}, F_{l,R}, F_{r,R}, \delta]^T$$

The components of the state vector are:

- s, n, μ: The vehicle's position relative to a predefined race line. s is the progress along the path, n is the lateral deviation (error) from the path, and μ is the heading angle error relative to the path's tangent.
- v_x, v_y, r: The vehicle's longitudinal velocity, lateral velocity, and yaw rate, respectively, in the body-fixed frame.
- $F_{l,F}, F_{r,F}, F_{l,R}, F_{r,R}$: The longitudinal forces applied to the front-left, front-right, rear-left, and rear-right wheels.
- δ: The steering angle of the front wheel. In this single-track model, this represents the steering angle for the combined front axle. A more complex four-wheel model would use the Ackermann steering geometry to calculate slightly different angles for the inner and outer front wheels, which is how a real car like Athena operates.

The **input vector** $\mathbf{u}$ represents the changes applied to the system by the controller:

$$\mathbf{u} = [\Delta F_{l,F}, \Delta F_{r,F}, \Delta F_{l,R}, \Delta F_{r,R}, \Delta\delta]^T$$

By defining the inputs as the *rate of change* of forces and steering angle, we can directly impose constraints on how quickly these values can change within the MPC algorithm. This helps generate smoother and more physically realistic control actions.

The dynamics of the system are described by the following set of differential equations:

$$\dot{s} = \frac{v_x \cos\mu - v_y \sin\mu}{1 - n\,\kappa(s)}, \tag{5.1}$$

$$\dot{n} = v_x \sin\mu + v_y \cos\mu, \tag{5.2}$$

$$\dot{\mu} = r - \kappa(s)\,\dot{s}, \tag{5.3}$$

$$\dot{v}_x = \frac{1}{m}\Big((F_{l,F} + F_{r,F})\cos\delta + F_{l,R} + F_{r,R} - F_{y,F}\sin\delta \tag{5.4}$$

$$+ m\,v_y\,r - F_{\text{fric}}\Big), \tag{5.5}$$

$$\dot{v}_y = \frac{1}{m}\Big((F_{l,F} + F_{r,F})\sin\delta + F_{y,F}\cos\delta + F_{y,R} - m\,v_x\,r\Big), \tag{5.6}$$

$$\dot{r} = \frac{1}{I_z}\Big(\big((F_{l,F} + F_{r,F})\sin\delta + F_{y,F}\cos\delta\big)\,l_F - F_{y,R}\,l_R\Big), \tag{5.7}$$

$$\dot{F}_{l,F} = \Delta F_{l,F}, \quad \dot{F}_{r,F} = \Delta F_{r,F}, \quad \dot{F}_{l,R} = \Delta F_{l,R}, \quad \dot{F}_{r,R} = \Delta F_{r,R}, \tag{5.8}$$

$$\dot{\delta} = \Delta\delta. \tag{5.9}$$

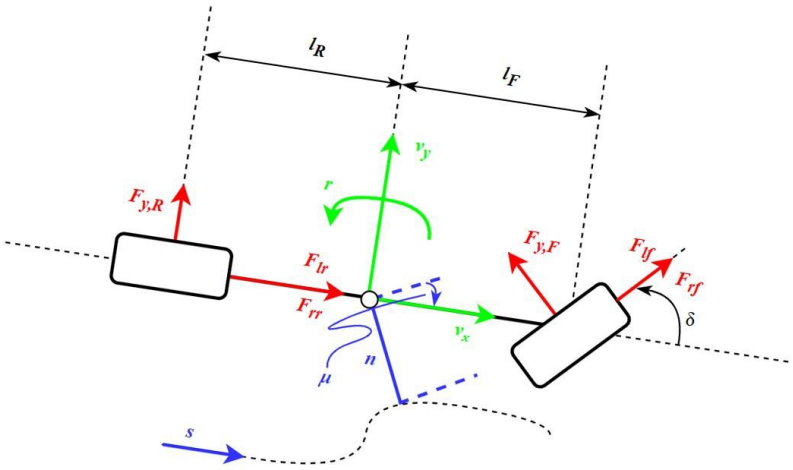


Figure 5.13: A diagram of the single-track (bicycle) model, showing the key state variables and forces. The forces F_{lf}, F_{rf}, F_{lr}, F_{rr} in the diagram are actually $F_{l,F}$, $F_{r,F}$, $F_{l,R}$, $F_{r,R}$ in the dynamics equations.

The key parameters defining the model's geometry and physics include l_F and l_R, the distances from the vehicle's center of mass to the front and rear axles, respectively; $k(s)$, the curvature of the reference path at a given progress s; and I_z, the vehicle's yaw moment of inertia. The terms mv_yr and mv_xr are inertial (centripetal) forces that naturally arise when applying Newton's laws in a rotating, body-fixed frame.

5.3.2 Longitudinal Force Modeling

The longitudinal forces acting on the vehicle are a combination of powertrain-generated forces and resistance forces.

Resistance Forces

The primary resistance forces are rolling resistance (F_{roll}) and aerodynamic drag (F_{aero}), which combine to form the total friction force, F_{fric}.

$$
\begin{aligned}
F_{roll} &= C_{roll}mg \\
F_{aero} &= \left(\frac{1}{2}\rho C_d A\right) v_x^2 = C_{aero}v_x^2 \\
F_{fric} &= F_{roll} + F_{aero}
\end{aligned}
$$

Here, ρ is the air density, C_d is the drag coefficient, and A is the vehicle's frontal area.

Powertrain Forces

Athena is a rear-wheel-drive electric vehicle. When a torque command is issued, the inverters draw DC power from the battery pack, convert it to AC, and drive the two rear motors. Consequently, propulsive forces are only applied to the rear wheels, so $F_{l,F}$ and $F_{r,F}$ are always zero. These terms are kept in the model to facilitate a future transition to a four-wheel-drive system.

To accurately model the conversion from electric torque to longitudinal

force at the wheels, the effect of vertical load on the tire radius must be considered. The model for this is as follows:

$$
\begin{aligned}
F_{z,F} &= mgl_R/(l_F + l_R) \quad \text{and} \quad F_{z,R} = mgl_F/(l_F + l_R) \\
R_{e,F} &= R_0 - (F_{z,F}/C_e) \cdot D_e \cdot \arctan(B_e) \\
R_{e,R} &= R_0 - (F_{z,R}/C_e) \cdot D_e \cdot \arctan(B_e) \\
k_f &= i_g \cdot \eta / R_{e,F} \quad \text{and} \quad k_r = i_g \cdot \eta / R_{e,R} \\
F_{l/r,F} &= \tau_{l/r,F} \cdot k_f \quad \text{and} \quad F_{l/r,R} = \tau_{l/r,R} \cdot k_r
\end{aligned}
$$

First, the static vertical loads on the front and rear axles, $F_{z,F}$ and $F_{z,R}$, are calculated. These loads compress the tires, reducing their radius. The **effective rolling radius**, $R_{e,F}$ and $R_{e,R}$, is then calculated using a Pacejka-like formula, where R_0 is the unloaded tire radius and B_e, C_e, D_e are empirical coefficients. This effective radius is then used to compute the reduction gains, k_f and k_r, which convert the input electric torques (τ) into longitudinal forces at the wheels, accounting for the gear ratio (i_g) and drivetrain efficiency (η). Since these gain values depend only on static parameters, they can be precalculated to improve computational performance.

5.3.3 Lateral Force Modeling

The lateral forces, $F_{y,F}$ and $F_{y,R}$, generated by the tires during cornering are highly non-linear and less straightforward to compute than longitudinal forces. To achieve a realistic estimation, we use the well-established Pacejka *Magic Formula* [18]. This analytical formula provides a robust and widely used approximation of tire behavior by fitting a curve to experimental data.

The core of the formula is the tire's **slip angle**, α, which is the difference between the direction a wheel is pointing and its actual direction of travel. For

our model, the front and rear slip angles are calculated as:

$$\alpha_F = \arctan\left(\frac{v_y + l_F r}{v_x}\right) - \delta$$
$$\alpha_R = \arctan\left(\frac{v_y - l_R r}{v_x}\right)$$

These slip angles are then fed into a specific variant of the Pacejka formula to determine the lateral force generated by each axle:

$$F_y = F_z D \sin\Big(C \arctan\Big(B\alpha - E(B\alpha - \arctan(B\alpha))\Big)\Big)$$

Here, F_z is the vertical load on the axle, and B, C, D, E are the Pacejka parameters that define the specific shape of the tire's force curve. These parameters are typically determined from experimental tire testing data. In our case, these equations are:

$$F_{y,F} = F_{z,F} D_F \sin\Big(C_F \arctan\Big(B_F\alpha_F - E_F(B_F\alpha_F - \arctan(B_F\alpha_F))\Big)\Big)$$
$$F_{y,R} = F_{z,R} D_R \sin\Big(C_R \arctan\Big(B_R\alpha_R - E_R(B_R\alpha_R - \arctan(B_R\alpha_R))\Big)\Big)$$

5.3.4 Model Parameters

The physical parameters used in the model are a combination of measured values from the vehicle and estimates from prior work. The complete list of parameters is provided in Table 5.6.

5.3.5 Model Justification and Variants

Justification for a Simplified Model The choice of a single-track model represents a deliberate trade-off between realism and computational expense. While more complex models (e.g., four-wheel models that include load transfer and Ackermann steering) offer higher fidelity, they also introduce many more parameters that are difficult to identify accurately without extensive

Table 5.6: Parameters of the vehicle dynamics model.

Parameter	Value	Unit
Vehicle Parameters		
Mass (m)	250.00	kg
Gravitational Acceleration (g)	9.81	m/s^2
Yaw Moment of Inertia (I_z)	107.03	kg·m^2
Unloaded Tire Radius (R_0)	0.23	m
Distance CoM to Front Axle (l_F)	0.89	m
Distance CoM to Rear Axle (l_R)	0.64	m
Resistance Parameters		
Rolling Resistance Coeff. (C_{roll})	0.01	-
Aerodynamic Drag Coeff. (C_{aero})	0.88	kg/m
Drivetrain Parameters		
Gear Ratio (i_g)	11.5	-
Drivetrain Efficiency (η)	0.95	-
Effective Radius Parameters		
Shape Factor (B_e)	20,000.00	-
Stiffness Factor (C_e)	77,117.00	N/m
Peak Factor (D_e)	0.28	-
Pacejka Tire Parameters		
Stiffness Factor ($B_{F/R}$)	16.30	-
Shape Factor ($C_{F/R}$)	1.35	-
Peak Factor ($D_{F/R}$)	2.50	-
Curvature Factor ($E_{F/R}$)	0.00	-

physical testing. The bicycle model is simple enough to be computationally efficient for real-time MPC, yet it captures the most important dynamic effects needed for high-performance path tracking.

Cartesian Coordinate System Variant While the path-relative coordinate system (s, n, μ) is ideal for the path-tracking formulation used in our MPC, the model can easily be adapted for use in a fixed Cartesian world frame for general-purpose simulation. To do this, the first three state equations governing path-relative motion (i.e., Equations 5.1-5.3) are replaced with the following, where (x, y) is the vehicle's global position and θ is its global yaw

angle:

$$\dot{x} = v_x \cos\theta - v_y \sin\theta \tag{5.10}$$

$$\dot{y} = v_x \sin\theta + v_y \cos\theta \tag{5.11}$$

$$\dot{\theta} = r \tag{5.12}$$

The remaining dynamic equations remain unchanged.

5.3.6 Model Validation

While the bicycle model is computationally efficient, it is crucial to validate its accuracy against a more realistic representation of the vehicle. To this end, the model's predictions were compared against those from a high-fidelity, multi-body simulation of Athena created in the professional software suite VI-grade CarRealTime. This validation focuses on one of the most critical aspects of vehicle dynamics: the generation of lateral tire forces.

The experiment involved simulating the vehicle at various constant longitudinal speeds (v_x) and steady-state steering angles (δ) in both models. The resulting lateral forces generated at the front ($F_{y,F}$) and rear ($F_{y,R}$) axles were then recorded and compared. The results of this comparison are shown in Table 5.7.

The analysis of these results reveals two key trends:

1. **Agreement at Low to Medium Speeds:** At lower speeds and moderate slip angles, the simplified bicycle model shows excellent agreement with the high-fidelity simulation. For instance, at 10 m/s with a steering angle of 0.1221 rad, the predicted forces from our model (846 N and 1160 N) are very close to the VI-grade results (820 N and 1195 N). This indicates that in the linear and early non-linear regions of tire behavior, our model effectively captures the vehicle's fundamental cornering dynamics.

Table 5.7: Comparison of steady-state lateral tire forces (in Newtons) between the custom bicycle model and the high-fidelity VI-grade simulation under various test conditions.

Test Condition		**Bicycle Model**		**VI-grade Simulation**	
v_x (m/s)	δ (rad)	$F_{y,F}$ (N)	$F_{y,R}$ (N)	$F_{y,F}$ (N)	$F_{y,R}$ (N)
3.00	0.1000	62	86	-	-
3.00	0.3000	200	263	-	-
10.00	0.0550	378	521	368	550
10.00	0.1000	691	949	-	-
10.00	0.1221	846	1160	820	1195
10.00	0.3000	2154	2842	-	-
20.00	0.0500	1373	1894	-	-
25.00	0.0174	746	1031	1145	1693
30.00	0.0200	1235	1706	-	-

2. **Divergence at High Speeds:** As the vehicle approaches its handling limits (e.g., the 25 m/s test case), a significant divergence between the models becomes apparent. Our model predicts much lower lateral forces (746 N and 1031 N) compared to the VI-grade simulation (1145 N and 1693 N). This discrepancy is expected and arises because the VI-grade model accounts for complex physical phenomena that our simplified model omits for computational reasons. These phenomena include dynamic vertical load transfer during cornering (which increases the grip of the outer tires), more sophisticated tire thermal models, and detailed suspension kinematics.

Despite these high-speed discrepancies, this validation confirms that our simplified bicycle model provides a sufficiently accurate representation of the vehicle's behavior for its intended application in the NMPC. Its computational efficiency is paramount for real-time operation, a task for which the high-fidelity VI-grade model would be far too slow. The model is most accurate in the operational regimes where the car will spend most of its time, making it a well-justified choice for the controller.

5.4 NMPC with the acados Library

This section provides a high-level overview of the implementation of a Nonlinear Model Predictive Controller (NMPC) for autonomous racing using the acados optimization library. We will first introduce the acados framework and our chosen development workflow, followed by the problem formulation for both a simplified kinematic model and the full dynamic model.

5.4.1 The acados Framework

The acados library [32] is highly flexible, offering several workflows for implementing an MPC. The main options are:

1. **Direct C Implementation:** This involves defining the Optimal Control Problem (OCP) and the vehicle model directly in C. This method offers the most control but requires a deep understanding of the library's internal workings.

2. **Python/MATLAB Interface with C Code Generation:** In this workflow, a high-level language like Python is used to define the OCP. acados then automatically generates highly optimized C code that solves this specific problem. This C code can then be easily linked into a larger C/C++ application, such as a ROS 2 node.

3. **High-Level Interface Only:** This option allows for the entire process, from OCP definition to solving, to be handled within a high-level language like Python or MATLAB. This is ideal for rapid prototyping but is not suitable for embedding the controller into a real-time C++ application.

Our team selected the second option, using the Python interface for code generation, for several practical reasons:

- **Development Time and Knowledge Transfer:** A pure C implementation would require a significant time investment to master. The Python interface, which uses the CasADi library for symbolic mathematics, abstracts much of this complexity. This makes it faster to develop the controller and easier for other team members to understand and contribute.

- **Performance:** Since the Python interface is only used for offline code generation, the final solver that runs on the vehicle is highly optimized C code. This approach avoids the performance overhead of interpreted languages during real-time execution, which is critical for meeting our tight time budgets.

- **System Integration:** Our controller must be a C++ ROS 2 node. The code generation workflow is perfectly suited for this, as it produces a self-contained C solver that can be seamlessly integrated into our existing C++ stack.

An important operational constraint is that the solver code must be compiled offline, before a mission begins. The on-board ASU lacks the power budget to perform code compilation during a run.

Ultimately, acados handles the complex numerical tasks, such as integrating the model dynamics over the prediction horizon. Our main task is to accurately define the OCP, which consists of the vehicle model, a cost function, and a set of constraints.

5.4.2 Kinematic Model Formulation

To introduce the OCP formulation, we first consider a simplified kinematic bicycle model.

Model Dynamics

The equations for the kinematic single-track model are given as:

$$\dot{x} = v \cos\theta \tag{5.13}$$

$$\dot{y} = v \sin\theta \tag{5.14}$$

$$\dot{\theta} = \frac{v}{l} \tan(\delta_{\text{target}}) \tag{5.15}$$

$$\dot{v} = K_P(v_{\text{target}} - v) \tag{5.16}$$

The state vector is $\mathbf{x} = [x, y, \theta, v]^T$, and the input vector is $\mathbf{u} = [v_{\text{target}}, \delta_{\text{target}}]^T$.

This model is purely kinematic, meaning it operates in a fixed Cartesian frame and does not account for forces, inertia, or tire slip. The state consists of the vehicle's 2D position (x, y), its absolute yaw angle (θ), and its longitudinal velocity (v). The vehicle's wheelbase is represented by l. The change in velocity is governed by a simple proportional controller with gain K_P, which drives the current velocity v towards the target velocity v_{target}. The target steering angle δ_{target} is assumed to be achieved by the actuator instantaneously.

A key feature of acados is its ability to work directly with non-linear, continuous-time models like this one. The MPC must ensure that the vehicle's predicted trajectory adheres to these dynamic equations. Instead of requiring the user to manually linearize or discretize the model, acados employs advanced numerical integration techniques (such as Runge-Kutta methods) to accurately simulate the model's behavior over the prediction horizon. This approach provides a more precise representation of the non-linear dynamics compared to simplified, linearized models, which is crucial for high-performance control.

If we define Equations 5.13-5.16 as $\dot{\mathbf{x}} = f_{\text{kin}}(\mathbf{x}, \mathbf{u})$, this system of equations forms the primary equality constraint for our OCP.

State and Input Constraints

The OCP must operate within a set of defined constraints. These ensure that the controller's solution is physically feasible and adheres to the vehicle's limits.

The primary state constraint is the initial condition. At the beginning of each control cycle, the MPC's prediction must start from the vehicle's current, most up-to-date state. We enforce this by setting $\mathbf{x}(0) = \mathbf{x}(t_0) = \hat{\mathbf{x}}$, where $\hat{\mathbf{x}}$ is the current state estimate provided by the localization system. In the acados framework, this is implemented by setting the lower and upper bounds for the initial state $\mathbf{x}(0)$ to be equal to $\hat{\mathbf{x}}$. These bounds are then updated at every iteration before the solver is called.

The inputs are constrained at every step of the prediction horizon. We define simple "box constraints," which set a fixed minimum and maximum value for each input:

$$v_{\text{target,min}} \leq v_{\text{target}}(t) \leq v_{\text{target,max}}$$
$$\delta_{\text{target,min}} \leq \delta_{\text{target}}(t) \leq \delta_{\text{target,max}}$$

These constraints limit the requested target velocity and steering angle to values that are safe and achievable by the vehicle's actuators.

Cost Function

The cost function defines the goal of the OCP. The solver's objective is to find a control sequence that minimizes this cost while respecting the model dynamics and constraints. The acados framework uses a standard structure composed of two parts: a path cost and a terminal cost.

- The **path cost** (or Lagrange term), denoted $l_{\text{kin}}(\mathbf{x}(t), \mathbf{u}(t))$, is applied at each step along the prediction horizon.
- The **terminal cost** (or Mayer term), denoted $m_{\text{kin}}(\mathbf{x}(T))$, is applied only

at the final step of the horizon, T.

The path cost is a quadratic function that penalizes the deviation of the predicted state and input from a desired reference trajectory:

$$l_{\text{kin}}(\mathbf{x}(t), \mathbf{u}(t)) = \frac{1}{2} \left\| \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{u}(t) \end{bmatrix} - \begin{bmatrix} \mathbf{y}_{\mathbf{x},\text{ref}}(t) \\ \mathbf{y}_{\mathbf{u},\text{ref}}(t) \end{bmatrix} \right\|_W^2$$

This formula calculates the squared error between the predicted state/input vector and the reference vector, weighted by the matrix W. Let's break down the components:

- $\mathbf{y}_{\mathbf{x},\text{ref}}(t)$: This is the reference state trajectory. For our problem, the reference positions (x, y) are sampled directly from the pre-calculated race line. Before each solver call, we find the closest point on the race line to the car's current position and then sample the next N points along the line to serve as the positional reference for the horizon. The reference values for heading (θ) and velocity (v) are left unspecified.

- $\mathbf{y}_{\mathbf{u},\text{ref}}(t)$: The reference for the control inputs is set to zero. This encourages the solver to find solutions that use minimal control effort, leading to smoother and more efficient driving.

- W: This is a diagonal weight matrix. The values on the diagonal determine how heavily the cost function penalizes deviations for each corresponding state and input variable. For example, a high weight on the positional error forces the controller to follow the race line closely. A weight of zero means a variable is not considered in the cost function; we use this for the θ and v states, as we do not have an explicit reference for them.

The terminal cost has a similar structure but applies only to the final state, $\mathbf{x}(T)$:

$$m_{\text{kin}}(\mathbf{x}(T)) = \frac{1}{2} \left\| \mathbf{x}(T) - \mathbf{y}_{\mathbf{x},\text{ref}}(T) \right\|_{W^e}^2$$

This cost uses a separate terminal weight matrix, W^e, which typically has larger weights to heavily penalize any deviation from the reference path at the end of the horizon, promoting stability.

The total cost, J_{kin}, that the solver seeks to minimize is the sum of the path costs over the horizon plus the final terminal cost:

$$J_{kin} = \left(\sum_{i=0}^{N-1} l_{\text{kin}}(\mathbf{x}(t_i), \mathbf{u}(t_i)) \right) + m_{\text{kin}}(\mathbf{x}(t_N)) \tag{5.17}$$

where $t_i = i\Delta t$ and Δt is the time interval between each step (or shooting node) i. Δt, for this problem formulation, is constant over the whole prediction horizon.

Optimal Control Problem Formulation

The concepts described in the previous sections can be combined into a formal Optimal Control Problem (OCP). The goal of the OCP is to find the optimal sequence of control inputs that minimizes the cost function while satisfying all constraints. Using a discrete-time formulation with N steps over the prediction horizon, the problem can be written as follows:

$$\min_{\mathbf{X},\mathbf{U}} \quad J_{\text{kin}} \tag{5.18}$$

$$\text{s.t.} \quad \mathbf{x}(t_{i+1}) = F_{\text{kin}}(\mathbf{x}(t_i), \mathbf{u}(t_i)), \qquad \forall i \in [0, N-1] \tag{5.19}$$

$$\mathbf{x}(0) = \hat{\mathbf{x}} \tag{5.20}$$

$$\mathbf{u}(t_i) \in \mathbf{U}, \qquad \forall i \in [0, N-1] \tag{5.21}$$

Each component of this formulation corresponds to the concepts we have discussed:

- **Objective Function** (5.18)**:** The primary objective is to find the state trajectory ($\{\mathbf{x}(t_i) \in \mathbf{X} \,|\, i \in [0, N]\}$) and control trajectory ($\{\mathbf{u}(t_i) \in \mathbf{U} \,|\, i \in [0, N-1]\}$) that minimize the total cost J_{kin}, which is the sum

of the path and terminal costs.

- **System Dynamics** (5.19)**:** This is the core equality constraint. It mandates that the state at the next step, $\mathbf{x}(t_{i+1})$, must be the result of applying the control input $\mathbf{u}(t_i)$ to the current state $\mathbf{x}(t_i)$. The function F_{kin} represents the discrete-time system dynamics, which is the result of integrating the continuous-time model f_{kin} over a single time step, Δt.

- **Initial State Constraint** (5.20)**:** This constraint initializes the OCP by forcing the first state of the prediction horizon, $\mathbf{x}(0)$, to be equal to the vehicle's current estimated state, $\hat{\mathbf{x}}$. This anchors the prediction to the real world.

- **Input Constraints** (5.21)**:** This requires that the control input at every step, $\mathbf{u}(t_i)$, must lie within the set of admissible inputs $\mathbf{U}$. In our case, this corresponds to the defined box constraints on target velocity and steering angle.

In summary, the acados solver searches for the sequence of control inputs $\mathbf{u}(t_0), \ldots, \mathbf{u}(t_{N-1})$ that steers the system along a trajectory $\mathbf{x}(t_0), \ldots, \mathbf{x}(t_N)$ that adheres to the vehicle's dynamics and limits, while minimizing the cost of deviating from the desired race line.

Experiments and Results

To evaluate the performance of the kinematic NMPC, it was compared against a standard Proportional-Derivative (PD) path-following controller. Both controllers were tested in a simulation environment using the same predefined race line. The primary performance metric was the lateral deviation from this reference path.

Controller Parameters The NMPC was configured with the following parameters:

- Proportional gain for velocity: $K_P = 10.0$
- Prediction horizon: $N = 40$ steps
- Time step: $\Delta t = 0.05$ s
- Input constraints ($\mathbf{U}$): $v_{\text{target}} \in [-8.0, 8.0]$ m/s, $\delta_{\text{target}} \in [-0.46, 0.46]$ rad
- Cost function weights: The diagonal of the weight matrix W was set to $[50.0, 50.0, 0, 0, 1.0, 1.0]$ for the state errors (x, y, θ, v) and input errors $(v_{\text{target}}, \delta_{\text{target}})$, respectively. The terminal weight matrix W^e was set to $10 \times W$.

The PD controller was configured with a proportional gain $k_p = 0.9$ and a derivative gain $k_d = 0.0$. Its lookahead distance was dynamically calculated based on the vehicle's speed.

Experimental Conditions A key aspect of this experiment is the low constant speed of 1 m/s. This speed is not an arbitrary choice but a direct consequence of the OCP formulation. The reference trajectory for the MPC is sampled from a race line where points are spaced 0.05 m apart. With a prediction horizon of $N = 40$ steps, the total reference path length over the horizon is 40×0.05 m $= 2.0$ m. The total time for this horizon is $T = N \times \Delta t = 40 \times 0.05$ s $= 2.0$ s. To minimize the tracking cost, the solver is naturally incentivized to find a solution that covers 2.0 meters in 2.0 seconds, which results in an average speed of 1.0 m/s. For a fair comparison, the PD controller's target speed was therefore also fixed at 1 m/s.

Results The performance of both controllers is summarized in Table 5.8.

The NMPC controller demonstrates a marginal but consistent improvement over the PD controller across all metrics. It achieves a lower maximum deviation (0.055 m vs. 0.061 m) and a slightly better mean deviation and standard deviation. This suggests that even with a simplified kinematic model,

Table 5.8: Comparison of lateral deviation from the race line between the NMPC and a PD controller at a constant speed of 1 m/s.

Controller	Max Deviation	Mean Deviation	Std. Dev.
NMPC (Kinematic)	0.055 m	0.021 m	0.014 m
PD Controller	0.061 m	0.023 m	0.015 m

the predictive nature of MPC allows it to generate smoother and more precise control actions compared to the purely reactive PD controller.

While these results are promising, it is important to note that this comparison is performed at a very low speed. The true advantages of MPC, particularly its ability to handle constraints and complex non-linear dynamics, become much more apparent at higher speeds, which will be explored using the full dynamic model.

5.4.3 Dynamic Model Formulation

We now extend the NMPC formulation to use the more realistic dynamic bicycle model presented in Section 5.3. This model allows the controller to account for forces and tire slip, which is essential for generating control actions that operate closer to the vehicle's physical handling limits.

Constraints

With the dynamic model, we can impose more sophisticated constraints that directly relate to the vehicle's physical limits.

- **Track Boundaries:** To ensure the vehicle remains entirely within the track, we apply heading-dependent geometric constraints. These constraints define a "safe" corridor for the vehicle by ensuring its corners do not cross the track boundaries. The allowable lateral deviation, n, becomes smaller as the vehicle's heading error, μ, increases. This is

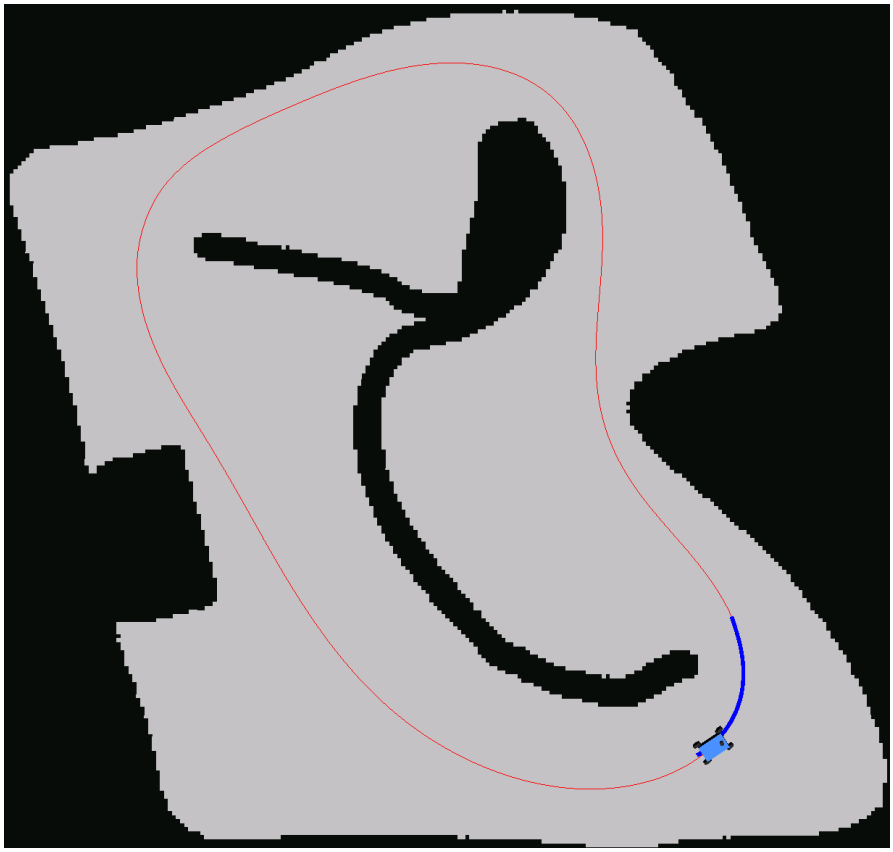


Figure 5.14: The NMPC controller operating in simulation. The vehicle (blue rectangle) follows an optimal state trajectory (blue line) computed by the acados solver. The controller's objective is to track the reference race line (red line) while respecting the vehicle's kinematic constraints.

expressed as:

$$n - \frac{L_c}{2}\sin|\mu| + \frac{W_c}{2}\cos\mu \leq N_L(s)$$
$$-n + \frac{L_c}{2}\sin|\mu| + \frac{W_c}{2}\cos\mu \leq N_R(s)$$

where L_c and W_c are the total length and width of the car, and $N_L(s)$ and $N_R(s)$ are the distances from the race line to the left and right track boundaries, respectively, at progress s.

- **Friction Ellipse:** To prevent the controller from demanding forces that the tires cannot physically produce, we apply a friction ellipse constraint. This constraint ensures that the combination of longitudinal and

lateral forces on each axle does not exceed the maximum available tire grip. The general form of the constraint is [5]:

$$\left(\frac{F_x}{\mu_x F_z}\right)^2 + \left(\frac{F_y}{\mu_y F_z}\right)^2 \leq 1$$

This inequality is applied independently to both the front and rear axles. For each axle:

- F_x is the total longitudinal force on the axle (e.g., $F_{l,F} + F_{r,F}$ for the front).
- F_y is the total lateral force on the axle (e.g., $F_{y,F}$ for the front).
- F_z is the total vertical load on the axle.
- μ_x is coefficient of sliding friction between the tire and the ground in the longitudinal direction.
- μ_y is coefficient of sliding friction in the longitudinal direction.

- **Input Constraints:** As with the kinematic model, we apply box constraints to the control inputs. These limit the rate of change of the longitudinal forces and the steering angle to ensure smooth and physically achievable commands:

$$\Delta F_{l,F,\text{min}} \leq \Delta F_{l,F}(t_i) \leq \Delta F_{l,F,\text{max}}$$
$$\Delta F_{r,F,\text{min}} \leq \Delta F_{r,F}(t_i) \leq \Delta F_{r,F,\text{max}}$$
$$\Delta F_{l,R,\text{min}} \leq \Delta F_{l,R}(t_i) \leq \Delta F_{l,R,\text{max}}$$
$$\Delta F_{r,R,\text{min}} \leq \Delta F_{r,R}(t_i) \leq \Delta F_{r,R,\text{max}}$$
$$\Delta \delta_{\text{min}} \leq \Delta \delta(t_i) \leq \Delta \delta_{\text{max}}$$

Cost Function

The objective for the dynamic model is to minimize lap time while maintaining stability. The total cost, J_{dyn}, is composed of a path cost l_{dyn} and a terminal

cost m_{dyn}. The path cost is a weighted sum of four key components:

$$l_{\text{dyn}}(\mathbf{x}(t_i), \mathbf{u}(t_i)) = -w_s\dot{s}(t_i) + w_n n(t_i)^2 + w_\beta(\beta_{\text{dyn}}(t_i) - \beta_{\text{kin}}(t_i))^2 + \mathbf{u}(t_i)^T R\mathbf{u}(t_i)$$

Each term has a specific purpose:

- $-w_s\dot{s}$: This term maximizes the progress rate along the path, $\dot{s}$, effectively minimizing the time taken to cover a certain distance. The weight w_s controls the aggressiveness of the controller.
- $w_n n^2$: This term penalizes the lateral deviation, n, from the reference race line, ensuring the vehicle follows the desired race line.
- $w_\beta(\beta_{\text{dyn}}(t_i) - \beta_{\text{kin}}(t_i))^2$: This term penalizes vehicle side slip. Instead of directly penalizing the individual front and rear tire slip angles, it minimizes the difference between the actual dynamic side slip angle of the vehicle body (β_{dyn}) and an idealized kinematic one (β_{kin}). The dynamic angle is calculated from the vehicle's velocity components as $\beta_{\text{dyn}} = \arctan(v_y/v_x)$, while the kinematic angle is an approximation based on the steering angle and vehicle geometry, given by $\beta_{\text{kin}} = \arctan(\delta\, l_R/(l_F + l_R))$. This encourages the vehicle to behave in a more stable and kinematically predictable manner.
- $\mathbf{u}^T R\mathbf{u}$: This term penalizes large control efforts, encouraging smooth changes to the steering angle and applied forces.

The terminal cost, $m_{\text{dyn}}(\mathbf{x}(T))$, is similar to $l_{\text{dyn}}(\mathbf{x}(t_i), \mathbf{u}(t_i))$:

$$m_{\text{dyn}}(\mathbf{x}(T)) = w_n n(T)^2 + w_\beta(\beta_{\text{dyn}}(T) - \beta_{\text{kin}}(T))^2$$

The total cost J_{dyn} has the same structure of Equation 5.17. The only necessary step is to replace the kinematic cost terms with the dynamic ones.

Optimal Control Problem Formulation

Combining the dynamic model, the new constraints, and the time-minimizing cost function, we can define the full OCP for high-performance driving:

$$\min_{\mathbf{X},\mathbf{U}} \quad J_{\text{dyn}} \tag{5.22}$$

$$\text{s.t.} \quad \mathbf{x}(t_{i+1}) = F_{\text{dyn}}(\mathbf{x}(t_i), \mathbf{u}(t_i)), \qquad \forall i \in \{0, \dots, N-1\} \tag{5.23}$$

$$\mathbf{x}(t_0) = \hat{\mathbf{x}} \tag{5.24}$$

$$\mathbf{u}(t_i) \in \mathbf{U}, \qquad \forall i \in \{0, \dots, N-1\} \tag{5.25}$$

$$\mathbf{x}(t_i) \in \mathbf{X}_{\text{Track}}, \qquad \forall i \in \{1, \dots, N\} \tag{5.26}$$

$$\mathbf{x}(t_i) \in \mathbf{X}_{\text{FE}}, \qquad \forall i \in \{1, \dots, N\} \tag{5.27}$$

where F_{dyn} is the discrete-time representation of the dynamic vehicle model, and $\mathbf{X}_{\text{Track}}$ and $\mathbf{X}_{\text{FE}}$ represent the feasible sets defined by the track boundary and friction ellipse constraints, respectively. This formulation directs the solver to find a control policy that maximizes progress along the track, while respecting the vehicle's dynamic limits, staying within the track boundaries, and avoiding tire slippage.

5.5 Online Hyperparameter Tuning with RL

Reinforcement Learning (RL) is a powerful paradigm within machine learning that enables an agent to learn optimal behavior by interacting with an environment. While applying RL for direct end-to-end vehicle control is currently infeasible for our project due to significant challenges in safety, interpretability, and generalization, its exceptional ability to find complex patterns can still be harnessed.

Our approach, therefore, is to use RL not to replace our classical controllers but to enhance them. We treat a trusted controller, like the PD path

follower, as part of the environment. An RL agent is then tasked with observing the vehicle's state and dynamically adjusting the controller's hyperparameters to optimize its performance in real-time. This hybrid method allows us to leverage the pattern-matching capabilities of deep learning without sacrificing the safety and predictability of a well-understood control algorithm.

This section details the application of this concept to the PD path-following controller.

5.5.1 Markov Decision Process for PD Controller Tuning

To apply any RL algorithm, we must first formally define the problem as a Markov Decision Process (MDP). This involves specifying the state space, action space, transition function, and reward function.

A crucial design choice is the use of a *relative* state representation. Instead of using the vehicle's absolute coordinates on the track, the state is defined relative to the geometry of the upcoming race line. This approach allows the agent to learn generalizable behaviors that are independent of any specific track layout.

State Representation

The state vector, $\mathbf{s}$, is composed of the vehicle's immediate condition relative to the race line, as well as a lookahead buffer describing the path's future geometry.

$$\mathbf{s} = [n, \kappa, v_x, v_y, \mu, r, d_{\text{next}}]^T \oplus \bigoplus_{j=1}^{N_h} [n_{R,j}, n_{L,j}, \kappa_{h,j}, \Delta\mu_{h,j}, v_{\text{ref},j}]^T$$

where the $\oplus$ and $\bigoplus$ symbols mean vector concatenation in this context. The components are:

- n, κ, μ: The current lateral deviation, path curvature, and relative heading error with respect to the closest point on the race line.

- v_x, v_y, r: The vehicle's longitudinal velocity, lateral velocity, and yaw rate.
- d_{next}: The Euclidean distance from the car's current position to the next point on the race line (therefore, the one at progress $s + 1$).
- The second part of the state vector describes the path for a horizon of N_h future points, spaced by a distance of Δs_h along the race line. For each future point j, the state includes:
  - $n_{R,j}, n_{L,j}$: The distance from the race line to the right and left track boundaries.
  - $\kappa_{h,j}$: The curvature of the race line.
  - $\Delta\mu_{h,j}$: The change in the race line's yaw angle relative to the current race line point heading (the one at progress s).
  - $v_{\text{ref},j}$: The pre-calculated target speed from the race line's speed profile.

Action Space

The agent's action, $\mathbf{a}$, is a continuous vector corresponding to the hyperparameters of the PD controller:

$$\mathbf{a} = [k_p, k_d, d_{\text{clip}}, g_{\text{la}}, d_{\text{la,min}}, g_v]^T$$

where:

- k_p, k_d: The proportional and derivative gains of the controller.
- d_{clip}: A limit applied to the derivative action to prevent instability.
- g_{la}: A gain multiplied by the current velocity, v_x, to compute the lookahead distance.
- $d_{\text{la,min}}$: The minimum lookahead distance.

- g_v: A multiplicative gain applied to the reference speed from the pre-calculated race line.

What about the dynamic model? If we had to use the same approach for the dynamic MPC formulation in 5.4.3, the action vector would change to:

$$\mathbf{a} = [w_s, w_n, w_\beta]^T \oplus \mathrm{vec}(R)$$

where $\mathrm{vec}(R)$ is the vectorization of the matrix R.

Transition Function

The environment's transition from one state to the next is handled by a deterministic physics simulator, which is based on the single-track model from CommonRoad [1]. A single step in the MDP unfolds as follows:

1. The RL agent observes the current environment state $\mathbf{s}$ and outputs an action $\mathbf{a}$, which is the set of new hyperparameters.
2. The PD controller's parameters are updated with the values from $\mathbf{a}$.
3. The newly configured PD controller computes a control command (e.g., a target steering angle).
4. This command is translated into the inputs required by the physics simulator (e.g., steering velocity and longitudinal acceleration).
5. The simulator integrates the vehicle's physics model over one time step, advancing the physical state from $\mathbf{x}$ to $\mathbf{x}'$.
6. The new environment state, $\mathbf{s}'$, is then calculated based on the new physical state $\mathbf{x}'$ and its relation to the race line.

Reward Function

The reward function, $R(\mathbf{s}, \mathbf{a}, \mathbf{s}')$, is designed to be smooth and to encourage maximizing progress along the track while penalizing collisions:

$$R(\mathbf{s}, \mathbf{a}, \mathbf{s}') = \begin{cases} \Delta s - (w_{\mu,m}|\mu| + w_{\mu,c}) & \text{if collision occurs} \\ \Delta s & \text{otherwise} \end{cases}$$

Here, Δs is the progress made along the race line during the transition, directly rewarding the agent for moving forward. If the car collides with a track boundary, a penalty is applied, which consists of a constant term $w_{\mu,c}$ and a term proportional to the vehicle's heading error μ at the moment of impact. It is important to distinguish the environment state vector, $\mathbf{s}$, from the scalar progress, s.

5.5.2 Algorithm Choice: Proximal Policy Optimization (PPO)

For this continuous control problem, we selected the Proximal Policy Optimization (PPO) algorithm [24]. PPO is a policy gradient method, making it well-suited for problems with continuous state and action spaces, which are key features of our MDP.

The primary advantage of PPO lies in its clipped surrogate objective function. In practice, this mechanism prevents the algorithm from making excessively large changes to the policy during a single training update. Tuning a controller's parameters online is a sensitive task; aggressive adjustments can lead to unstable or oscillatory vehicle behavior. PPO's clipped objective ensures that learning proceeds in small, stable steps, which is critical for maintaining robust control.

Furthermore, PPO offers an excellent balance between implementation simplicity, computational efficiency, and performance. It provides many of the stability benefits of more complex algorithms like Trust Region Policy

Optimization (TRPO) but is significantly easier to implement and tune. As an on-policy algorithm, PPO learns from data generated by its most recent policy, which integrates seamlessly with our simulation-based training workflow.

In summary, PPO's ability to handle continuous actions, its inherent stability, and its practical design make it a strong choice for the task of optimizing controller hyperparameters in our simulated environment.

5.5.3 Experiments and Results

To evaluate the PPO agent's ability to tune the PD controller, a series of experiments were conducted in a simulation environment. The agent was trained using a set of five distinct maps, with an additional sixth map reserved exclusively for testing to measure generalization performance. To avoid overfitting to a specific direction of travel, the race lines for the training maps were used in clockwise or counter-clockwise orientations. During training, each episode was initialized by placing the agent at a random position on the track with a heading roughly parallel to the race line, with small Gaussian noise added to encourage robustness. An episode finishes after a collision with the track boundaries or after having completed two laps.

Experimental Setup and Parameters

The experiments were configured with the following MDP and PPO parameters. For the PPO training, the library RL Baselines3 Zoo [20] has been used.

MDP Parameters:

- **Reward Weights:** $w_{\mu,m} = 10/\pi$, $w_{\mu,c} = 20$.
- **Update Frequency:** 1. The PPO agent updates the PD controller's parameters at every control step.
- **State Horizon:** Horizon steps $N_h = 25$, with a spacing of $\Delta s_h = 0.2$ m between points.

PPO Parameters:

- **Discount Factor (γ):** 0.99
- **Network Architecture:** The Actor and Critic networks both use a Multi-Layer Perceptron (MLP) architecture with three hidden layers of 256 neurons each and ReLU activation functions.
- **Learning Rate:** 3×10^{-4}
- **Parallel Environments (n_{envs}):** 16
- **Transition function steps per PPO update (n_{steps}):** $2048 \times n_{\text{envs}}$
- **Clip Range (ϵ):** 0.2

We now present the results from four different experimental configurations. Each validation run was performed over 50,000 simulation steps.

Baseline: Hand-Tuned PD Controller

The first experiment establishes a baseline using a single set of hand-tuned PD parameters ($\mathbf{a} = [0.9, 0.0, 0.2, 0.2, 0.4, 1.0]$), which remain fixed across all tracks. The goal is to demonstrate the limitations of a non-adaptive controller. The results are shown in Table 5.9 (for all the tables of this section, the "Mean Reward" column is the mean cumulative reward per episode without γ).

Table 5.9: Performance of the baseline PD controller with fixed, hand-tuned parameters on all six tracks. Standard deviation in parenthesis. Laps are measured in seconds.

Track	Mean Reward	Collision Rate	Best Lap	Mean Lap
Atlanta	-6.77 (12.12)	100.0%	-	-
CUSB	58.66 (31.85)	13.9%	7.67	7.85 (0.18)
London	72.65 (40.03)	15.6%	8.90	9.10 (0.20)
Milan	50.96 (25.63)	11.4%	6.19	6.38 (0.19)
Workshop	-11.47 (8.43)	100.0%	-	-
Test Track	-6.81 (9.98)	100.0%	6.68	6.71 (0.02)

The results clearly show that a single set of parameters is not optimal for all conditions. While the controller performs reasonably well on the CUSB,

London, and Milan tracks, it fails completely on the more challenging Atlanta and Workshop tracks, where it crashes in every episode. Even on the test track, it is highly unstable. This demonstrates the need for an adaptive approach, if we don't want to finetune the parameters manually for every track.

Single-Track Specialization

In this experiment, a separate PPO agent was trained for each of the five training maps. The goal was to verify that the agent is capable of learning to optimize the controller for a specific, known environment. The results are shown in Table 5.10.

Table 5.10: Performance of PPO agents trained and evaluated on a single, specific track after 2.5 million transition function steps. Standard deviation in parenthesis. Laps are measured in seconds.

Track	Mean Reward	Collision Rate	Best Lap	Mean Lap
Atlanta	43.34 (58.54)	96.2%	30.56	30.96 (0.27)
CUSB	59.32 (31.00)	13.3%	6.00	6.24 (0.20)
London	65.22 (45.66)	22.6%	9.99	10.23 (0.19)
Milan	46.62 (29.98)	16.9%	4.31	4.60 (0.25)
Workshop	66.76 (53.77)	27.8%	18.25	18.54 (0.38)

The results show a marked improvement. The PPO agent successfully learns to tune the PD controller to complete laps on all tracks, including the difficult Workshop track where the baseline failed. Furthermore, on the tracks where the baseline already succeeded (CUSB and Milan), the PPO agent achieves significantly faster lap times. This confirms that the PPO agent can effectively solve the hyperparameter optimization problem for a known track.

Generalization Across Multiple Tracks

This experiment tests the agent's ability to generalize. A single agent was trained on all five maps simultaneously and then evaluated on the unseen test map. The performance was recorded at different stages of training. The results are shown in Table 5.11.

Table 5.11: Performance of a single PPO agent on the unseen test map after being trained on five other maps. Standard deviation in parenthesis. Laps are measured in seconds.

Training Steps	Mean Reward	Collision Rate	Best Lap	Mean Lap
2.5 Million	36.91 (34.34)	26.5%	6.53	6.83 (0.26)
5.0 Million	16.79 (32.32)	67.0%	4.64	4.99 (0.24)
7.5 Million	-11.66 (8.25)	100.0%	5.09	5.14 (0.06)
10.0 Million	-14.87 (7.11)	100.0%	4.56	4.56 (0.00)

These results reveal a critical trend. As training progresses, the agent becomes more aggressive in pursuit of higher rewards. This is evident from the ”Best Lap Time,” which improves consistently, from 6.53s to an impressive 4.56s. However, this increase in speed comes at a severe cost to stability. The collision rate rises dramatically, from 26.5% to 100%, indicating that the agent is learning to push the controller beyond its stable operating limits. The agent likely learns to increase the speed gain (g_v) to maximize the Δs reward term, but the simple PD controller cannot handle the resulting high speeds, leading to frequent crashes.

Generalization with State Normalization

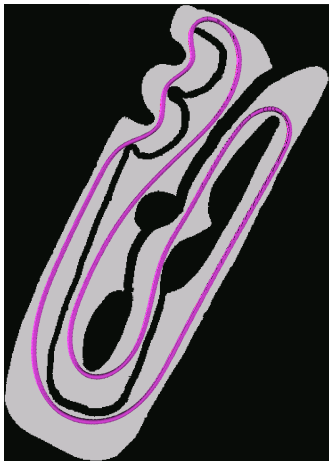
The final experiment investigated whether normalizing the state vector—a common technique in RL to stabilize training—could mitigate the instability observed in the previous experiment. The results are shown in Table 5.12.

Table 5.12: Performance on the test map of an agent trained with state normalization. Standard deviation in parenthesis. Laps are measured in seconds.

Training Steps	Mean Reward	Collision Rate	Best Lap	Mean Lap
2.5 Million	-5.44 (5.40)	99.5%	6.00	6.37 (0.24)
5.0 Million	-6.30 (3.31)	100.0%	6.07	6.22 (0.17)
7.5 Million	-6.51 (2.06)	100.0%	-	-

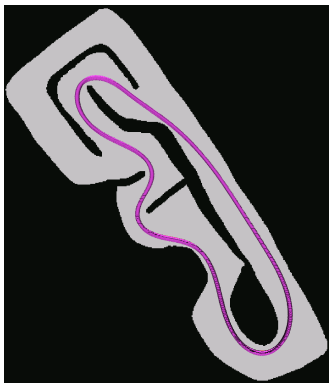
The results indicate that state normalization did not solve the underlying problem. The agent remains highly unstable, with collision rates near 100%

across all training stages. This suggests that the issue is not with the learning process itself, but rather with the fundamental limitations of the PD controller being tuned and maybe too simple definition of the MDP state $\mathbf{s}$ and reward function. The agent correctly learns that higher speed leads to higher rewards, but the simple controller it is tuning cannot safely handle those speeds.

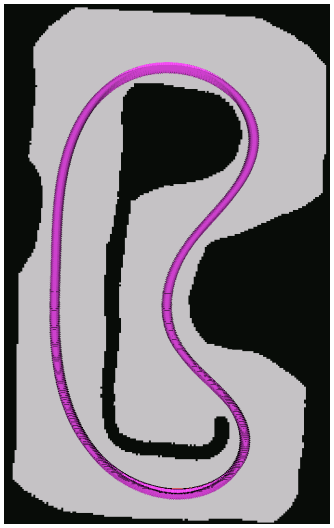


(a) Atlanta Map

(b) CUSB Map



(c) London Map



(d) Milan Map

(e) Workshop Map

(f) The Test map used for validation.

Figure 5.15: The six race track maps used in these experiments. The purple line indicates the calculated race line.

Chapter 6

Conclusions and Future Work

This thesis has detailed the design, implementation, and validation of a comprehensive software stack for an autonomous race car, developed in the context of the Formula SAE Driverless competition. As the first iteration of a driverless system for our team's electric vehicle, this work established a foundational architecture and explored key challenges in perception, planning, and control. This chapter summarizes the primary contributions of this research, discusses its current limitations, and outlines promising directions for future development.

6.1 Summary of Contributions

The main contributions of this thesis are distributed across several core areas of the autonomous stack. A robust, supervisory node, the ASU Manager, was designed and implemented to manage the lifecycle of all software processes, providing a reliable foundation for starting, stopping, and restarting the system. In the perception domain, a complete stereocamera pipeline was developed to detect and localize track cones in 3D. This pipeline includes a custom-trained YOLO object detector and a multi-stage stereo matching algorithm, where we demonstrated that incorporating ORB feature matching provides more stable 3D position estimates than using only bounding box centers.

On the modeling front, a high-fidelity dynamic bicycle model was formulated, incorporating a non-linear Pacejka tire model. This model, validated against a professional simulation suite, serves as a first iteration for a future accurate and computationally efficient basis for advanced control algorithms. This led to the design of a complete NMPC framework using the `acados` toolkit, for which formulations were developed for both a simple kinematic model and the full dynamic model. The kinematic NMPC was implemented and tested, demonstrating a performance improvement over a classical PD controller. Finally, this thesis explored a novel approach to controller enhancement by using a PPO reinforcement learning agent to dynamically tune the hyperparameters of a PD path-follower, providing valuable insights into the challenges and potential of applying learning-based optimization to classical control systems.

6.2 Limitations and Future Work

While this work establishes a solid foundation, several areas have been identified for improvement and further investigation. These represent the logical next steps for the project.

In perception, the current stereo matching pipeline is constrained by the physical limits of the sensor and classical computer vision techniques, with depth accuracy degrading significantly beyond 15 meters. A promising avenue for future work is to investigate end-to-end, deep learning-based approaches for depth estimation. Models like CREStereo [13] or RAFT-Stereo [14] could potentially learn to produce more robust and accurate depth maps by leveraging learned priors about the scene, especially in challenging lighting conditions.

In control, while the NMPC framework was fully formulated for the dynamic vehicle model, only the simplified kinematic version was implemented and tested. The low-speed nature of these tests did not fully showcase the

advantages of MPC. Therefore, the immediate next step is to implement and integrate the NMPC using the full dynamic model with `acados`. This will be the key to unlocking high-speed performance, as the controller will be able to reason about tire forces and vehicle dynamics at the limits of handling.

Regarding the reinforcement learning experiments, our results showed that the agent successfully optimized for speed but consequently pushed the simple PD controller into unstable regimes. This highlights a limitation not in the learning algorithm, but in the system it was tasked to optimize. The RL approach therefore requires further refinement. The reward function should first be augmented to include penalties for instability, such as high lateral acceleration or aggressive control inputs, to encourage the agent to find a balance between speed and stability. A more powerful application of this concept would then be to use the RL agent to tune the NMPC itself. The agent could learn to adjust the weights of the NMPC's cost function in real-time, adapting the controller's behavior to different sections of the track.

In conclusion, this thesis represents a successful first step in the development of a competitive driverless race car. The software architecture, perception pipeline, and control formulations presented here provide a robust and well-understood platform upon which future development can be built. The challenges encountered and the lessons learned have paved a clear path forward for achieving higher levels of performance and reliability in future competition seasons.

Bibliography

[1] M. Althoff and G. Würsching. The CommonRoad Vehicle Models Repository. Hosted on GitLab. URL: `https://gitlab.lrz.de/tum-cps/commonroad-vehicle-models/` (visited on 07/02/2025).

[2] A. Alvarez, N. Denner, Z. Feng, D. Fischer, Y. Gao, L. Harsch, S. Herz, N. L. Large, B. Nguyen, C. A. R. Pozo, S. Schaefer, A. Terletskiy, L. Wahl, S. Wang, J. Yakupova, and H. Yu. The Software Stack That Won the Formula Student Driverless Competition. *ArXiv*, abs/2210.10933, 2022.

[3] N. Baumann, E. Ghignone, J. Kühne, N. Bastuck, J. Becker, N. Imholz, T. Kränzlin, T. Y. Lim, M. Lötscher, L. Schwarzenbach, et al. Forza-ETH Race Stack—Scaled Autonomous Head-to-Head Racing on Fully Commercial Off-the-Shelf Hardware. *Journal of Field Robotics*, 2024.

[4] J. Betz, T. Betz, F. Fent, M. Geisslinger, A. Heilmeier, L. Hermansdorfer, T. Herrmann, S. Huch, P. Karle, M. Lienkamp, B. Lohmann, F. Nobis, L. Ögretmen, M. Rowold, F. Sauerbeck, T. Stahl, R. Trauth, F. Werner, and A. Wischnewski. TUM autonomous motorsport: An autonomous racing software for the Indy Autonomous Challenge. *Journal of Field Robotics*, 40(4):783–809, January 2023. ISSN: 1556-4967. DOI: `10.1002/rob.22153`.

[5] M. Brach and R. Brach. Modeling Combined Braking and Steering Tire Forces. *Proceedings of the 2000 Automotive Dynamics and Stability Conference-P-354*, 2000.

[6] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[7] M. Cellina, M. Corno, and S. M. Savaresi. Lidar-based vehicle detection and tracking for autonomous racing. *ArXiv*, abs/2501.14502, 2025.

[8] Z. Demeter, M. Hell, and G. Hajgató. Lessons Learned from an Autonomous Race Car Competition. *Engineering Proceedings*, 79(1), 2024. ISSN: 2673-4591. DOI: 10.3390/engproc2024079025. URL: https://www.mdpi.com/2673-4591/79/1/25.

[9] Formula Student Rules. URL: https://www.formulastudent.de/fsg/rules/ (visited on 07/02/2025).

[10] G. Jocher, J. Qiu, and A. Chaurasia. Ultralytics YOLO, version 8.0.0, January 2023. URL: https://github.com/ultralytics/ultralytics/.

[11] J. Kabzan, M. de la Iglesia Valls, V. Reijgwart, H. F. C. Hendrikx, C. Ehmke, M. Prajapat, A. Bühler, N. B. Gosala, M. Gupta, R. Sivanesan, A. Dhall, E. Chisari, N. Karnchanachari, S. Brits, M. Dangel, I. Sa, R. Dubé, A. Gawel, M. Pfeiffer, A. Liniger, J. Lygeros, and R. Siegwart. AMZ Driverless: The Full Autonomous Racing System. *CoRR*, abs/1905.05150, 2019. arXiv: 1905.05150. URL: http://arxiv.org/abs/1905.05150.

[12] M.-S. Kim and T.-H. Park. Model predictive control with reinforcement learning-based speed profile generation in racing simulator. *IEEE Access*, 2025.

[13] J. Li, P. Wang, P. Xiong, T. Cai, Z. Yan, L. Yang, J. Liu, H. Fan, and S. Liu. Practical stereo matching via cascaded recurrent network with adaptive correlation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16263–16272, 2022.

[14] L. Lipson, Z. Teed, and J. Deng. RAFT-Stereo: Multilevel Recurrent Field Transforms for Stereo Matching. In *International Conference on 3D Vision (3DV)*, 2021.

[15] Low level Linux camera driver for the ZED USB3 stereocameras. Hosted on GitHub. URL: `https://github.com/stereolabs/zed-open-capture/` (visited on 07/02/2025).

[16] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. DOI: `10.1126/scirobotics.abm6074`.

[17] A. A. Melnikov et al. Projective simulation applied to the grid-world and the mountain-car problem. *Artif. Intell. Res.*, 3(3):24–34, 2014. DOI: `10.5430/air.v3n3p24`. URL: `https://doi.org/10.5430/air.v3n3p24`.

[18] H. B. Pacejka. Chapter 4 - Semi-Empirical Tire Models. In *Tire and Vehicle Dynamics (Third Edition)*, page 165. Butterworth-Heinemann, Oxford, 2012. ISBN: 978-0-08-097016-5. DOI: `https://doi.org/10.1016/B978-0-08-097016-5.00004-8`.

[19] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994. ISBN: 978-0-47161977-2. DOI: `10.1002/9780470316887`. URL: `https://doi.org/10.1002/9780470316887`.

[20] A. Raffin. RL Baselines3 Zoo. `https://github.com/DLR-RM/rl-baselines3-zoo`, 2020.

[21] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection, 2016. arXiv: `1506.02640 [cs.CV]`. URL: `https://arxiv.org/abs/1506.02640`.

[22] A. Romero, Y. Song, and D. Scaramuzza. Actor-Critic Model Predictive Control. *2024 IEEE International Conference on Robotics and Automation (ICRA)*:14777–14784, 2023. URL: https://api.semanticscholar.org/CorpusID:259187711.

[23] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: An efficient alternative to SIFT or SURF. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011. DOI: 10.1109/ICCV.2011.6126544.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[25] S. Srinivasan, S. N. Giles, and A. Liniger. A holistic motion planning and control solution to challenge a professional racecar driver. *IEEE Robotics Autom. Lett.*, 6(4):7854–7860, 2021. DOI: 10.1109/LRA.2021.3101244.

[26] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[27] The Global Race Trajectory Optimization Repository. Hosted on GitHub. URL: https://github.com/TUMFTM/global_racetrajectory_optimization (visited on 07/07/2025).

[28] The OpenVINO Repository. Hosted on GitHub. URL: https://github.com/openvinotoolkit/openvino/ (visited on 07/02/2025).

[29] The TensorRT Repository. Hosted on GitHub. URL: https://github.com/NVIDIA/TensorRT (visited on 07/02/2025).

[30] The Unity Website. URL: https://unity.com/ (visited on 07/02/2025).

[31] J. L. Vázquez, M. Brühlmeier, A. Liniger, A. Rupenyan, and J. Lygeros. Optimization-based hierarchical motion planning for autonomous racing. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*, pages 2397–2403. IEEE, 2020. DOI: 10.1109/IROS45743.2020.9341731.

[32] R. Verschueren, G. Frison, D. Kouzoupis, J. Frey, N. van Duijkeren, A. Zanelli, B. Novoselnik, T. Albin, R. Quirynen, and M. Diehl. Acados – a modular open-source framework for fast embedded optimal control. *Mathematical Programming Computation*, 2021.

[33] N. Vödisch, D. Dodel, and M. Schötz. Fsoco: the formula student objects in context dataset. *SAE International Journal of Connected and Automated Vehicles*, 5(12-05-01-0003), 2022.

Acknowledgements

I’m very grateful to my family, who gave me the opportunity to reach this level of education, to the friends that I met along the way, to myself, who never considered forgetting the objective, and to my professor, who always gave me insights and pieces of advice on this project.

And finally, thanks to the UniBo Motorsport team. Together, over the last three years, we have really achieved important and unbelievable milestones. I really hope the team will continue to grow from these solid foundations.