Squadra Corse Driverless

Autonomous System
Documentation

Politecnico di Torino

November 4, 2024

Contents

1 Introduction

The purpose of this document is to provide a comprehensive technical overview of the autonomous system by the Squadra Corse Driverless (SCD) team from Politecnico di Torino. The team focuses on the design, development, and testing of a fully autonomous electric race car, aiming to compete in Formula SAE competitions.

This document is intended to:

- Present the competition and dynamic events
- Present the hardware equipment.
- Present an overview of the autonomous system.
- Dive into the specific problems of the autonomous system.
- Describe the implementation and deployment decisions.
- Introduce the simulation environment.
- Describe the key design decisions, challenges, and solutions implemented during the project.

2 Formula SAE Competition

Formula SAE (FSAE) is an international engineering competition held annually, where teams of university students from around the world design, build, and race formula-style racing cars. The competition challenges teams to demonstrate their engineering skills not only through vehicle performance on the track but also through a series of static events, including design presentations and cost analysis.

The FSAE rules [10] provide guidelines that outline all design constraints as well as the structure of static and dynamic events.

The DV (Driverless Vehicle) class competition consists of the subsequent dynamic events:

- **Acceleration**: evaluates the vehicle acceleration in a 75 m long and 4.9 m wide straight line on flat pavement.
- **Skidpad**: measures the vehicle cornering ability on a flat surface while making a constant radius turn.

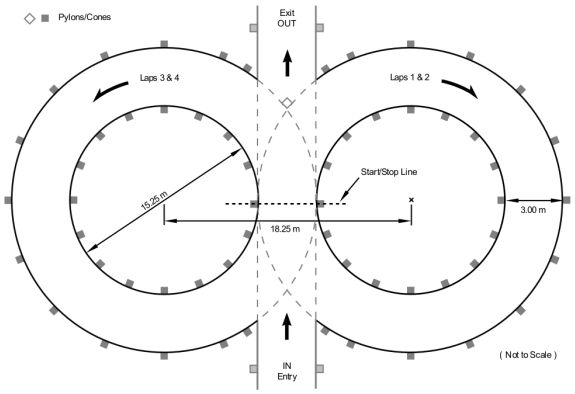

Figure 1: Skidpad track specifications

- **Autocross**: evaluates the vehicle racing on an unknown track for one lap.
- **Trackdrive**: evaluates the vehicle racing on the same autocross track for ten laps.

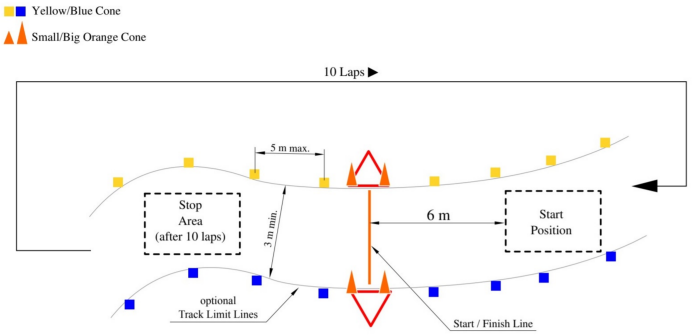

Figure 2: Autcross/Trackdrive track specifications

3 Hardware equipment

3.1 LiDAR

LiDAR (Light Detection and Ranging) is a remote sensing technology that uses laser light to measure distances to objects or surfaces. It works by emitting rapid pulses of laser light and measuring the time it takes for these pulses to bounce back after hitting an object. By analyzing this return time, LiDAR systems can accurately calculate distances and create detailed 3D models of the environment.

By knowing the speed of light, the system computes the distance between the LiDAR sensor and the hit object based on the time delay. Distance measurements are combined with the sensor's orientation and location data to create a dense point cloud describing the environment.

Figure 3: LiDAR Ouster OS1 64-U

Ouster OS1 64-U specifications:

- Vertical channels: 64
- Points per vertical channel: 1024
- Acquisition rate: 10 Hz / 20 Hz
- Horizontal field of view: 360°
- Vertical field of view: 45°
- Horizontal resolution: $\simeq 0.35°$
- Vertical resolution: $\simeq 0.7°$
- Range: 0.5 m – 200 m

For our application, we opted for a 10 Hz acquisition rate to maintain resolution, and the horizontal field of view is cropped to 180° in front of the vehicle.

3.2 Cameras

A dual-camera system is employed to capture images of the surrounding environment. The cameras are oriented forward, configured with a tight overlap to maximize the effective field of view.

Figure 4: Camera Alvium 1800 U-507

Alvium 1800 U-507 specifications:

- Resolution: 2464 x 2056
- Maximum acquisition rate: 35 fps
- Horizontal field of view: $\simeq 60°$
- Format: RGB8
- Features: Auto exposure, auto gain, ROI cropping.

Cameras can acquire images on a precise ROI (Region Of Interest) cropping, by hardware, the acquired image before send it. In our implementation, the images are cropped to a resolution of 1000 x 2056, to ensure maximum frame rate, discarding irrelevant informations by the images. Images are then rescaled by 1/3 (333 x 685) to reduce the amount of data sent between processes. The combined effective field of view is approximately 120°.

3.3 IMU

The Inertial Measurement Unit (IMU) is a sensor system used to measure and report a body's specific force, angular velocity, and the magnetic field surrounding the body. IMUs provide essential data for determining orientation and position in three-dimensional space.

An IMU consists of three types of sensors:

- **Accelerometers**: Measure linear acceleration along x, y and z axes allowing the IMU to determine speed and displacement when integrated over time.
- **Gyroscopes**: Measure angular velocity, which is the rate of rotation around x, y and z axes tracking the orientation (respectively roll, pitch and yaw) by integrating angular velocity over time.
- **Magnetometers**: Measure the magnetic field strength and direction, providing an additional reference point for orientation.

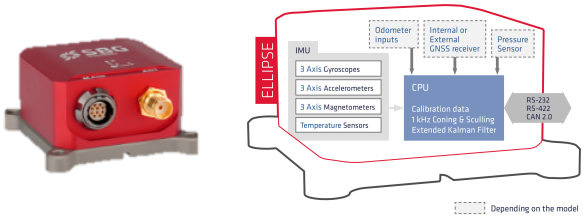

Figure 5: IMU SBG Ellipse-N

SBG Ellipse-N specifications:

- Integgrated GNSS
- Roll/Pitch accuracy: 0.1° (SP) / 0.05° (RTK)
- Yaw accuracy: 0.2°
- Velocity accuracy: 0.03 m/s
- GNSS Position accuracy: 1 cm
- Raw data accessible

3.4 Wheel and steering encoders

Wheel encoders are included in the AMK motor setup.

3.5 GSS

A custom GSS is in development.

3.6 Autonomous Control Unit

The Autonomous Control Unit (ACU) primarily focuses on managing the main computer vision sensors and executing a variety of algorithms essential for tasks such as Computer Vision, Simultaneous Localization and Mapping (SLAM), Path Planning, and Path Tracking.

Figure 6: ACU Nvidia AGX Orin Developer kit

ACU Nvidia AGX Orin Developer kit specifications:

- Arm Cortex-A78AE 12-core 64-bit 2.2 GHz
- Nvidia Ampere 2048 core with 64 Tensor core 1.3 GHz
- Enhanced deep learning capabilities
- Power consumption: 15 W to 60 W ($\simeq$ 30 W measured)

3.7 Electronic Control Unit

The ECU acts as a central hub for communication and control, ensuring that various subsystems operate cohesively to achieve autonomous functionality.

Figure 7: dSPACE Microautobox 3

dSPACE Microautobox 3 specifications:

- Real time architecture
- Operational frequency: 20 Hz
- Comprehensive automotive I/Os
- Matlab - Simulink environment with code generation
- CAN interface channels: 6

The ECU facilitates communication between the Autonomous Control Unit (ACU) and other boards, ensuring data exchange and coordination among various system components. It is responsible of low-level controls for four AMK electric motors, along with a brushless motor that manages steering. It is also responsible for implementing the vehicle's state machine, which governs the vehicle's operational states, and performing state estimation, which involves calculating the vehicle's current position and velocity.

4 Autonomous system

4.1 Overview

In Figure 8, the logical block diagram of the autonomous system is presented. LiDAR and camera data are processed to gather informations about the environment, which are then merged using a sensor fusion process. A SLAM (Simultaneous Localization And Mapping) algorithm is responsible for mapping the track in real time and localizing the vehicle, optimizing both the positions of the cones and the vehicle state.

Each time the map is updated, a path to follow is generated by the path planning process, which is subsequently refined by a trajectory optimizer. A velocity profiler determines the desired velocity to track at each point along the path. Finally, the path tracker computes the commands the vehicle should follow, including steering angle, and velocity reference or throttle input.

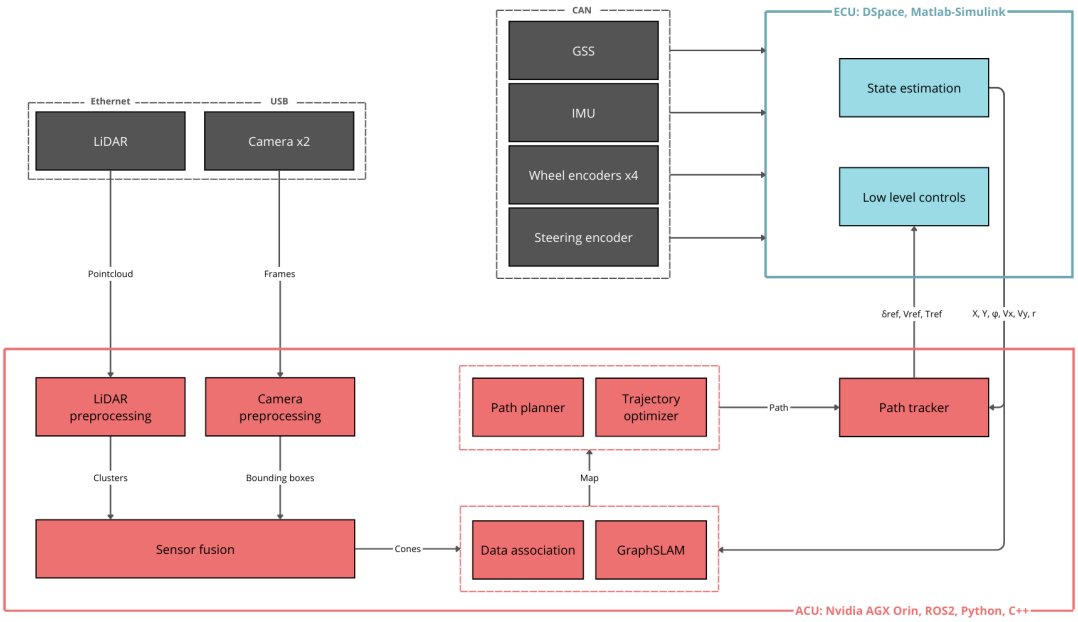

Figure 8: Autonomous system overview

4.2 Perception

The perception system's primary objective is to recognize and interpret the vehicle's environment. The track boundaries, in FSAE competitions, are defined by cones. Blue and yellow cones are used to mark the left and right boundaries of the track, respectively, while orange cones serve to indicate key details such as the starting line, stopping line, and other important markers along the track.

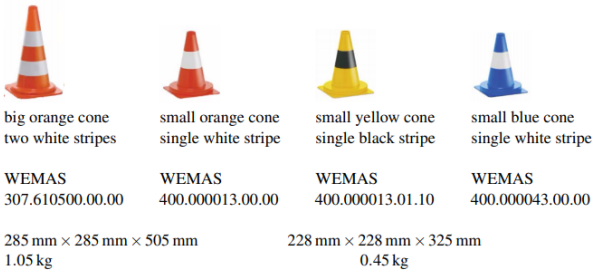

Figure 9: Cones rule specifications

Sensors data sources are evaluated together to extract key information about the cones, such as their position, shape, and color. The 3D point cloud offers precise spatial data, allowing the system to determine the cones exact locations in the environment, while the camera images provide the necessary color and shape details to distinguish between the different cone types. This combined analysis, after a sensor fusion step, enables the vehicle to accurately detect and classify the cones.

4.2.1 Sensors synchronization

LiDAR and cameras work in synchronization to provide data for the perception system. To ensure alignment between LiDAR and camera data, a trigger-based synchronization mechanism is employed. The LiDAR sensor serves as the trigger, since it is the slower sensor. This synchronization process guarantees that the LiDAR point cloud and the camera images are captured almost at the same time, allowing for accurate fusion of spatial and visual information.

Here is described the maximum time delay between LiDAR and cameras data, which is in the worst case the time needed to acquire an image:

$$\Delta t = \frac{1}{FPS_{camera}} = \frac{1}{35} \approx 28.57\,\text{ms}$$

4.2.2 LiDAR preprocessing

The produced LiDAR pointcloud consists of a dense set of individual points, each representing a specific location in 3D space.

Each point in a 3D point cloud contains several data attributes. Here's a breakdown of each:

- **x, y, z**: These represent the Cartesian coordinates of the point in 3D space. The x coordinate corresponds to forward distance, y to lateral distance, and z to height, relatively to the LiDAR frame.
- **Intensity**: This is a measure of how strongly the LiDAR pulse was reflected back to the sensor from the object. It provides insights into the material properties of the object.
- **Timestamp**: This is the timestamp indicating when the specific point was recorded.

- **Reflectivity**: This refers to the inherent ability of a surface to reflect LiDAR laser pulses. It is similar to intensity but can be more stable across different conditions, offering additional information about surface characteristics.
- **Ring**: The ring value specifies which laser beam within the LiDAR sensor captured the point, helping reconstruct the full 3D point cloud with better vertical resolution.
- **Ambient**: This value represents the ambient light level detected by the LiDAR sensor in the environment. It is often used to gauge lighting conditions.
- **Range**: This is the calculated distance between the LiDAR sensor and the object that reflected the laser pulse.

Geometrical filtering The point cloud is first passed through a filter to remove irrelevant sections of the point cloud. A filter is applied by removing points that exceed the height of the cones, as they do not contribute to cone detection. Points that belong to the vehicle's chassis, as these are part of the vehicle itself and not relevant to the surrounding environment, are removed.

Ground Plane Removal The ground plane removal algorithm is designed to efficiently filter out ground points from the point cloud. First, the point cloud is compacted by organizing the data based on distance and rotation angle, allowing for a more structured, two-dimensional representation. Each point is then described using its distance from the LiDAR sensor and its height, which helps in distinguishing ground points from non-ground objects. To achieve outlier-robust linear fitting, the RANSAC (Random Sample Consensus) algorithm is employed, which effectively models the ground plane while ignoring outliers such as objects or noise. A detailed explaination of RANSAC algorithm can be found in Path Planning section 4.4.1.

For enhanced efficiency, the algorithm is parallelized. The point cloud is divided into multiple sections, and the ground plane removal process is applied independently to each section using multithreading. This approach ensures faster processing times by utilizing multiple computational cores simultaneously, significantly speeding up the overall filtering process while maintaining accuracy.

After applying the ground plane removal process, it was empirically observed that approximately 98.3% of the points are filtered out by the geometrical filter and the ground plane removal.

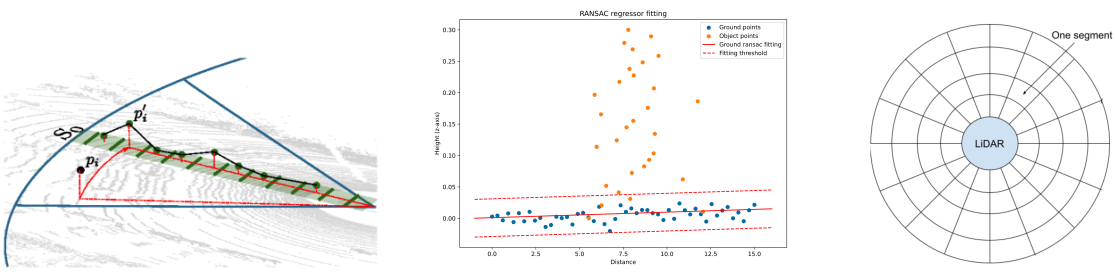

Figure 10: Ground Plane Removal

Clustering The DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm is employed for detecting clusters of points that represent cones within the filtered point cloud. This method identifies groups of closely packed points while effectively distinguishing them from noise and outliers. By analyzing the spatial distribution of the points, DBSCAN detects clusters based on their density, allowing it to reliably recognize the locations of cones. Once the clusters are identified, the algorithm calculates the centroid of each cluster, providing precise positional information for each detected cone. This centroid serves as a critical reference point for further processing and decision-making within the autonomous system.

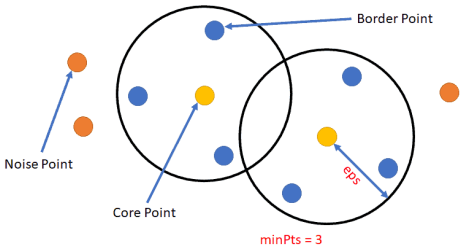

Figure 11: DBSCAN

Shape recognition The evaluation of clusters for cone shape recognition employs the known geometric characteristics of the cones to enhance detection accuracy. Each identified cluster is assessed by scoring its conformity to the expected cone shape, which involves calculating the distances of the cluster points to the theoretical cone surface. This scoring mechanism provides a quantitative assessment of how closely the points align with the ideal cone geometry.

Before verifying the geometrical similarity of a cluster to a cone, it is necessary to estimate the center of the cone relative to the centroid of the cluster under evaluation. Since a LiDAR only scans part of a cone's surface, the centroid of the point cloud cluster generated from this scan does not coincide with the actual center of the cone. To correct for this, the center of the geometric cone is approximated by adding a displacement vector to the cluster centroid in the direction of the LiDAR. This displacement is computed to be approximately 4.5 cm based on simulation results. The corrected cone center $\mathbf{C}_{cone}$ can be expressed as:

$$B = C_{cluster} + \mathbf{d}_{LiDAR}$$

where $C_{cluster}$ is the centroid of the point cloud cluster and $\mathbf{d}_{LiDAR}$ is the displacement vector along the LiDAR direction.

Figure 12, left image, shows in red the real cone and in blue LiDAR points on the surface and the estimated cone built approximating the cone center as explained above.

A cone is symmetrical around its direction vector $\mathbf{b}$. To analyze the distance from a specific point P to the cone's surface, we can define a plane using the cone's apex A and the point $B = A + \mathbf{b}$ (Figure 12). This transformation allows us to simplify the problem into two dimensions, where we know the shortest path must lie within this plane.

In this 2D representation, we can categorize the situation into three distinct areas marked in blue, red, and green. If point P is located in the blue region, the shortest

distance will be the length of v_1. For points in the red area, the distance corresponds to v_2, while points in the green area yield a distance of v_3.

It is important to note that our goal is to compute the distance to the cone's surface area, which, in this case, translates to measuring the distance to the line AC.

This leads us to the first step of the algorithm being to determine which case we are dealing with. We start by computing the coordinates of point C. To do this, we first need a direction vector $\mathbf{d}$ as:

$$\mathbf{d} = (P - A) - \frac{(P - A) \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}} \mathbf{b}$$

which will be perpendicular to $\mathbf{b}$. We then compute C as:

$$C = A + \mathbf{b} + \frac{\mathbf{d}}{\|\mathbf{d}\|} \tan \alpha \|\mathbf{b}\|$$

Next, we determine which case we are dealing with. If we are in the blue case, it follows that $(P - A) \cdot (C - A) < 0$. In a similar manner, if we are in the green case, the condition $(P - C) \cdot (A - C) < 0$ must hold true. If neither of these conditions is satisfied, we conclude that we are in the red case, and we can compute the distance as follows:

$$\|\mathbf{v_2}\| = \|(P - A) - \frac{(P - A) \cdot (C - A)}{(C - A) \cdot (C - A)} (C - A)\|$$

Clusters that fall below a specified score threshold are discarded, effectively filtering out those that do not represent the expected cone shape. This approach ensures that only the most reliable clusters are retained for further analysis, significantly improving the precision of cone identification and contributing to the robustness of the overall perception system.

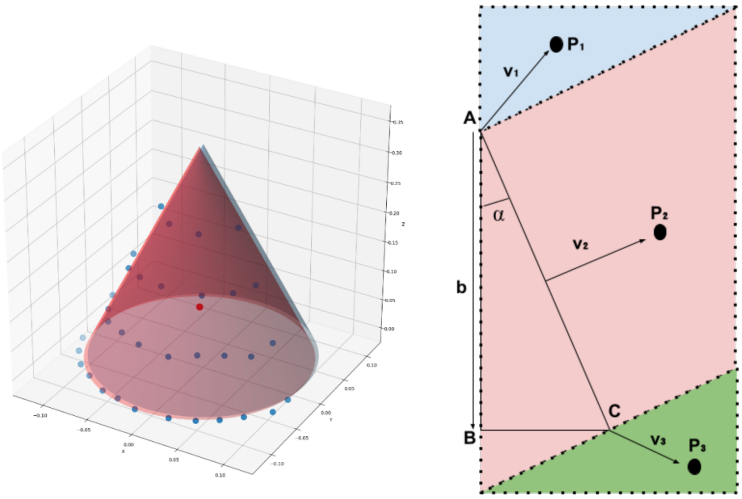

Figure 12: Shape Recognition

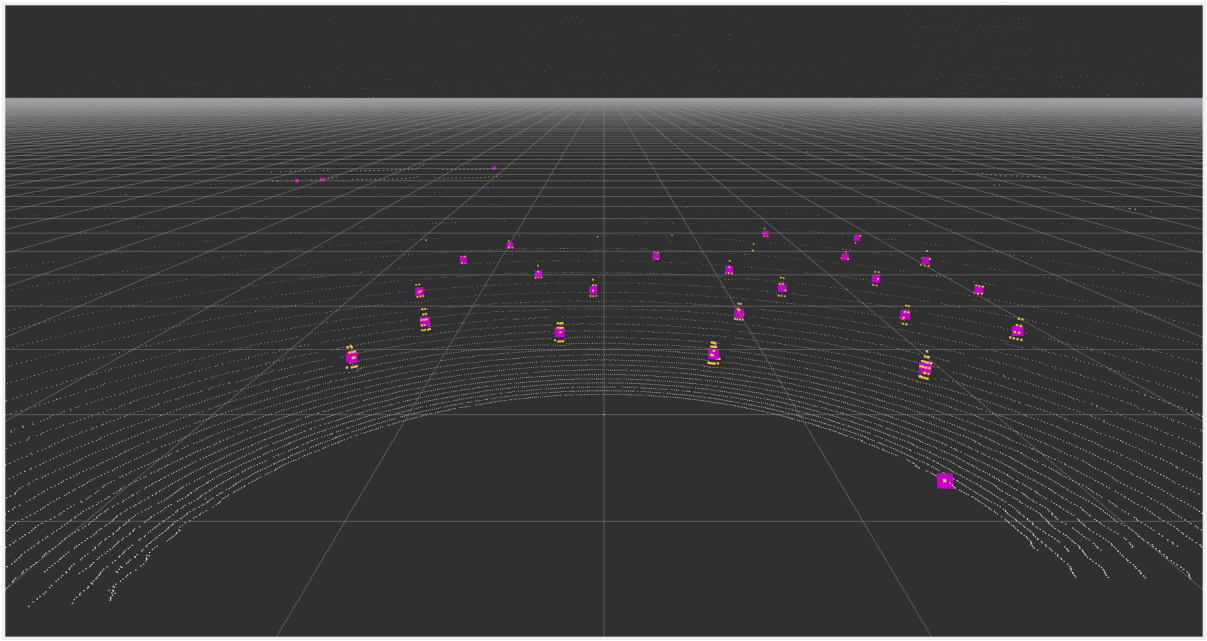

Figure 13: LiDAR preprocessing: point cloud in white, obstacle point cloud in yellow, centroids in violet

4.2.3 Camera preprocessing

YOLO The YOLO (You Only Look Once) algorithm is a cutting-edge real-time object detection system, designed to identify and localize objects within images. YOLO reframes object detection as a single regression problem, directly predicting bounding boxes and class probabilities in one forward pass through the neural network.

YOLOv8 utilizes anchor-free detection, improved feature pyramids, and a more efficient backbone to extract features and detect objects with greater speed and precision. In addition YOLOv10, which builds upon the YOLOv8 architecture, incorporates additional optimizations, such as RepVGG blocks for improved feature extraction and efficiency.

In our implementation, we specifically employ the YOLOv8 [6] and YOLOv10 [2] versions, developed by Ultralytics. For this task, YOLOv8 and YOLOv10 models have been trained using the FSCOCO dataset, which provides extensive labeled data on the four cone classes. Two parallel processes are employed, one for each camera feed.

To evaluate their performance, we benchmarked nano (n), small (s) and medium (m) versions of YOLOv8 and YOLOv10 using standard metrics, including mean Average Precision (mAP), inference latency and peak GPU memory usage. Results of which are presented in the table below:

Model	Version	mAP50)	mAP50-95)	Latency (ms)	GPU Mem (MB)
YOLOv8	n	0.55	0.36	7.6	64.4
YOLOv8	s	0.53	0.34	8.2	110.7
YOLOv8	m	0.57	0.37	12.7	196.7
YOLOv10	n	0.57	0.37	9.5	64.9
YOLOv10	s	0.53	0.33	9.2	99.1
YOLOv10	m	0.60	0.40	12.7	157.6

Table 1: Benchmark Results for YOLOv8 and YOLOv10 on FSCOCO Dataset

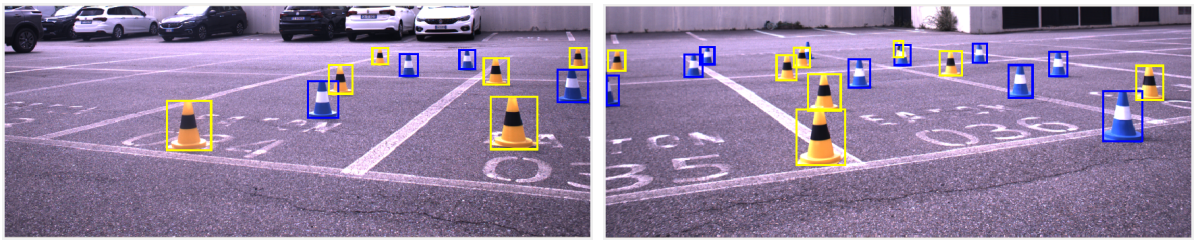

Figure 14: YOLO

4.2.4 Sensor Fusion

The primary objective of sensor fusion is to integrate data from LiDAR and camera sensors to achieve a more accurate understanding of track information. LiDAR preprocessing provides precise information regarding the position and shape of cones, allowing for accurate spatial representation. Cameras preprocessing delivers detailed insights into the shape and classification of the cones. By combining these complementary data sources, sensor fusion gives reliability and accuracy in detecting track cones.

The sensor fusion operation is performed once for each camera and then results are merged to avoid duplicates.

Points 2D Mapping Mapping LiDAR points onto images mapping is achieved using two key matrices: the extrinsic matrix and the intrinsic matrix. The extrinsic matrix $\mathbf{E}$ is defined as:

$$\mathbf{E} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{bmatrix}$$

where $\mathbf{R}$ represents the rotation matrix that aligns the LiDAR coordinate system with the camera coordinate system, and $\mathbf{t}$ is the translation vector. This matrix roto-translates the LiDAR points $\mathbf{P}_{LiDAR}$ into the camera's coordinate system as follows:

$$\mathbf{P}_{camera} = \mathbf{E} \cdot \mathbf{P}_{LiDAR}$$

Following this, the intrinsic matrix $\mathbf{K}$ is used to map the transformed points onto the 2D image plane. Specifically, the intrinsic matrix serves as an approximation of

the camera's spatial model, incorporating both the intrinsic parameters and the camera distortion characteristics. The intrinsic matrix is defined as:

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where f_x and f_y are the focal lengths, and (c_x, c_y) is the principal point. The final mapping of the points onto the image plane is given by:

$$\mathbf{P}_{image} = \mathbf{K} \cdot \mathbf{P}_{camera}$$

In this equation, $\mathbf{P}_{image}$ represents the coordinates of the projected points in the image plane, while $\mathbf{P}_{camera}$ denotes the coordinates of the points in the camera's 3D space. Here is the extension of the formula provided above:

$$x_i = f_x \frac{x_c}{z_c} + c_x \quad , \quad y_i = f_y \frac{y_c}{z_c} + c_y$$

Extrinsic and intrinsic matrices can be found with automatic tool like Matlab LiDAR Camera Calibrator tool using a checkerboard and acquiring sensors data. A manual calibration can be easily done computing analytically the extrinsic matrix and adjusting intrinsic parameters.

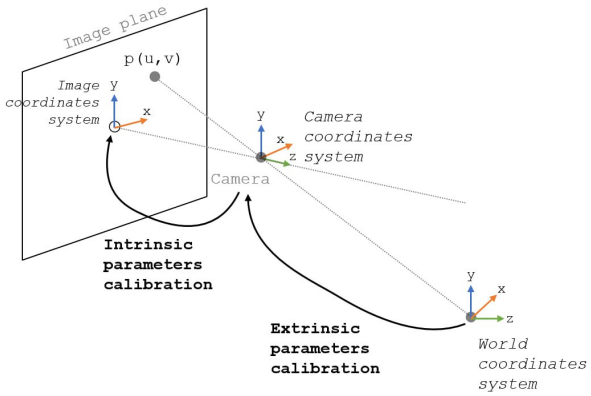

Figure 15: Extrinsic and intrinsic matrices

Point to bounding box association The association between bounding boxes and mapped points is established through two scoring functions designed to evaluate the proximity and relevance of each mapping. The first score function calculates the distance between the center of the bounding box and the mapped point in the image, defined as:

$$S_1 = \sqrt{(x_{bb} - x_p)^2 + (y_{bb} - y_p)^2}$$

where (x_{bb}, y_{bb}) are the coordinates of the bounding box center, and (x_p, y_p) are the coordinates of the mapped point.

The second score function assesses the bounding box height as a function of distance from the camera, ensuring that the height of the bounding box is appropriately scaled based on the distance to the object. This is expressed as:

$$S_2 = f(h_{bb}, d) = (h_{bb} - g(d))^2$$

where h_{bb} is the height of the bounding box and $g(d)$ is a non-linear function approximation that models the expected height with distance, derived from polynomial fitting.

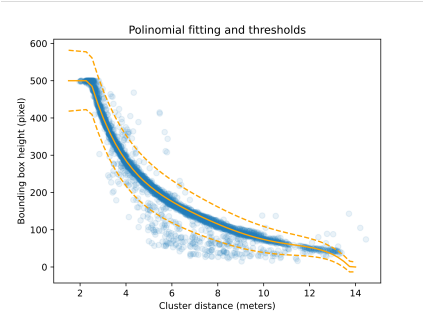

Figure 16: Polinomial fitting of distance to bounding box height function

Combining these scoring functions, a robust association between bounding boxes and mapped points can be achieved.

Merge Once the sensor fusion step is executed for both cameras, acquired informations are merged in order to avoid duplicates.

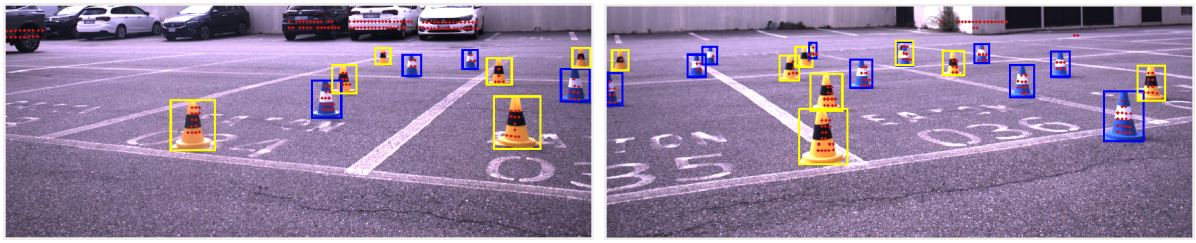

Figure 17: Sensor Fusion obstacle point cloud mapping onto processed images

4.2.5 Results

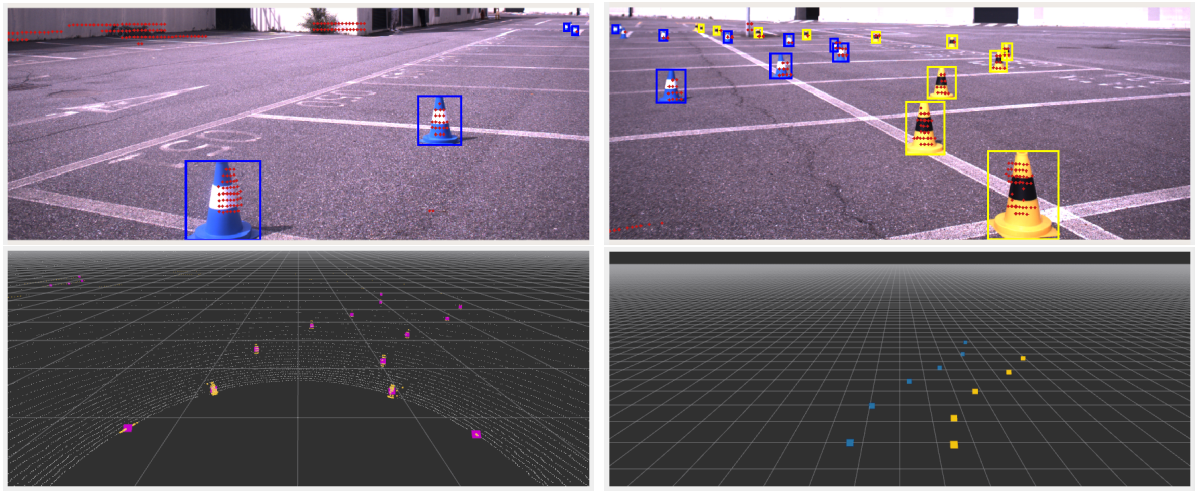

Figure 18: Perception track results

4.3 Simultaneous Localization And Mapping

In robotics, Simultaneous Localization and Mapping (SLAM) is a class of algorithms designed to allow a robot, such as an autonomous vehicle, to simultaneously estimate its position within an environment and construct a map of its surroundings. This environment is often characterized by a set of fixed features, known as landmarks, which serve as reference points. In this case, on a racetrack, landmarks include cones placed at fixed locations as track boundaries.

Formally, the SLAM problem can be stated as follows: given a sequence of sensor measurements and control inputs, SLAM seeks to estimate and optimize both the robot's trajectory over time and the spatial positions of landmarks. This estimation is refined iteratively as new data becomes available, enabling a dynamic and accurate model of the environment.

The GraphSLAM algorithm presented below is implemented using the g2o library [7], taking as reference also [5] and [3].

4.3.1 GraphSLAM

Graph-based SLAM has been chosen as the approach to formulate the SLAM problem. This method is characterized by the construction of a graph in which nodes represent the robot's poses and landmarks, while edges denote the constraints between pairs of nodes. The primary objective is to identify a node configuration that optimally satisfies all constraints.

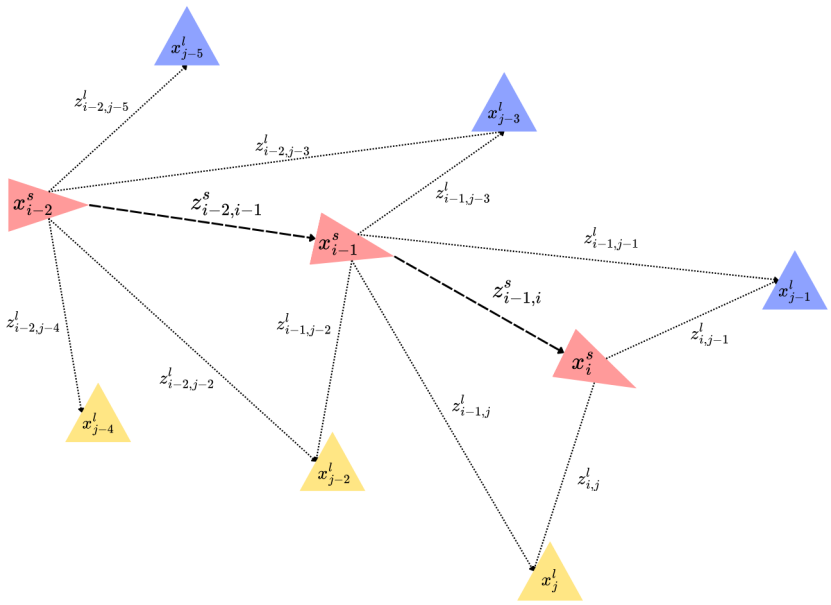

Figure 19: GraphSLAM graph representation

Graph Construction The graph construction is performed as follows:

- For each odometry acquisition $\mathbf{z}_{i-1,i}$, a new node corresponding to the pose x^s_i is added to the graph. An edge representing the control input $\mathbf{z}_{i-1,i}$ is then inserted between the nodes x^s_{i-1} and x^s_i.

- For each landmark measurement $\mathbf{z}_{i,j}$, a data association algorithm checks for a correspondence with existing nodes:
  - If a match is found, a constraint between the corresponding nodes is created based on the measurement $\mathbf{z}_{i,j}$.
  - If no match is detected, a new node representing a landmark x_j^l is added to the graph.

Graph Optimization The optimization of the graph is performed by minimizing the overall error in the node configuration.

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \sum_k \mathbf{e}_k^T \Omega_k \mathbf{e}_k \tag{1}$$

where the sum is taken over all edges. Here, $\mathbf{e}_k$ represents the error term for each constraint, defined as:

$$\mathbf{e}_k = \mathbf{e}_{i,j} = \mathbf{z}_{i,j}^n - h^n(\hat{x}_i, \hat{x}_j) \tag{2}$$

In this expression:

- $\mathbf{z}_{i,j}^n$ is the observed measurement of the node x_j as seen from node x_i.
- $h^n(\hat{x}_i, \hat{x}_j)$ is the expected measurement based on the graph configuration.

The notation $n \in \{l, p\}$ indicates whether the measurement corresponds to a landmark ($\mathbf{z}_{i,j}^l$) or a pose ($\mathbf{z}_{i,j}^s$). Finally, $\Omega_k = \Omega_{i,j}^n$ represents the information matrix of the measurement, which is adjusted accordingly.

Localization-Only Mode The same algorithm can also operate in a localization-only mode interrupting the optimization of nodes corresponding to landmarks and performing the optimization only on vehicle pose nodes.

Alternatively, an a priori known environment map can be provided to the graph (e.g. skidpad).

Implementation Details The graph structure and optimization back-end are implemented using the g2o library [7], a C++ library that provides high computational efficiency. The graph is updated with each new acquisition, while the optimization process is tuned for real-time performance.

To maintain efficiency, optimization is performed every 10 acquisitions, focusing only on the last N poses, ensuring real-time operation.

4.3.2 Data Association

Data association is a critical component in solving the SLAM problem, as the optimization of the robot's pose depends heavily on the recognition of repeated observations of the same landmarks. Since all cones in the environment appear visually identical, traditional feature extraction methods cannot reliably distinguish individual landmarks.

K-nearest neighbour A non-Bayesian approach to the problem is the k-nearest neighbors (kNN) method which offers a simpler alternative. However, it is sensitive to noise and may struggle if landmarks are not well separated.

Joint Compatibility Branch and Bound (JCBB) The best results in data association have been achieved with the Joint Compatibility Branch and Bound (JCBB) algorithm. JCBB use a Byesian approach that works by recursively assigning each new data point to a previously recognized point. Compatibility pruning within JCBB is conducted based on the Mahalanobis distance, which provides two types of compatibility:

- **Individual Compatibility**: The Mahalanobis distance between a new point and its assigned point.
- **Joint Compatibility**: The overall compatibility between all assigned points as a set.

To enhance performance, a modified version of the JCBB algorithm [11] has been implemented. This version reduces the computational complexity of joint compatibility checks from $O(n^2)$ to $O(1)$, making it more suitable for real-time applications.

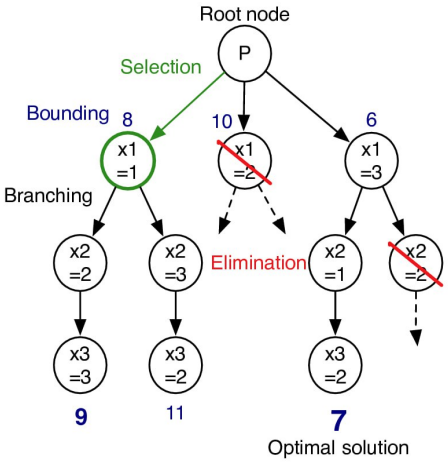

Figure 20: JCBB recursive steps

4.3.3 Lap Counter

The lap counter is integrated within the SLAM process to monitor the number of laps completed by the vehicle. The mechanism operates as follows:

- When the vehicle approaches the map's origin (the starting point) within a specified radius r_{in}, the lap counter is triggered.
- The counter is incremented once the vehicle exits the origin region, defined by an outer radius r_{out}.

4.3.4 Results

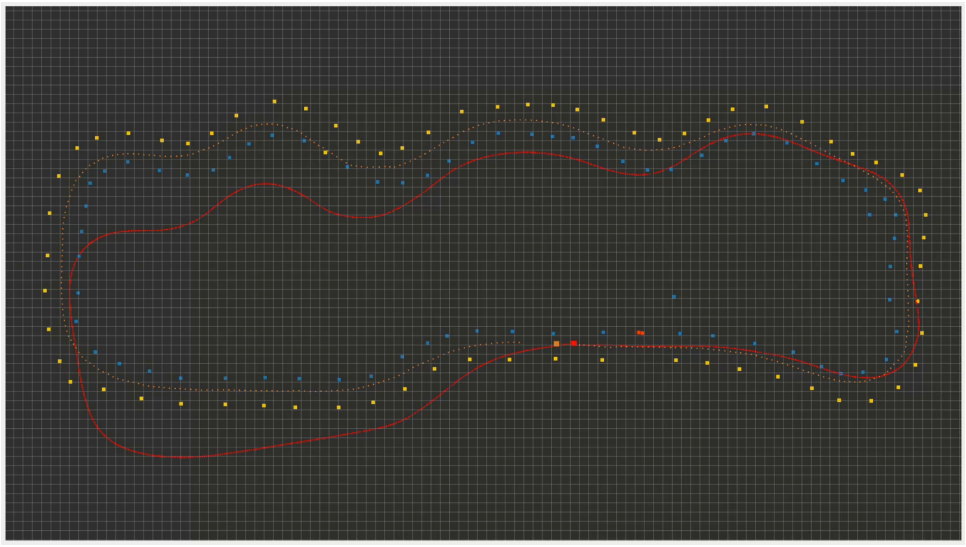

Figure 21: GraphSLAM track test (in red state estimated, in orange optimized state)

4.4 Path Planning

The main purpose of path planning solutions is to define an optimal path for a vehicle to follow while avoiding track obstacles.

Actual path planner implementations employ a Delaunay triangulation to find middle points of the track.

Delaunay Triangulation Delaunay triangulation is a way of connecting a set of points in a plane to form triangles, such that no point is inside the circumcircle (the circle passing through all three vertices) of any triangle in the triangulation. The Delaunay triangulation is closely related to the Voronoi diagram, a geometric structure that divides a plane into regions based on the closest proximity to a set of points.

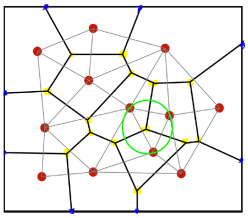

Figure 22: Delaunay triangulation and Voronoi diagram

The track environment, composed by cones as 2D positions in the plane, is discretized using Delaunay triangulation, which helps in identifying the waypoints as the middle points of triangles edges having vertices of different color.

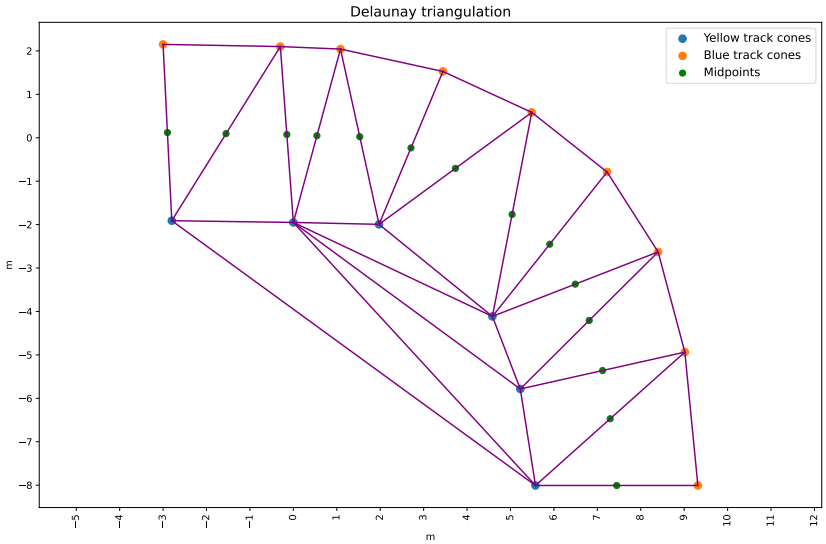

Figure 23: Delaunay waypoints

4.4.1 RANSAC Planning

Delaunay Triangulation A Delaunay triangulation between detected cones is employed to find center waypoints of the path, ensuring that the cones are sufficiently close by comparing their Euclidean distance, formulated as:

$$d(P_1, P_2) = \|P_1 - P_2\| < d_{\text{threshold}}$$

RANSAC algorithm RANSAC (Random Sample COnsensus) algorithm is an iterative method used to estimate a model from a dataset that contains both inliers and outliers, assuming that there could be some outliers waypoints, for instance due to wrong perception acquiring. The RANSAC algorithm, for each iteration, fits a line to a random pair of midpoints, checking how many points lie close to the line using the perpendicular distance from a point to the line, given by:

$$d(P, l) = \frac{|y - mx_P - b|}{\sqrt{m^2 + 1}}$$

where:

$$l : y = mx + b$$

After the classification of of midpoints as inliers or outliers, the current best solution is updated if the number of inliers is greather than the previous best solution.

Finally, the best-fit line is refined using linear regression, defining a line in the form:

$$y = \beta_0 + \beta_1 x + \epsilon,$$

where ϵ is an error term.

The goal of linear regression is to find the best-fitting line that minimizes the sum of the squared differences between the observed values y_i and the predicted values $\hat{y}_i$. This method, known as least squares regression, minimizes the cost function:

$$\text{Cost}(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} \left(y_i - (\beta_0 + \beta_1 x_i)\right)^2 .$$

In matrix form, for n data points:

$$\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

where:

- $\mathbf{y}$ is an $n \times 1$ vector of observed values,
- X is an $n \times 2$ matrix of input data, with the first column being all ones (for the intercept) and the second column containing the x-values,
- $\boldsymbol{\beta}$ is a 2×1 vector containing β_0 and β_1,
- $\boldsymbol{\epsilon}$ is an $n \times 1$ vector of errors.

The least squares solution, which provides the values of β_0 and β_1 that minimize the sum of squared errors, is given by:

$$\boldsymbol{\beta} = (X^T X)^{-1} X^T \mathbf{y},$$

Line Fitting The path is finally fitted with a defined fitting step. A corresponding Δx to the fitting step is computed as follows:

$$\Delta x = \text{step} \times \cos\left(\arctan(m)\right)$$

Computing (x_i, y_i) value pairs according to Δx solution, a fitted path is found.

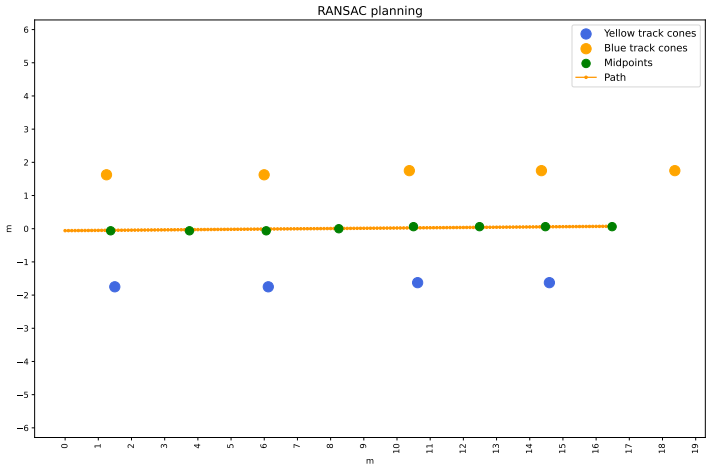

Figure 24: RANSAC path

4.4.2 Skidpad Path Solution

In the skidpad event, the path that the vehicle must follow is precomputed in advance. This precomputed path defines the optimal trajectory for the vehicle to navigate through figure-eight pattern on the track.

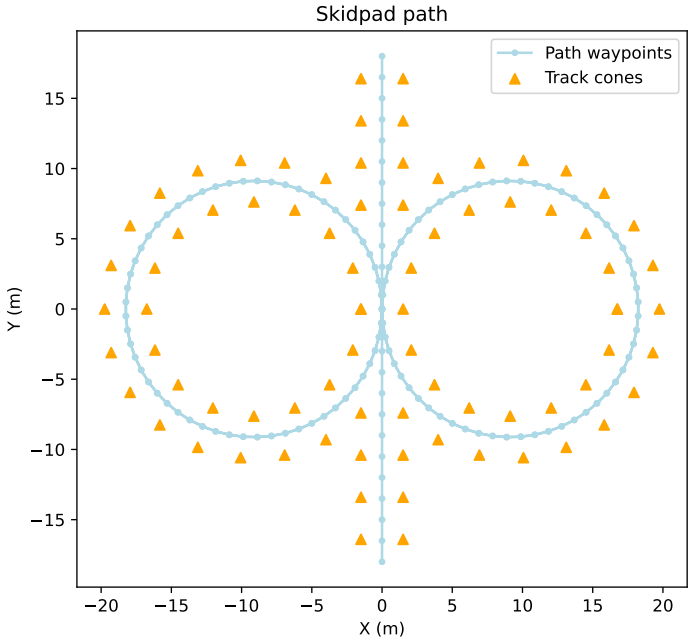

Figure 25: Skidpad path

4.4.3 Local Path Planning

Local path planning algorithms are first implementations of path planners emplyed to compute the path to follow in Autocross and Trackdrive events. These algorithms compute path in local coordinates, and it is computed with the cone informations in front of the vehicle every time the vehicle state changes.

RRT Planner and Discrete Tree Planner implementations differs on the way is defined the paths tree starting by the vehicle position, sharing the heuristic and the score function used to evaluate the paths and chose with an exhaustive search the best path.

RRT Planner tree generation The Rapidly-exploring Random Tree (RRT) is a path planning algorithm widely used in robotics and autonomous systems, particularly for high-dimensional and complex environments. The algorithm begins by initializing a tree rooted at the vehicle position, the zero in local coordinates. At each iteration, a random sample is taken from the configuration space. The algorithm then finds the nearest node in the existing tree to this random sample and attempts to extend the tree in that direction by a specified step size, generating a new node. If the path from the nearest node to the new node is collision-free, the new node is added to the tree. The process of random sampling, extending, and collision checking continues until a maximum number of iterations is completed.

Discrete Tree Planner tree generation Generating an RRT every time a path needs to be computed is time consuming and as an improvement the Discrete Tree Planner defines once a fixed tree as a pattern in which search algorithms are performed.

Starting by the origin, the tree is constructed recursively by expanding nodes at specified step sizes and angular increments from parent nodes. Recursively each parent node n_i is expanded with children nodes n_{i+1}. For each angle within a fixed range $(-\theta_{\max}, +\theta_{\max})$ and an angle step $\Delta\theta$ a new node is generated with a position calculated by applying the current step size in the direction determined by $\theta + \Delta\theta$.

Once created, each node is stored within a map, that groups nodes by their depth, facilitating easy lookup during path evaluation. In order to facilitate the path search leaf nodes which distance from nearest midpoint is under a certain threshold are then filtered out.

Exhaustive search An exhaustive search is employed to evaluate candidate paths. In order to evaluate the path a Delaunay triangulation is performed to midpoints, then used as a contribution in the score function. The heuristic used to choose the best path evaluates candidate paths by assigning each a cumulative score based on the proximity to midpoints, proximity to obstacles, and angular smoothness between nodes. For each path, a backward traversal is performed from each leaf node to the start node, calculating the path's score. The total score for a path is obtained by summing up individual node scores along the path, calculated as follows.

For each node n in a path, the node score S_n is computed using the following criteria:

- **Proximity to midpoints**: For each midpoint m_i, the distance d_{m_i} between n and m_i is computed. If d_{m_i} is within a specified threshold D_{mid}, the node receives a positive score proportional to the defined distance, normalized to D_{mid}:

$$S_{\text{midpoint}} = \sum_i \left(\frac{D_{\text{mid}} - d_{m_i}}{D_{\text{mid}}} \cdot k_{m_i} \right), \quad \text{if } d_{m_i} < D_{\text{mid}}$$

  where k_{m_i} is a factor between 0 and 1 defining the importance of the midpoint, higher if it's generated by a couple of cones of different color, lower if one of them hasn't a defined color.

- **Proximity to Obstacles (Cones)**: The distance d_{c_j} from node n to each obstacle c_j is similarly calculated. If d_{c_j} is within a specified threshold D_{cone}, a negative score, normalized to D_{cone} is applied, penalizing proximity to obstacles:

$$S_{\text{cone}} = -\sum_j \left(\frac{D_{\text{cone}} - d_{c_j}}{D_{\text{cone}}} \right), \quad \text{if } d_{c_j} < D_{\text{cone}}$$

- **Smoothness (Angular Change)**: A smoothness penalty is applied based on the change in angle $\Delta\theta$ between each node and its parent node. This change in orientation penalizes sharp turns, calculated as:

$$S_{\text{theta}} = -\Delta\theta$$

The total node score S_n is computed as:

$$S_n = S_{\text{midpoint}} \cdot k_{\text{midpoint}} + S_{\text{cone}} \cdot k_{\text{cone}} + S_{\text{theta}} \cdot k_{\text{theta}}$$

The path score S_{path} is then the sum of all node scores S_n along the path:

$$S_{\text{path}} = \sum_n S_n$$

The path with the highest S_{path} is selected as the best path, balancing proximity to midpoints, avoidance of obstacles, and smoothness.

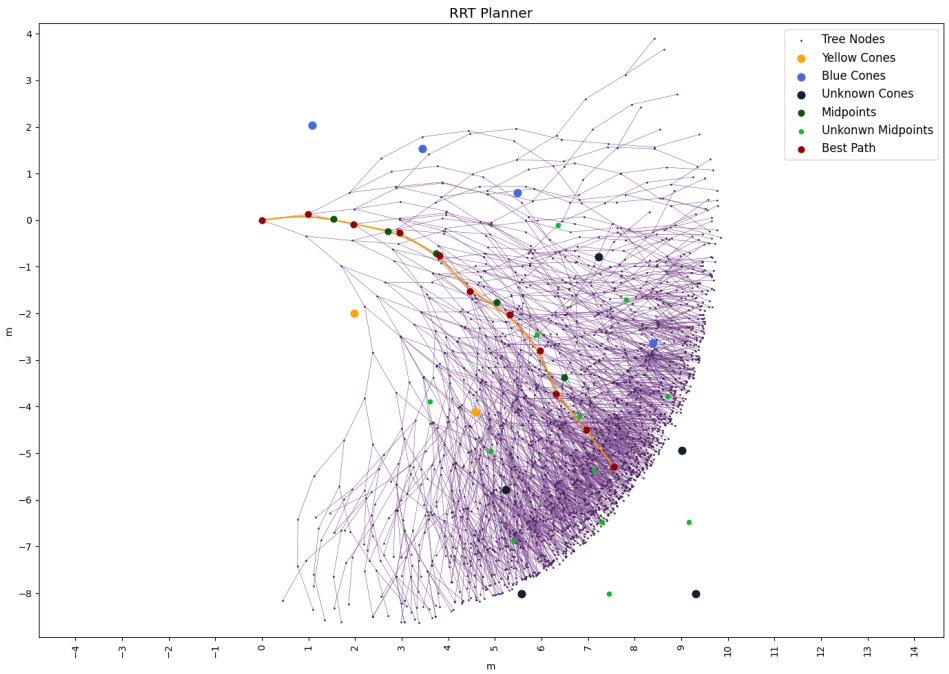

Figure 26: RRT Planner

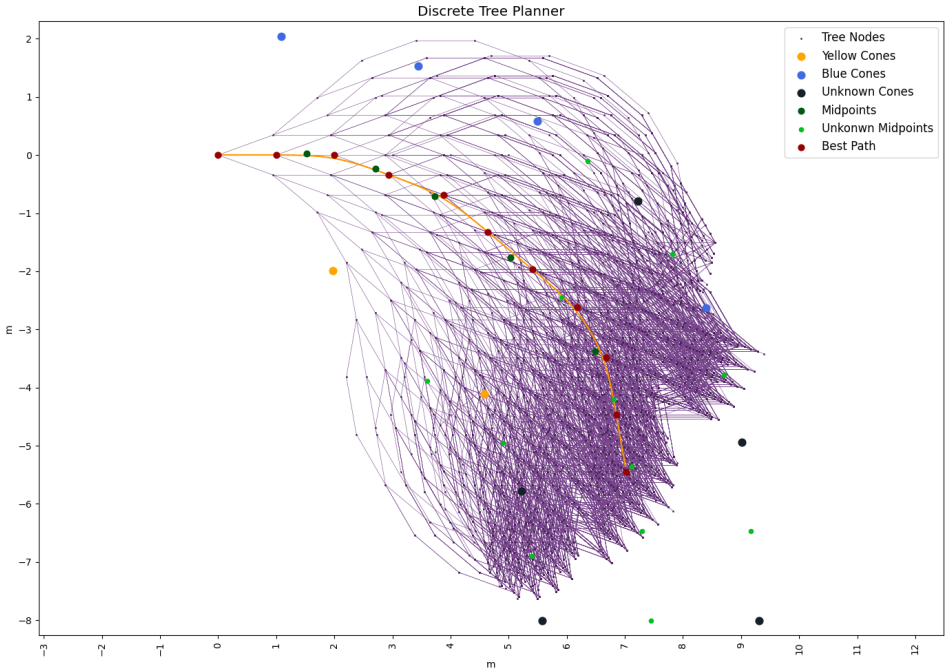

Figure 27: Discrete Tree Planner

4.4.4 Global Path Planning

The Global Path Planning algorithm is employed to compute the path to follow in Autocross and Trackdrive events, where the track is unknown.

The algorithm computes the optimal trajectory based on the cone map and the current vehicle state. The primary goal is to generate a central trajectory that follows the midpoints between the cones, ensuring the midpoints are aligned with the track's direction. To achieve this, the algorithm employs a cone sorting mechanism based on Delaunay triangulation.

Delaunay Triangulation A Delaunay triangulation between detected cones is employed to find center waypoints of the path, ensuring that the cones are sufficiently close by comparing their Euclidean distance, formulated as:

$$d(P_1, P_2) = \|P_1 - P_2\| < d_{\text{threshold}}$$

Path Starting Points Selection In order to perform a path search algorithm, two starting waypoints need to be defined.

The algorithm begins by selecting the closest waypoint behind the vehicle as the first starting point. To achieve this, all waypoints coordinates $(x_{\text{global}}, y_{\text{global}})$ are transformed into the vehicle's local frame applying the transformation explained below:

$$P_{\text{local}} = \mathbf{R}^T(P_{\text{global}} - \mathbf{t})$$

where:

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \quad \mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Having as a result the subsequent extended formula:

$$\begin{bmatrix} x_{\text{local}} \\ y_{\text{local}} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \left(\begin{bmatrix} x_{\text{global}} \\ y_{\text{global}} \end{bmatrix} - \begin{bmatrix} t_x \\ t_y \end{bmatrix} \right)$$

The search criterion is based on the absolute value of the $\text{atan2}(y_{\text{local}}, x_{\text{local}})$ function, ensuring the cone lies within a specified angular limit behind the vehicle.

The second starting point is chosen as the closest cone in front of the vehicle, following the same formula for calculating $\text{atan2}(y_{\text{local}}, x_{\text{local}})$, but with a different angular limit to ensure the cone is ahead of the vehicle.

Path Search The search algorithm begins by forming an initial vector between the first and second starting waypoints. In each iteration, the algorithm searches from the last sorted midpoint in the direction of the last vector, within a semi-circle in front of the vehicle. The last vector is computed as:

$$\mathbf{v}_{\text{last}} = P_i - P_{i-1}$$

The algorithm then finds the closest unsorted midpoint in this direction and verifies it against the following constraints:

- **Consecutive Midpoint Maximum Distance**: The distance between two consecutive midpoints d_{mid} is computed as:

  $$d_{\text{mid}} = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

  This distance must not exceed a predefined threshold to ensure smooth path planning.

- **Consecutive Midpoint Maximum Angle Deviation**: The angle deviation between consecutive vectors is determined by first calculating the dot product of two vectors, followed by the angle deviation θ:

  $$\cos\theta = \frac{\mathbf{v}_{i-1} \cdot \mathbf{v}_i}{|\mathbf{v}_{i-1}||\mathbf{v}_i|}$$

  The angle deviation θ is then obtained as:

  $$\theta = \arccos(\cos\theta)$$

  The deviation must remain within a defined limit to ensure trajectory continuity.

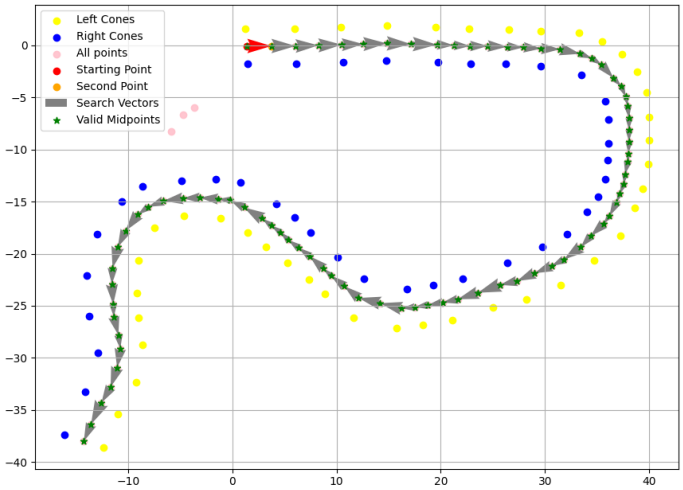

Figure 28: Path searching

Sorting Cones After sorting the midpoints, the algorithm generates sorted arrays of blue and yellow cones for further trajectory computation.

Unknown Cones Placement The algorithm also handles the placement of unknown cones.

The input unknown cones are first filtered to ensure that each unknown cone is placed within a threshold distance from at least one of the already sorted cones (blue or yellow).

After filtering, a recursive approach is used to compute the total angle deviation in the cone sequence by inserting the unknown cone in various positions within the sorted blue or yellow cone arrays.

The total angle deviation is computed using the following formula:

$$\text{total\_deviation} = \sum_{i=1}^{n-1} \theta_i$$

where θ_i is the angle deviation at the i-th cone in the cone sequence, computed as:

$$\cos(\theta_i) = \frac{\mathbf{v_1} \cdot \mathbf{v_2}}{|\mathbf{v_1}||\mathbf{v_2}|}$$

Here, $\mathbf{v_1}$ and $\mathbf{v_2}$ are vectors formed by the points P_1, P_2, and P_3 (the cones):

$$\mathbf{v_1} = P_2 - P_1$$

$$\mathbf{v_2} = P_3 - P_2$$

Finally, the angle deviation is calculated as:

$$\theta_i = \arccos\left(\text{clip}(\cos(\theta_i), -1.0, 1.0)\right)$$

where the clipping function ensures that the cosine value remains within the valid range for the *arccosine* function.

The algorithm tries various configurations of placing the unknown cones into the blue or yellow cone arrays. The configuration that results in the minimum total angle

deviation is selected, resulting in updated sorted arrays of blue and yellow cones, including the unknown cones.

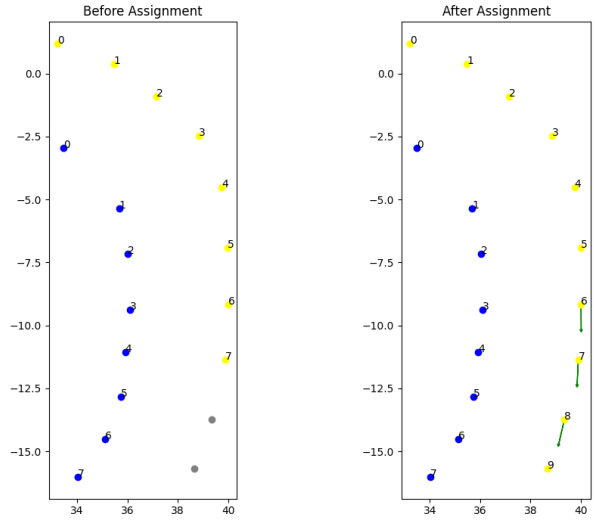

Figure 29: Unknown cones color assignment

Path Definition After sorting the cones, including the placement of unknown cones, Delaunay triangulation is performed again on the sorted arrays of blue and yellow cones and waypoints are computed again. This step ensures that the midpoints between the cones are accurately calculated based on the current cone configuration.

The found sorted midpoints represent the central trajectory of the track.

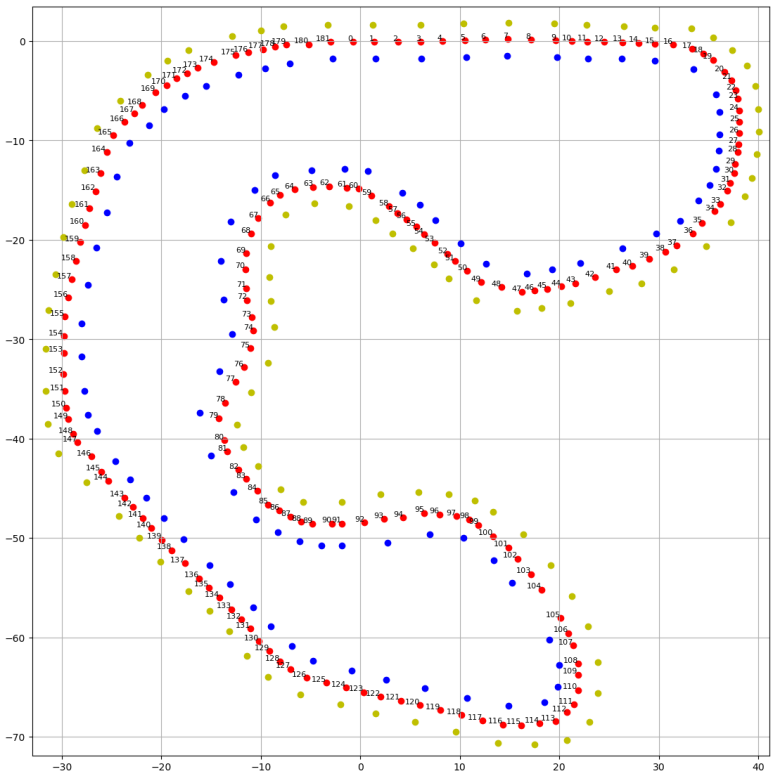

Figure 30: Final path

4.4.5 Trajectory Optimization

The trajectory optimization aims to calculate the best path to follow on the track in order to minimize lap time. Solving this task by directly minimizing lap time is often impractical due to its complexity. Instead, the minimum curvature heuristic is commonly used, as it is faster and produces effective results. The principle is to find the path that minimizes the car's overall steering angle. The resulting path is commonly known as the racing line.

This approach is adapted from the method outlined in [1] and customized to account for specific vehicle and problem constraints.

Algorithm Overview The algorithm requires an array of sorted midpoints, each with associated left and right cones. The main idea is to calculate a shift for each midpoint, either to the right or left, to create the racing line as a spline that interpolates these shifted midpoints. In tight curves, additional constraints on boundaries are applied by introducing virtual midpoints, based on track curvature. As curvature increases, more virtual midpoints are added to better respect these constraints.

Curvature evaluation To generate virtual midpoints, spline interpolation is first applied to real midpoints, and n equally spaced points are taken along this spline curve.

Before determining which virtual midpoints to retain, track curvature is evaluated.

First, tangent vectors to the spline interpolating the midpoints are computed. The tangent vector at each midpoint is approximated using finite differences:

$$\mathbf{T}_i = \begin{bmatrix} \Delta x_i \\ \Delta y_i \end{bmatrix},$$

where

$$\Delta x_i = x_i - x_{i-1}, \quad \Delta y_i = y_i - y_{i-1},$$

with (x_i, y_i) representing the smoothed coordinates.

Normalization is then applied to obtain the unit tangent vector:

$$\hat{\mathbf{T}}_i = \frac{\mathbf{T}_i}{\|\mathbf{T}_i\|}.$$

To measure curvature, the vector changes along the curve are analyzed. The change in the components of the tangent vector is given by:

$$\Delta \hat{T}_{x,i} = \hat{T}_{x,i} - \hat{T}_{x,i-1}, \quad \Delta \hat{T}_{y,i} = \hat{T}_{y,i} - \hat{T}_{y,i-1},$$

where $\hat{T}_{x,i}$ and $\hat{T}_{y,i}$ are the components of the unit tangent vector.

The differential arc length ds between consecutive points is calculated as:

$$ds_i = \sqrt{\Delta x_i^2 + \Delta y_i^2}.$$

The curvature k_i at each point is defined as the rate of change of the unit tangent vector with respect to arc length:

$$k_i = \sqrt{\left(\frac{\Delta \hat{T}_{x,i}}{ds_i}\right)^2 + \left(\frac{\Delta \hat{T}_{y,i}}{ds_i}\right)^2}.$$

To smoothen the curvature values, a moving average is applied. The curvature array k is extended at both ends, and the smoothed curvature k_{smoothed} is computed as:

$$k_{\text{smoothed}} = \frac{1}{N} \sum_{j=-4}^{4} k_{i+j},$$

where $N = 9$ is the window size. The smoothed array is then trimmed to match the original length of k.

Virtual midpoints selection Virtual midpoints are selected based on the curvature coefficients vector k obtained. By default, one midpoints in every three midpoints are retained to ensure coverage across the track. Additionally, midpoints with $k_i \geq 0.13$ are retained to maintain an adequate number of virtual midpoints close to the number of original midpoints.

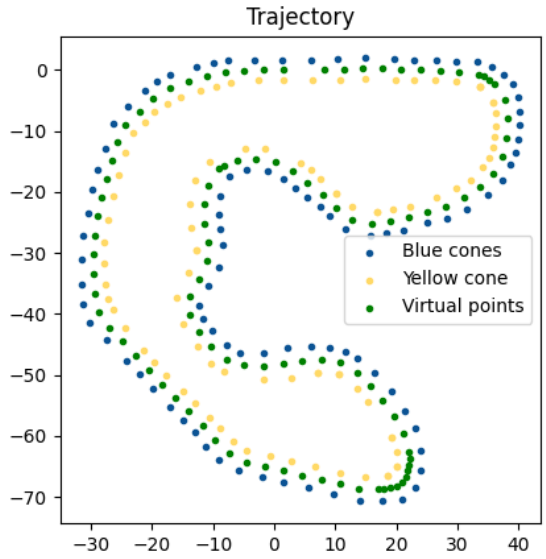

Figure 31: Virtual Points

Associating Cones to Virtual Midpoints Virtual midpoints do not initially have associated cones, unlike the real midpoints. To establish boundaries for each virtual midpoint, the nearest blue and yellow cones are assigned. Although these cones may not perfectly align with the virtual midpoint, the error remains minimal due to the number of virtual points generated. A margin error is applied to mitigate alignment discrepancies.

Determining Left and Right Bounds The final step before implementing the quadratic programming algorithm is to calculate the left and right distances from each midpoint to its assigned cones. To account for potential errors in cone mapping and alignment, a margin error is subtracted from the actual Euclidean distance, ensuring that the midpoint's maximum shift remains within a safe margin.

Quadratic Programming Algorithm With all necessary bounds defined, the minimum-curvature solution is determined using the quadratic programming algorithm described in [1]. In this context, QP is formulated to minimize the total curvature along the planned path. This optimization problem involves both the minimization objective and constraints to account for the physical limitations of the vehicle's steering, as well as the track boundaries.

The optimization problem is defined over a set of points r_i along the center line of the track, which are simply the virtual midpoints discussed above. Each point r_i can be adjusted along its normal vector $\mathbf{n}_i$ by a scalar parameter α_i, which allows the trajectory to shift within track boundaries. The normal vector is normal with respect to the interpolating spline that fits the set of midpoints r_i.

$$\vec{r}_i = \vec{p}_i + \alpha_i \vec{n}_i$$

The set of parameters $\alpha = [\alpha_1, \ldots, \alpha_N]^\top$ is what the QP formulation seeks to solve, minimizing the sum of squared curvatures along the path.

The minimization objective is given by:

$$\min_{\alpha} \sum_{i=1}^{N} \kappa_i^2$$

where κ_i is the curvature at each point i, defined as:

$$\kappa_i = \frac{x_i' y_i'' - y_i' x_i''}{(x_i'^2 + y_i'^2)^{3/2}}$$

with x_i and y_i representing the x and y spline interpolating between point r_i and r_{i+1}, and therefore x_i' and y_i' representing the first derivatives, and x_i'' and y_i'' the second derivatives of these splines. The curvature at each point i can be linearized and represented in matrix form as:

$$\kappa = \mathbf{Q}_x \mathbf{x}'' + \mathbf{Q}_y \mathbf{y}''$$

where $\mathbf{Q}_x$ and $\mathbf{Q}_y$ are matrices defined as:

$$\begin{cases} Q_{x_{i,j}} = \frac{y_i'^2}{(x_i'^2 + y_i'^2)^{\frac{3}{2}}} & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases}$$

and

$$\begin{cases} Q_{y_{i,j}} = \frac{x_i'^2}{(x_i'^2 + y_i'^2)^{\frac{3}{2}}} & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases}$$

The final QP formulation takes the standard form:

$$\min_{\alpha} \frac{1}{2} \alpha^\top H \alpha + f^\top \alpha$$

The bounds for α_i are set according to track constraints:

$$\alpha_i \in \left[-w_{\text{left},i} + \frac{w_{\text{veh}}}{2},\ w_{\text{right},i} - \frac{w_{\text{veh}}}{2} \right]$$

The definitions of f, H and P are extensively covered in the paper [1].
The iterative approach refines the solution by repeatedly updating the reference line to match the current optimized trajectory, recalculating H and f to reduce errors from linear approximations. This iterative QP therefore solves for the optimal shifts α_i, that are then applied to each midpoint at each iteration, achieving a minimum-curvature path after 3 or 4 optimization iterations.

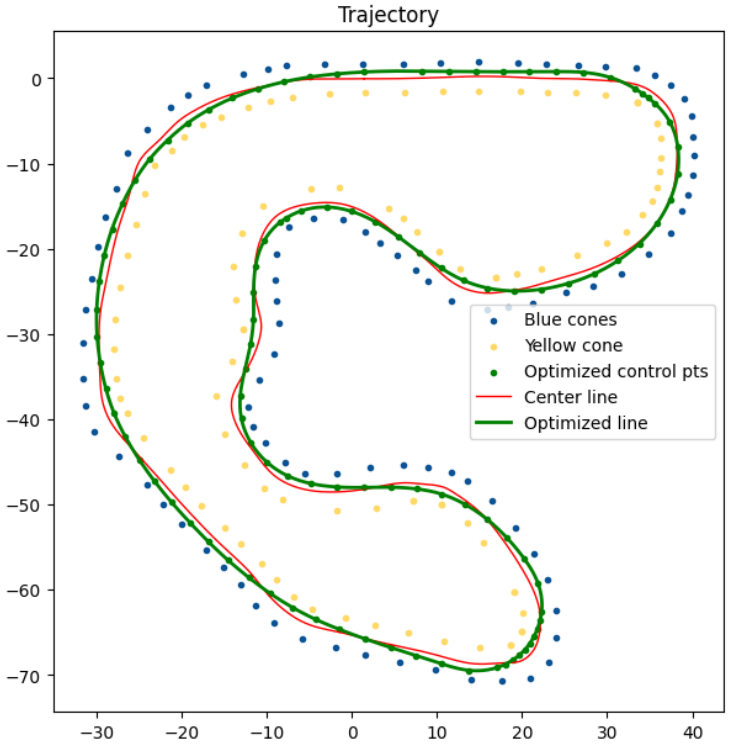

Figure 32: Optimized Trajectory

4.5 Path Tracking

4.5.1 Pure Pursuit

The Pure Pursuit control algorithm is a path-following technique used for steering towards a target point on a predefined path. The algorithm computes the required steering angle by continuously aiming towards a lookahead point that lies a fixed distance ahead on the path.

Let (x_t, y_t) represent the current position of the vehicle, and (x_p, y_p) be the coordinates of the lookahead point on the path, where the lookahead point is chosen such that the Euclidean distance from the vehicle is L_d (lookahead distance).

$$L_d = \sqrt{(x_p - x_t)^2 + (y_p - y_t)^2}$$

The algorithm works by computing the curvature κ of the circular arc that connects the vehicle's current position to the lookahead point. The steering angle δ is then computed based on this curvature.

First the heading error is computed as:

$$\alpha = \arctan\left(\frac{y_p - y_t}{x_p - x_t}\right) - \theta_t$$

where θ_t is the current heading angle of the vehicle. The curvature κ can be approximated as:

$$\kappa = \frac{2\sin\alpha}{L_d}$$

Finally the steering angle δ is then calculated using the curvature κ and the vehicle's wheelbase L:

$$\delta = \arctan(L \cdot \kappa)$$

Substituting κ into this equation, we obtain:

$$\delta = \arctan\left(\frac{2L\sin\alpha}{L_d}\right)$$

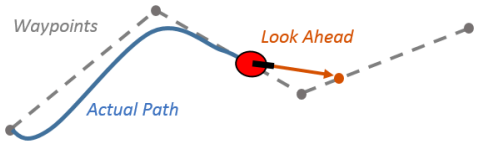

Figure 33: Pure Pursuit

4.5.2 Velocity Reference Based on Lateral Acceleration Limits

To define a velocity reference that accounts for both the curvature of the path and a limit on lateral acceleration, we start with the basic relationship between lateral acceleration, velocity, and curvature. The lateral acceleration experienced by a vehicle when following a curved path is given by:

$$a_{\text{lat}} = v^2 \kappa,$$

The lateral acceleration a_{lat} must not exceed a predefined maximum value, denoted as $a_{\text{lat\_max}}$.

$$v^2 \kappa \leq a_{\text{lat\_max}}.$$

Solving for the reference velocity v_{ref}, we obtain:

$$v_{\text{ref}} = \sqrt{\frac{a_{\text{lat\_max}}}{\kappa}}.$$

As the curvature increases (tighter turns), the reference velocity decreases to ensure that the lateral acceleration stays within the allowable limits.

In practical implementation, the maximum lateral acceleration $a_{\text{lat\_max}}$ is specified as a multiple of gravitational acceleration $g = 9.81\,\text{m/s}^2$, (g-force):

$$a_{\text{lat\_max}} = \alpha g = \alpha \cdot 9.81\,\text{m/s}^2,$$

$$v_{\text{ref}} = \sqrt{\frac{\alpha \cdot 9.81}{\kappa}}.$$

4.5.3 Velocity Profiling

The goal of velocity profiling is to compute an optimal velocity profile for a given set of trajectory points. The velocity profiler determines the optimal speed at each point, taking into account various physical constraints such as acceleration limits, drag, and maximum curvature. This results in a path where each point is associated with an optimal velocity value.

Optimal Velocity Calculation Let $p_i = (x_i, y_i)$ be the coordinates of each trajectory point. The goal is to calculate the maximum achievable velocity $v_{\text{max},i}$ at each point $\mathbf{p}_i$, given the constraints on the lateral acceleration a_y, longitudinal acceleration a_x, and drag forces. First, the maximum allowable velocity at each point in the trajectory is calculated by taking the track curvature into account. The curvature coefficient k_i for each point is calculated as described in 4.4.5, without the final smoothing step. The maximum achievable velocity $v_{\text{max},i}$ at each point is then given by:

$$v_{\text{max},i} = \sqrt{\frac{a_y}{k_i}}$$

subject to a predefined and adjustable speed limit, $v_{\text{max},i} \leq 50$ m/s.

Real Velocity Profile Calculation To compute the actual velocity profile the velocity in both forward and backward passes over the trajectory points, ensuring that the velocity respects the maximum allowable acceleration a_x and a_y values. This process is broken down as follows:

- **Forward Pass**:

  The forward pass takes in account the lateral acceleration constraints, as well as the longitudinal ones, when accelerating. Starting from an initial apex (local minimum of $v_{\max}$), the velocity at each point is computed by integrating the effect of longitudinal and lateral acceleration. The forward velocity v_i is given by:

$$v_i = \min\left(v_{\max,i}, v_{i-1} + \frac{a_x}{v_{i-1}} \cdot ds\right)$$

  where ds is the distance between point p_{i-1} and p_i, and a_x is calculated as:

$$a_x = a_{\text{x\_max}}\sqrt{1 - \left(\frac{a_y}{a_{\text{x\_max}}}\right)^2} - \text{drag} \cdot v_{i-1}^2$$

  where a_y is calculated as:

$$a_y = \min\left(a_{\text{lat\_max},i}, v_{i-1}^2 \cdot k_i\right)$$

  The drag is accounted for with a coefficient drag $= 0.00066$.

- **Backward Pass**:

  To ensure the constraints are also respected when braking, the backward pass recalculates the velocity from the last apex towards the initial point. If the velocity at point p_i is less than the velocity at point p_{i-1}, then we are in a braking phase, and we have to account for different longitudinal limits. This is because we could have a different maximum acceleration and deceleration limit. The velocity at each point v_i is adjusted based on braking acceleration a_{x_b}, calculated as:

$$v_i = \min\left(v_i, v_{i+1} + \frac{a_{x_b}}{v_{i+1}} \cdot ds\right)$$

  where:

$$a_{x_b} = a_{\min}\sqrt{1 - \left(\frac{a_y}{a_{\min}}\right)^2} + \text{drag} \cdot v_{i+1}^2$$

4.5.4 Model Predictive Contouring Control

Model Predictive Control In control theory, a Model Predictive Control (MPC) problem is formulated to optimize a system's future control actions over a prediction horizon while satisfying system constraints. Given a system model, MPC aims to compute a control sequence that minimizes a cost function, typically representing deviations from desired states and control effort. The problem can be expressed mathematically as follows:

$$\begin{aligned} \min \quad & J = \sum_{k=0}^{N-1} \left(x_k^T Q x_k + u_k^T R u_k \right) + x_N^T Q_f x_N \\ \text{s. t.} \quad & x_{k+1} = f(x_k, u_k), \\ & x_k \in \mathcal{X}, \quad u_k \in \mathcal{U}, \quad \forall k = 0, \ldots, N-1, \end{aligned}$$

where J represents the objective function, x_k is the system state at time step k, u_k is the control input, Q and R are weight matrices, and Q_f is the terminal weight. The function $f(x_k, u_k)$ represents the system dynamics, while $\mathcal{X}$ and $\mathcal{U}$ are the feasible sets of states and controls, respectively. By solving this optimization problem at each time step and applying only the first control action u_0 to the system, MPC provides an optimal and feasible control strategy that accounts for future constraints and objectives.

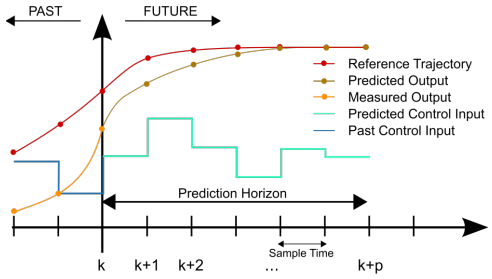

Figure 34: MPC receding horizon

Vehicle model The vehicle model employed in this MPC formulation is a bicycle model, a simplified representation of vehicle dynamics, where the vehicle is modeled as having two wheels: one at the front and one at the rear. This model assumes that both the front and rear axles behave as if there is a single tire at each axle, which simplifies the dynamics while retaining key characteristics of vehicle handling. Only the in-plane motions are considered, the pitch and roll dynamics as well as load changes are neglected.

The state and input vectors are represented as follows:

$$\mathbf{x} = [x, y, \theta, v_x, v_y, r]$$

$$\mathbf{u} = [D, \delta]$$

In this model, x and y are the global positions of the vehicle's center of gravity (in an inertial frame), θ is the yaw angle, v_x and v_y are the longitudinal and lateral velocities in the vehicle body frame, and r is the yaw rate. As control input D and δ represent respectively the throttle (as duty cycle) and the steering angle.

The dynamic equations are expressed as a set of first-order differential equations representing the evolution of these states. The state-space equations for the system are:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v_x} \\ \dot{v_y} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} v_x \cos(\theta) - v_y \sin(\theta) \\ v_x \sin(\theta) + v_y \cos(\theta) \\ r \\ \frac{1}{m}(F_x + F_{fric} - F_{yf} \sin(\delta)) + v_y r \\ \frac{1}{m}(F_{yf} \cos(\delta) + F_{yr}) - v_x r \\ \frac{1}{I_z}(l_f F_{yf} \cos(\delta) - l_r F_{yr}) \end{bmatrix}$$

where:

- m is the mass of the vehicle.
- I_z is the yaw moment of inertia of the vehicle.
- l_f and l_r are the distances from the center of gravity to the front and rear axles, respectively.
- F_x is the longitudinal force acting on the vehicle (applied at the CoG).

$$F_x = (C_{m1} - C_{m2} v_x) D$$

- F_{fric} is a contribute due to drag forces composed by the rolling resistance C_{r0}, which represents the resistance due to the friction between the tires and the road and aerodynamic drag, which models the increasing resistance experienced by the vehicle as its speed increases, due to air drag.

$$F_{fric} = -C_{r0} - C_{r2} v_x^2,$$

- F_{yf} and F_{yr} are the lateral tire forces at the front and rear axles, which are obtained from the simplified Pacejka tire model, which relates the tire forces to the slip angles α_f and α_r through the parameters D, C, and B:

$$F_{yf} = D_f \sin(C_f \arctan(B_f \alpha_f)), \quad F_{yr} = D_r \sin(C_r \arctan(B_r \alpha_r)),$$

where the slip angles α_f and α_r are given by:

$$\alpha_f = \delta - \arctan\left(\frac{v_y + l_f r}{v_x}\right), \quad \alpha_r = -\arctan\left(\frac{v_y - l_r r}{v_x}\right).$$

The terms $v_x \cos(\theta) - v_y \sin(\theta)$ and $v_x \sin(\theta) + v_y \cos(\theta)$ in the global position equations $\dot{x}$ and $\dot{y}$ represent the transformation from the vehicle's body frame to the inertial (global) frame. This allows the model to capture the vehicle's motion in real-world coordinates while maintaining the internal dynamics in the vehicle's local frame.

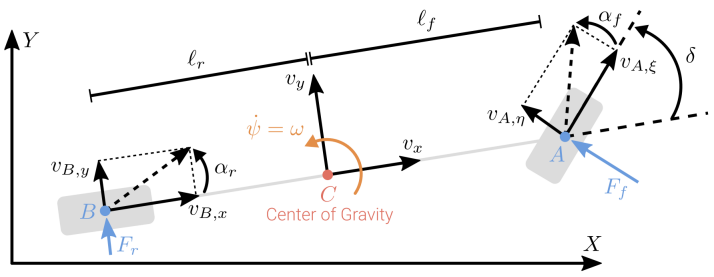

Figure 35: Bicycle model

Contouring formulation Taking as references [8] and [9], a high-performance adaptation of a Model Predictive Contouring Control has been developed. This approach aims to maximize the distance traveled along a reference path within a defined prediction horizon. The center path is employed as a reference path, primarily as a measure of progress, by assigning, as a strategy, low weights to the contouring error, which results in a trajectory that closely resembles those driven by expert drivers. This problem is solved in real-time by approximating the nonlinear programming (NLP) problem with local convex quadratic programming (QP) approximations at each sampling interval.

The Model Predictive Contouring Controller (MPCC) is designed to follow a predefined reference path, represented by X_{ref} and Y_{ref}. This is achieved by augmenting the model system explained before with an integrator state s, which approximates the progress along the reference path. The system is also extended considering input derivatives to control input variation.

The state and input vectors are represented as follows:

$$\mathbf{x} = [x, y, \theta, v_x, v_y, r, s, D, \delta, v_s]$$

$$\mathbf{u} = [dD, d\delta, dv_s]$$

And the system is described as follows:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v_x} \\ \dot{v_y} \\ \dot{r} \\ \dot{s} \\ \dot{D} \\ \dot{\delta} \\ \dot{v_s} \end{bmatrix} = \begin{bmatrix} v_x \cos(\theta) - v_y \sin(\theta) \\ v_x \sin(\theta) + v_y \cos(\theta) \\ r \\ \frac{1}{m}(F_x + F_{fric} - F_{yf} \sin(\delta)) + v_y r \\ \frac{1}{m}(F_{yf} \cos(\delta) + F_{yr}) - v_x r \\ \frac{1}{I_z}(l_f F_{yf} \cos(\delta) - l_r F_{yr}) \\ v_s \\ dD \\ d\delta \\ dv_s \end{bmatrix}$$

The s state is coupled to the real dynamics using the lag error $\hat{e}^l$, which is penalized in the cost function. Additionally, the contouring error $\hat{e}^c$ (lateral deviation from the reference path) is penalized in the cost function to ensure minimal deviation from the path.

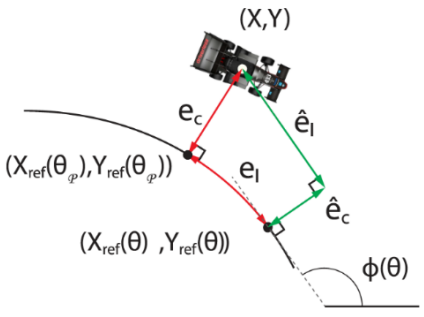

Figure 36: Lag error and contouring error

To further improve path tracking, progress along the reference path is maximized, encouraging the vehicle to follow the path as quickly as possible. Moreover, the rate of change of the inputs is penalized to avoid abrupt maneuvers. To ensure the vehicle remains within the track, track constraints are imposed, along with bounds on all states, inputs, and their rates. The resulting optimization problem is shown in the following equation:

$$
\begin{aligned}
\min \quad & \sum_{k=1}^{N} \begin{bmatrix} \hat{e}_k^c \\ \hat{e}_k^l \end{bmatrix}^T \begin{bmatrix} q_c & 0 \\ 0 & q_l \end{bmatrix} \begin{bmatrix} \hat{e}_k^c \\ \hat{e}_k^l \end{bmatrix} - q_v v_{s,k} + \Delta u_k^T R_\Delta \Delta u_k \\
\text{s.t.} \quad & x_0 = x(0) \\
& x_{k+1} = f(x_k, u_k) \\
& \hat{e}^c(x_k) = \sin\left(\Phi^{\text{ref}}(s_k)\right)\left(X_k - X^{\text{ref}}(s_k)\right) - \cos\left(\Phi^{\text{ref}}(s_k)\right)\left(Y_k - Y^{\text{ref}}(s_k)\right) \\
& \hat{e}^l(x_k) = -\cos\left(\Phi^{\text{ref}}(s_k)\right)\left(X_k - X^{\text{ref}}(s_k)\right) - \sin\left(\Phi^{\text{ref}}(s_k)\right)\left(Y_k - Y^{\text{ref}}(s_k)\right) \\
& \Delta u_k = u_k - u_{k-1} \\
& x_k \in \mathcal{X}_{\text{Track}} \\
& \underline{x} \leq x_k \leq \overline{x} \\
& \underline{u} \leq u_k \leq \overline{u} \\
& \underline{\Delta u} \leq \Delta u_k \leq \overline{\Delta u}
\end{aligned}
$$

HPIPM solver interface HPIPM (High-Performance Interior Point Method) [4] is an advanced open-source quadratic programming (QP) framework, designed to handle high-performance Model Predictive Control (MPC) applications. It provides efficient solutions for linear-quadratic optimal control problems (OCPs) using a family of interior-point method (IPM) solvers that balance computational speed and robustness. HPIPM is modular, supporting various QP types, including dense QPs, OCP QPs, and tree-structured OCP QPs. For MPC applications, such as path-following controllers with contouring formulations, HPIPM offers specialized routines for condensing and partially condensing OCP QPs, enhancing efficiency by transforming them into dense QPs or shortened-horizon OCP QPs. Its C-based API is encapsulated, making the framework extensible and efficient for embedded applications, with support for memory management tailored to real-time use.

Here is the description of HPIPM OCP QP fomrulation employed in this implementation:

$$
\begin{aligned}
\min_{x,u,s} \quad & \sum_{n=0}^{N} \frac{1}{2} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} R_n & S_n & r_n \\ S_n^T & Q_n & q_n \\ r_n^T & q_n^T & 0 \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} \\
& \quad + \frac{1}{2} \begin{bmatrix} s_n^l \\ s_n^u \\ 1 \end{bmatrix}^T \begin{bmatrix} Z_n^l & 0 & z_n^l \\ 0 & Z_n^u & z_n^u \\ (z_n^l)^T & (z_n^u)^T & 0 \end{bmatrix} \begin{bmatrix} s_n^l \\ s_n^u \\ 1 \end{bmatrix} \\
\text{s.t.} \quad & x_{n+1} = A_n x_n + B_n u_n + b_n, \quad n \in \mathcal{H} \setminus \{N\} \\
& \begin{bmatrix} \underline{u}_n \\ \underline{x}_n \\ \underline{d}_n \end{bmatrix} \leq \begin{bmatrix} J_n^{b,u} & 0 \\ 0 & J_n^{b,x} \\ D_n & C_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \end{bmatrix} + \begin{bmatrix} J_n^{s,u} \\ J_n^{s,x} \\ J_n^{s,g} \end{bmatrix} s_n^l, \quad n \in \mathcal{H} \\
& \begin{bmatrix} J_n^{b,u} & 0 \\ 0 & J_n^{b,x} \\ D_n & C_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \end{bmatrix} - \begin{bmatrix} J_n^{s,u} \\ J_n^{s,x} \\ J_n^{s,g} \end{bmatrix} s_n^u \leq \begin{bmatrix} \overline{u}_n \\ \overline{x}_n \\ \overline{d}_n \end{bmatrix}, \quad n \in \mathcal{H} \\
& s_n^l \geq \underline{s}_n^l, \quad n \in \mathcal{H} \\
& s_n^u \geq \underline{s}_n^u, \quad n \in \mathcal{H}
\end{aligned}
$$

The overall cost function for a single step n can be expressed as:

$$J_n = J_{n,1} + J_{n,2}$$

where:

- $J_{n,1}$ corresponds to the contribution from the first term which, involving u_n and x_n, determines the optimization function.
- $J_{n,2}$ corresponds to the contribution from the second term which, involving s_n^l and s_n^u, defines soft constraints cost function.

The first term can be expanded and expressed as:

$$\begin{aligned} J_{n,1} &= \frac{1}{2}\begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} R_n & S_n & r_n \\ S_n^T & Q_n & q_n \\ r_n^T & q_n^T & 0 \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} \\ &= \frac{1}{2}u_n^T R_n u_n + \frac{1}{2}x_n^T Q_n x_n + u_n^T S_n x_n + r_n^T u_n + q_n^T x_n \end{aligned}$$

The second term can be expanded and expressed as:

$$\begin{aligned} J_{n,2} &= \frac{1}{2}\begin{bmatrix} s_n^l \\ s_n^u \\ 1 \end{bmatrix}^T \begin{bmatrix} Z_n^l & 0 & z_n^l \\ 0 & Z_n^u & z_n^u \\ (z_n^l)^T & (z_n^u)^T & 0 \end{bmatrix} \begin{bmatrix} s_n^l \\ s_n^u \\ 1 \end{bmatrix} \\ &= \frac{1}{2}(s_n^l)^T Z_n^l (s_n^l) + \frac{1}{2}(s_n^u)^T Z_n^u (s_n^u) + (z_n^l)^T s_n^l + (z_n^u)^T s_n^u \end{aligned}$$

Combining those terms the optimization function for a single iteration n can be expressed as:

$$\begin{aligned} J_n &= J_{n,1} + J_{n,2} \\ &= \frac{1}{2}u_n^T R_n u_n + \frac{1}{2}x_n^T Q_n x_n + u_n^T S_n x_n + r_n^T u_n + q_n^T x_n \\ &\quad + \frac{1}{2}(s_n^l)^T Z_n^l (s_n^l) + \frac{1}{2}(s_n^u)^T Z_n^u (s_n^u) + (z_n^l)^T s_n^l + (z_n^u)^T s_n^u \end{aligned}$$

Hard and soft constraints are described in the condition expression below:

$$\begin{aligned} &\begin{bmatrix} \underline{u}_n \\ \underline{x}_n \\ \underline{d}_n \end{bmatrix} \leq \begin{bmatrix} J_n^{b,u} & 0 \\ 0 & J_n^{b,x} \\ D_n & C_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \end{bmatrix} + \begin{bmatrix} J_n^{s,u} \\ J_n^{s,x} \\ J_n^{s,g} \end{bmatrix} s_n^l, \quad n \in \mathcal{H}, \\ &\begin{bmatrix} J_n^{b,u} & 0 \\ 0 & J_n^{b,x} \\ D_n & C_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \end{bmatrix} - \begin{bmatrix} J_n^{s,u} \\ J_n^{s,x} \\ J_n^{s,g} \end{bmatrix} s_n^u \leq \begin{bmatrix} \overline{u}_n \\ \overline{x}_n \\ \overline{d}_n \end{bmatrix}, \quad n \in \mathcal{H}, \\ &s_n^l \geq \underline{s}_n^l, \quad n \in \mathcal{H}, \\ &s_n^u \geq \underline{s}_n^u, \quad n \in \mathcal{H}. \end{aligned}$$

where J matricies are made by rows of an identity matrix to select some components in input, state, and combined constraints ($J_n^{b,u}, J_n^{b,x}$), or to select which constraints are softened ($J_n^{s,u}, J_n^{s,x}, J_n^{s,g}$).

Expanding the constraints expressions the subsequent constraints fomrulation are obtained:

$$\begin{aligned} \underline{u}_n &\leq J_n^{b,u} u_n + J_n^{s,u} s_n^l, & J_n^{b,u} u_n - J_n^{s,u} s_n^u &\leq \overline{u}_n, \\ \underline{x}_n &\leq J_n^{b,x} x_n + J_n^{s,x} s_n^l, & J_n^{b,x} x_n - J_n^{s,x} s_n^u &\leq \overline{x}_n, \\ \underline{d}_n &\leq C_n x_n + D_n u_n + J_n^{s,g} s_n^l, & C_n x_n + D_n u_n - J_n^{s,g} s_n^u &\leq \overline{d}_n. \end{aligned}$$

System linearization and discretization To linearize the nonlinear vehicle dynamics model, we compute the Jacobians of the system's equations with respect to the state and input vectors, resulting in the matrices A, B, and g. The linearized continuous-time dynamics can be expressed in the form:

$$\dot{x} = Ax + Bu + g$$

where A is the Jacobian of the system dynamics with respect to the state vector x, B is the Jacobian of the system dynamics with respect to the input vector u, and g is the zero-order term representing the deviation of the actual dynamics from the linearized approximation. The zero-order term g is used to compensate the linearization in a non equilibrium point (the current vehicle state). Each entry in A and B is derived by computing the partial derivatives of the system dynamics with respect to each state and input variable at the operating point.

To use this linearized model in a discrete-time setting, we convert A, B, and b into their discrete-time equivalents, A_d, B_d, and g_d, using a suitable discretization method (Mixed RK4-EXPM). This results in the discrete-time dynamics:

$$x[k+1] = A_d x[k] + B_d u[k] + g_d$$

where $A_d = e^{A\Delta t}$, $B_d = \left(\int_0^{\Delta t} e^{A\tau}\, d\tau\right) B$, and $g_d = g\Delta t$, with Δt being the discretization time step.

Cost matrices definition

$$Q_n = \begin{bmatrix} Q_{xx} & Q_{xy} & & \cdots & \cdots & & Q_{xs} & & & \\ Q_{xy} & Q_{yy} & & \cdots & \cdots & & Q_{ys} & & & \\ & & Q_{\theta\theta} & & & & & & & \\ \vdots & \vdots & & \ddots & & & & & & \\ \vdots & \vdots & & & \ddots & & & & & \\ & & & & & Q_{rr} & & & & \\ Q_{xs} & Q_{ys} & & & & & Q_{ss} & & & \\ & & & & & & & Q_{DD} & & \\ & & & & & & & & Q_{\delta\delta} & \\ & & & & & & & & & Q_{v_s v_s} \end{bmatrix}, \quad q_n = \begin{bmatrix} q_x \\ q_y \\ q_\theta \\ \vdots \\ \vdots \\ \\ q_s \\ \vdots \\ \\ q_{v_s} \end{bmatrix}$$

$$R_n = \begin{bmatrix} R_{DD} & & \\ & R_{\delta\delta} & \\ & & R_{v_s v_s} \end{bmatrix}, \quad r_n = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \quad S_n = 0$$

Input cost: The input cost factors are used to control the input (and derivatives) change over time.

$$\begin{aligned} Q_{DD} &= 2r\_D & R_{DD} &= 2r\_dD \\ Q_{\delta\delta} &= 2r\_\delta & R_{\delta\delta} &= 2r\_d\delta \\ Q_{v_s v_s} &= 2r\_v_s & R_{v_s v_s} &= 2r\_dv_s \end{aligned}$$

where $r\_D, r\_\delta, r\_v_s, r\_dD, r\_d\delta, r\_dv_s$ are parameters.

Heading cost: The heading cost factor is used to limit the error due to the difference between the vehicle heading and the target vehicle heading, limiting also the yaw rate changing.

$$\begin{aligned} e_h &= (\theta - \theta_{\text{ref}})^2 = \theta^2 - 2\theta\theta_{\text{ref}} - \theta_{\text{ref}}^2 \\ Q_{\theta\theta} &= 2q\_\mu \\ q_\theta &= -2q\_\mu\theta_{\text{ref}} \\ Q_{rr} &= q\_r \end{aligned}$$

where q_μ, q_r are parameters.

Contouring cost: This contouring formulation makes the vehicle stay close to the middle line trying to maximize the progress over time v_s.

$$\begin{aligned} \hat{e}^c &= \sin(\theta_{\text{ref}}(s))(X - X_{\text{ref}}(s)) - \cos(\theta_{\text{ref}}(s))(Y_k - Y_{\text{ref}}(s)) \\ \hat{e}^l &= -\cos(\theta_{\text{ref}}(s))(X - X_{\text{ref}}(s)) - \sin(\theta_{\text{ref}}(s))(Y_k - Y_{\text{ref}}(s)) \end{aligned}$$

Contouring error $\hat{e}^c$ is employed to make the vehicle stay close to the center line, while lag error $\hat{e}^l$ is used to accurately estimate the contouring error, reducing s state prediction error.

Using Taylor approximation on $\hat{e}^c$ around inital guess x_i:

$$\begin{aligned}\hat{e}^c &= \hat{e}^c(x_i) + \nabla\hat{e}^c(x - x_i) = \\ &= [\hat{e}^c(x_i) - \nabla\hat{e}^c x_i] + \nabla\hat{e}^c x = \\ &= \hat{e}^c_0 + \nabla\hat{e}^c x, \quad \hat{e}^c_0 = \hat{e}^c(x_i) - \nabla\hat{e}^c x_i\end{aligned}$$

The quadratic error is so computed as follows:

$$\begin{aligned}(\hat{e}^c(x))^2 &= (\hat{e}^c_0 + \nabla\hat{e}^c x)^2 = \\ &= (\hat{e}^c_0)^2 + 2\hat{e}^c_0\nabla\hat{e}^c x + (\nabla\hat{e}^c)^2 x^2\end{aligned}$$

Defining the error vector below the Jacobians are computed as follows:

$$e = \begin{bmatrix}\hat{e}^c \\ \hat{e}^l\end{bmatrix}^T, \quad de = \begin{bmatrix}(\nabla\hat{e}^c)^T \\ (\nabla\hat{e}^l)^T\end{bmatrix} = \begin{bmatrix}\frac{\partial\hat{e}^c}{\partial x} & \frac{\partial\hat{e}^c}{\partial y} & \dots & \frac{\partial\hat{e}^c}{\partial s} & \dots \\ \frac{\partial\hat{e}^l}{\partial x} & \frac{\partial\hat{e}^l}{\partial y} & \dots & \frac{\partial\hat{e}^l}{\partial s} & \dots\end{bmatrix}$$

Finally Q and q matricies contributions to contouring and lag errors are computed as:

$$\begin{aligned}Q_{xx} &= 2\left(q_c\left(\frac{\partial\hat{e}^c}{\partial x}\right)^2 + q_l\left(\frac{\partial\hat{e}^l}{\partial x}\right)^2\right) \\ Q_{yy} &= 2\left(q_c\left(\frac{\partial\hat{e}^c}{\partial y}\right)^2 + q_l\left(\frac{\partial\hat{e}^l}{\partial y}\right)^2\right) \\ Q_{ss} &= 2\left(q_c\left(\frac{\partial\hat{e}^c}{\partial s}\right)^2 + q_l\left(\frac{\partial\hat{e}^l}{\partial s}\right)^2\right) \\ Q_{xy} &= 2\left(q_c\left(\frac{\partial\hat{e}^c}{\partial x}\frac{\partial\hat{e}^c}{\partial y}\right) + q_l\left(\frac{\partial\hat{e}^l}{\partial x}\frac{\partial\hat{e}^l}{\partial y}\right)\right) \\ Q_{sx} &= 2\left(q_c\left(\frac{\partial\hat{e}^c}{\partial s}\frac{\partial\hat{e}^c}{\partial x}\right) + q_l\left(\frac{\partial\hat{e}^l}{\partial s}\frac{\partial\hat{e}^l}{\partial x}\right)\right) \\ Q_{sy} &= 2\left(q_c\left(\frac{\partial\hat{e}^c}{\partial s}\frac{\partial\hat{e}^c}{\partial y}\right) + q_l\left(\frac{\partial\hat{e}^l}{\partial s}\frac{\partial\hat{e}^l}{\partial y}\right)\right)\end{aligned}$$

$$\begin{aligned}q_x &= 2q_c\hat{e}^c_0\frac{\partial\hat{e}^c}{\partial x} + 2q_l\hat{e}^l_0\frac{\partial\hat{e}^l}{\partial x} \\ q_y &= 2q_c\hat{e}^c_0\frac{\partial\hat{e}^c}{\partial y} + 2q_l\hat{e}^l_0\frac{\partial\hat{e}^l}{\partial y} \\ q_s &= 2q_c\hat{e}^c_0\frac{\partial\hat{e}^c}{\partial s} + 2q_l\hat{e}^l_0\frac{\partial\hat{e}^l}{\partial s} \\ q_{v_s} &= -q\_v_s\end{aligned}$$

where q_c, q_l, q_v_s are parameters.

Constraint matrices definition

$$C_n = \begin{bmatrix} C_{tx} & C_{ty} & \cdots & & \cdots & & \cdots \\ & \cdots & C_{t_r v_x} & C_{t_r v_y} & C_{t_r r} & C_{t_r D} & \\ & \cdots & C_{\alpha_f v_x} & C_{\alpha_f v_y} & C_{\alpha_f r} & & C_{\alpha_f \delta} \end{bmatrix}$$

$$D_n = 0$$

$$\underline{d}_n = \begin{bmatrix} \underline{d}_t \\ \underline{d}_{t_r} \\ \underline{d}_{\alpha_f} \end{bmatrix}, \quad \overline{d}_n = \begin{bmatrix} \overline{d}_t \\ \overline{d}_{t_r} \\ \overline{d}_{\alpha_f} \end{bmatrix}$$

Track constraint: The track constraint ensure that the vehicle stays, with certain limits, close to the center line.

Let's define the vector p_{center} as the perpendicular vector to the spline approximating center line in the s_{ref} point:

$$p_{\text{center}} = \begin{bmatrix} -\frac{dy}{ds}\Big|_{s_{\text{ref}}} \\ \frac{dx}{ds}\Big|_{s_{\text{ref}}} \end{bmatrix}$$

Define now, using $r\_\text{in}, r\_\text{out}$ parameters, the inner and outer boundaries that needs to be respected as:

$$P_{\text{out}} = P_{\text{ref}} + r\_\text{out} \cdot p_{\text{center}} = \begin{bmatrix} x_{\text{out}} \\ y_{\text{out}} \end{bmatrix}$$
$$P_{\text{in}} = P_{\text{ref}} - r\_\text{in} \cdot p_{\text{center}} = \begin{bmatrix} x_{\text{in}} \\ y_{\text{in}} \end{bmatrix}$$

Formally matricies parameters are so defined as:

$$C_{tx} = -\left.\frac{dy}{ds}\right|_{s_{\text{ref}}}$$
$$C_{ty} = \left.\frac{dx}{ds}\right|_{s_{\text{ref}}}$$

$$\underline{d}_t = -\left.\frac{dy}{ds}\right|_{s_{\text{ref}}} \cdot x_{\text{out}} + \left.\frac{dx}{ds}\right|_{s_{\text{ref}}} \cdot y_{\text{out}}$$
$$\overline{d}_t = -\left.\frac{dy}{ds}\right|_{s_{\text{ref}}} \cdot x_{\text{in}} + \left.\frac{dx}{ds}\right|_{s_{\text{ref}}} \cdot y_{\text{in}}$$

Tire friction ellipse constraint:

$$\left(\epsilon_{\text{long}}\frac{F_x}{F_N}\right)^2+\left(\frac{F_y}{F_N}\right)^2\leq\left(\frac{\epsilon_{\text{eps}}D}{F_N}\right)^2$$

where F_x, F_y and F_N are the longitudinal, lateral and normal forces on the tire. Scaling factor for longitudinal force and grip level are represented by ϵ_{long} and ϵ_{eps}. D is a parameter associated with the friction limit of the tire, determining the maximum combined force it can handle.

This equation limits the combined longitudinal and lateral forces acting on tires. It's known as a friction ellipse because the relationship between F_x and F_y forms an ellipse when plotted. The friction ellipse constraint is essential to keep the tire forces within the adhesion limits of the tire. Exceeding these limits would cause the tire to lose grip, resulting in instability. The ellipse defines the maximum force capacity based on friction, and the vehicle must operate within this boundary to maintain control.

Normal forces for rear and front tires are computed as follows:

$$F_{N,r}=\frac{l_r}{l_f+l_r}mg,\quad F_{N,f}=\frac{l_f}{l_f+l_r}mg$$

Defining tire constraint as:

$$F_{\text{tire}}=\left(\epsilon_{\text{long}}\frac{F_x}{F_N}\right)^2+\left(\frac{F_y}{F_N}\right)^2$$

The Jacobian is then defined as:

$$J_{F_{\text{tire}}}=\begin{bmatrix}\dots & \dots & \frac{\partial F_{\text{tire}}}{\partial v_x} & \frac{\partial F_{\text{tire}}}{\partial v_y} & \frac{\partial F_{\text{tire}}}{\partial r} & \dots & \frac{\partial F_{\text{tire}}}{\partial D} & \dots\end{bmatrix}$$

And maximum force as:

$$F_{\max}=\left(\frac{\epsilon_{\text{eps}}D}{F_N}\right)^2$$

Using Taylor expansion the constraint can be now defined as:

$$0\leq J_{F_{\text{tire}}}(x-x_i)+F_{\text{tire}}(x_i)\leq F_{\max}$$

$$J_{F_{\text{tire}}}x_i-F_{\text{tire}}(x_i)\leq J_{F_{\text{tire}}}x\leq F_{\max}+J_{F_{\text{tire}}}x_i-F_{\text{tire}}(x_i)$$

The entries of the matrices in the solver formulation are expressed below:

$$C_{t_rv_x}=\frac{\partial F_{\text{tire}}}{\partial v_x}$$
$$C_{t_rv_y}=\frac{\partial F_{\text{tire}}}{\partial v_y}$$
$$C_{t_rr}=\frac{\partial F_{\text{tire}}}{\partial r}$$
$$C_{t_rD}=\frac{\partial F_{\text{tire}}}{\partial D}$$

$$\underline{d}_{t_r} = J_{F_{\text{tire}}} x_i - F_{\text{tire}}(x_i)$$
$$\overline{d}_{t_r} = F_{\max} + J_{F_{\text{tire}}} x_i - F_{\text{tire}}(x_i)$$

Front slip tire constraint:

$$\alpha_{\min} \leq \alpha_f \leq \alpha_{\max}$$

In vehicles, especially in sharp turns or high-speed maneuvers, the slip angle can increase, causing a loss of traction if it becomes too large. The slip angle constraint ensures that the front wheel's slip angle stays within a safe range. If α_f becomes too large, the front wheels may lose lateral grip, causing understeer (the vehicle doesn't turn as much as commanded). If it's too small, the vehicle might not be utilizing available tire forces optimally for steering.

The Jacobian of α_f is computed as follows:

$$J_{\alpha_f} = \begin{bmatrix} \cdots & \cdots & \frac{\partial \alpha_f}{\partial v_x} & \frac{\partial \alpha_f}{\partial v_y} & \frac{\partial \alpha_f}{\partial r} & \cdots & \frac{\partial \alpha_f}{\partial \delta} & \cdots \end{bmatrix}$$

Using Taylor expansion the constraint can be now defined as:

$$-\alpha_{\max} \leq J_{\alpha_f}(x - x_i) + \alpha_f(x_i) \leq \alpha_{\max}$$

$$-\alpha_{\max} + J_{\alpha_f} x_i - \alpha_f(x_i) \leq J_{\alpha_f} x \leq \alpha_{\max} + J_{\alpha_f} x_i - \alpha_f(x_i)$$

The entries of the matrices in the solver formulation are expressed below:

$$C_{\alpha_f v_x} = \frac{\partial \alpha_f}{\partial v_x}$$
$$C_{\alpha_f v_y} = \frac{\partial \alpha_f}{\partial v_y}$$
$$C_{\alpha_f r} = \frac{\partial \alpha_f}{\partial r}$$
$$C_{\alpha_f \delta} = \frac{\partial \alpha_f}{\partial \delta}$$

$$\underline{d}_{\alpha_f} = -\alpha_{\max} + J_{\alpha_f} x_i - \alpha_f(x_i)$$
$$\overline{d}_{\alpha_f} = \alpha_{\max} + J_{\alpha_f} x_i - \alpha_f(x_i)$$

Soft constraint cost matrices definition

$$Z_n = \begin{bmatrix} Z_t & & \\ & Z_{t_r} & \\ & & Z_{\alpha_f} \end{bmatrix}, \quad z_n = \begin{bmatrix} z_t \\ z_{t_r} \\ z_{\alpha_f} \end{bmatrix}$$

$$\begin{aligned} Z_t &= \text{sc\_quad\_track} \\ Z_{t_r} &= \text{sc\_quad\_tire} \\ Z_{\alpha_f} &= \text{sc\_quad\_alpha} \\ z_t &= \text{sc\_lin\_track} \\ z_{t_r} &= \text{sc\_lin\_tire} \\ z_{\alpha_f} &= \text{sc\_lin\_alpha} \end{aligned}$$

These parameters are used to tune how much constraints are strict or softened. The higher the parameter value, the more the constraint is considered strict.

Simulation results In Figure 37 are shown simulation results plots obtained with an offline simulation using as plant model the same vehicle model employed in MPCC poblem.

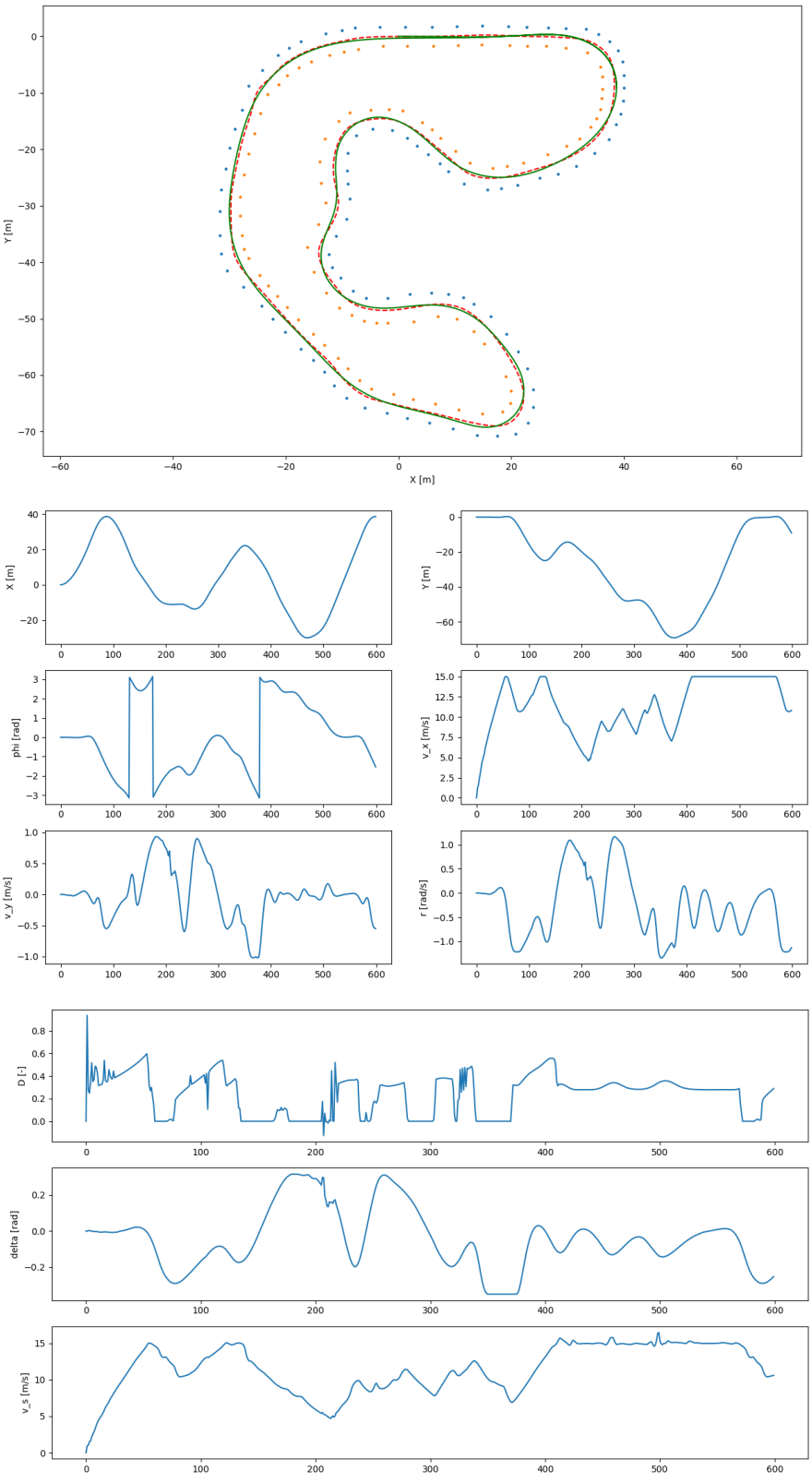

Figure 37: MPCC simulation results

5 Project configuration

5.1 ROS2 framework

Figure 38: ROS2 framework with C++ and Python

ROS 2 (Robot Operating System 2) is a powerful framework designed to facilitate communication between distributed processes, often used in robotic applications. It provides a middleware that allows different processes, known as nodes, to communicate through publish-subscribe mechanisms. ROS 2 supports both C++ and Python as primary programming languages, making it highly versatile. The structure of ROS 2 applications typically rely on packages, which are collections of nodes, configurations, and resources bundled together for modular development. Each package may contain several nodes written in either C++ or Python, and these nodes interact by publishing and subscribing to topics, calling services, or setting/getting parameters. The flexible communication architecture of ROS 2, based on DDS (Data Distribution Service), enables reliable and real-time communication between nodes across different machines and platforms, making it an ideal choice for complex robotic systems.

In our application ROS 2 humble version is employed.

5.1.1 Launch configuration

In ROS 2, launch configurations provide a flexible and modular way to start multiple nodes and manage their execution. A typical launch setup is often defined in a launch.py file within a package, which serves as the entry point for launching one or more nodes. To support complex systems, a package's launch.py file can call multiple other launch.py files located in different packages, creating a hierarchical structure of launch files. This allows developers to organize and manage large-scale applications efficiently.

In our application, a main launch packages contains a launch.py which collects all settings parameters by argument and call other packages launch.py propagating, for each node, its arguments.

5.1.2 RViz2 visualization tool

RViz2 is a powerful 3D visualization tool in ROS 2, designed to assist in visualizing and debugging robotic systems by providing real-time feedback on the state of various sensors, robot models, and algorithms. As an essential component of the ROS 2 ecosystem, RViz2 allows developers to view a wide range of data, such as point clouds, images, robot trajectories, and transformations, through a customizable interface. This tool supports the visualization of various types of ROS 2 messages, enabling developers to monitor topics and interact with data streams as nodes execute.

In our application this tool is used to visualize point clouds, images and plots dealing with SLAM, path planning and path tracking algorithms.

5.1.3 Rqt

RQt is a flexible, graphical user interface (GUI) framework in ROS 2 that allows users to visualize, monitor, and control various aspects of a robotic system. RQt provides a collection of plugins that offer functionality, mainly for plotting data, inspecting node graphs (rqt_graph), visualize message contents, and interacting with parameters or services.

5.1.4 Bags

ROS 2 bags are an essential tool for recording and replaying data in robotic systems, allowing developers to capture message traffic for later analysis and debugging. A bag file stores in a database the messages exchanged between nodes during runtime, with timestamp information. These recorded sessions enable developers to inspect and replay the exact sequence of events, making it easier to diagnose issues or test new algorithms without needing to recreate real-world conditions.

In our application bags are highly used to collect data during testing and replaying them to test and improve algorithms with real data. A bag_logger node is responsible to subscribe topics to be logged and handle the bag writing.

5.2 Automatic start

A system service (Autostart.service) is employed to automatically start the ROS 2 pipeline upon system wake-up, ensuring that the robotic processes are initialized without manual intervention. This service is configured to call the general launch.py launching the necessary ROS 2 nodes and bring the entire system online. The service is managed by the operating system's service manager systemd.

6 Simulation

Figure 39: SCD Simulator

Figure 40: Simulation environment with Docker and Unity

A custom simulator is employed to provide a flexible and modular framework for simulating the vehicle, built on ROS 2 to take advantage of its modularity and plug-and-play capabilities.

6.1 Docker Virtualization

ROS 2 is executed inside a Docker container, providing a virtualized environment that isolates the simulation system and simplifies deployment across various platforms. This containerized setup enables easy management of dependencies and ensures reproducibility, while also facilitating the integration of the ROS 2 nodes with the external Unity application. The overall architecture allows developers to independently modify different components of the simulation, making it highly adaptable to various testing scenarios and configurations.

6.2 Unity application

This simulator ROS2 pipeline, running into a docker container, communicates with a Unity application in loopback via a TCP-based inter-process communication (TCP IPC) system. The Unity application handle visualization, simulation of the environment and the simulation of sensor data (LiDAR and cameras). A percetion_simulation ROS2 node allow the handling of simulated data from unity sensors and allow also a simulation of SLAM, path planning and path tracking algorithms by directly sending cones ignoring the perception computations. Sensors synchroniation is performed on unity application, sending cameras data when triggered by the publishing of LiDAR data at the rate of 10 Hz.

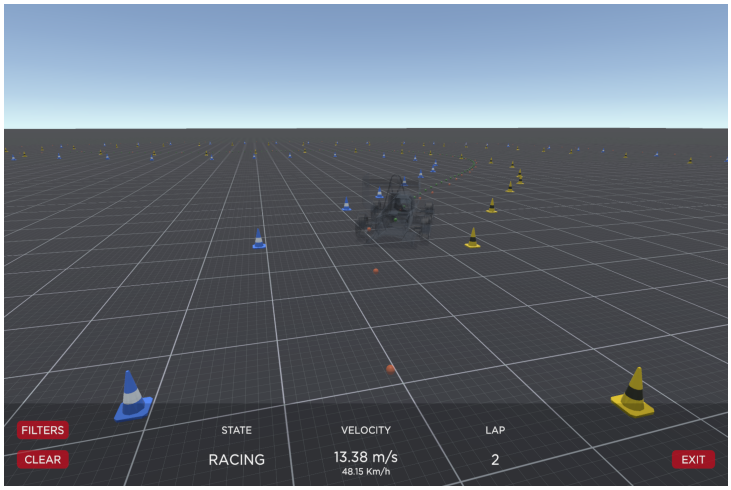

Figure 41: SCD Simulator interface

6.3 Simulator to Autonomous System Deployment

The power of this simulation setup stands on the modularity, offered by packages and nodes structure of ROS2, so that a package can be tested on the simulator and directly deployed to the Autonomous System project. Some little differences are handled inside packages to fill the differences between simulation scenario and the real one, like default parameters and raw sensors data differences between the simualted and real environments, in order to make the code directly deployable.

6.4 Vehicle model

The vehicle model employed in the simulation is the same exposed in the paragraph 4.5.4 used in MPCC. This environment simulate an ideal environment in which the plant to control is perfectly modeled. A future improvement would be replace this model with a more complex solution to emulate rality gap.

6.5 Random Track Generation

A random track generator has been developed to provide robust simulations on FSAE rule compliant random tracks. The track generation process employs a combination of mathematical techniques and algorithms to create a random 2D racetrack.

6.5.1 Path Generation

The generation of a random racetrack involves the parameterization of points around the unit circle given by

$$z_t = e^{2\pi i t/n_{\text{points}}} \quad \text{for } t = 0, 1, \ldots, n_{\text{points}} - 1.$$

A wave function is constructed using a Fourier series expansion as follows:

$$W(z) = \sum_{f=2}^{f_{max}} \frac{z^f}{\phi_f \cdot (f+1)} + \frac{\phi_f}{z^f \cdot (f-1)},$$

where ϕ_f is a phase factor defined by $\phi_f = e^{2\pi ir}$, where r is random. The points along the track are represented as

$$P(t) = z + A \cdot W(z),$$

where A is the starting amplitude. The first and second derivatives are computed as:

$$\frac{dP}{dt} = iz\left(1 + A \cdot \frac{dW}{dz}\right).$$

$$\frac{d^2P}{dt^2} = i\frac{dz}{dt}\left(z \cdot A \cdot \frac{d^2W}{dz^2} + 1 + A \cdot \frac{dW}{dz}\right)$$

The corner radius R is defined as

$$R = \frac{1}{\left|\frac{d^2P}{dt^2}\right|},$$

and is scaled to meet a minimum radius condition:

$$\text{scale} = \frac{R_{\min}}{\min\left(|R|\right)}.$$

The total track length is computed with

$$L = \text{scale} \cdot \sum_{k=1}^{n_{\text{points}}-1} |P_k - P_{k-1}|,$$

Then amplitude A is fine tuned to reach a track length with a certain accuracy. After the amplitude tuning all points are passed through the scale factor to meet curvature radius constraints.

6.5.2 Self-Intersection Check

The algorithm checks if a racetrack path intersects itself by first calculating the normal vectors $N(t)$ at each point $P(t)$ along the path. The normals are given by:

$$N(t) = \frac{i \cdot S(t)}{|S(t)|}$$

where $S(t)$ are the slopes at each point, and i is the imaginary unit. Two offset paths are then created: the outer boundary $P_{outer}(t)$ and the inner boundary $P_{inner}(t)$, defined as:

$$P_{outer}(t) = P(t) + margin \cdot N(t)$$
$$P_{inner}(t) = P(t) - margin \cdot N(t)$$

Finally intersections are checked recursively along these offset paths.

6.5.3 Starting Line Selection

The function selects a suitable starting point along the track by first computing the curvature κ at each point, where the curvature is defined as the reciprocal of the corner radius R:

$$\kappa = \frac{1}{R}$$

The points are then sorted by curvature, and only the points with the lowest curvature are considered. A smoothing operation is applied over a stretch of the track with a length proportional to the starting straight length. The starting point is chosen as the point at the end of the stretch with the smallest average curvature. Finally, the track is translated such that the starting point is at $(0, 0)$ and rotated so that the starting direction faces the positive x axis.

6.5.4 Cones Placement

The density D of cones along the track is determined by the curvature (reciprocal of the corner radius R) of the track, and is given by:

$$D(R) = D_{min} + \frac{c_1}{R} + \frac{c_2}{|R|}$$

where:

$$c_1 = \frac{(D_{max} - D_{min})}{2} \left((1 - b_{cone\_spacing}) \cdot R_{min} - (1 + b_{cone\_spacing}) \cdot \frac{w_{track}}{2} \right)$$

$$c_2 = \frac{(D_{max} - D_{min})}{2} \left((1 + b_{cone\_spacing}) \cdot R_{min} - (1 - b_{cone\_spacing}) \cdot \frac{w_{track}}{2} \right)$$

where $b_{cone\_spacing}$ is the cone spacing bias and w_{track} is the track width.

When $b_{cone\_spacing} = 0$ cone spacing is uniform, regardless of whether they are on the inside or outside of the curve, while with $b_{cone\_spacing} > 0$ it decreases the spacing on the inside of the turn. With $b_{cone\_spacing} < 0$ a reversed effect is obtained.

Left and right boundaries are computed with an offset from the center line of the track as:

$$L(t) = P(t) + N(t) \cdot \frac{w_{track}}{2}$$

$$R(t) = P(t) - N(t) \cdot \frac{w_{track}}{2}$$

Finally a subset of points in both boundaries are selected based on the density D to become cones.

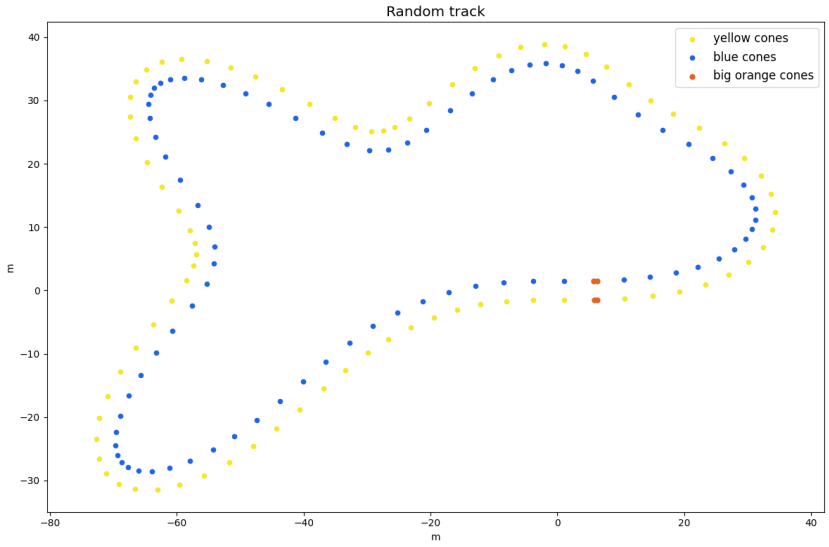

Figure 42: Random generated track

7 ROS2 Pipeline

Figure 43 illustrates the ROS2 pipeline, highlighting the communication topics between various package nodes. Below is a legend explaining the classification of the packages:

- **Green**: Perception packages, responsible for processing sensor data and understanding the environment.
- **Red**: Packages for SLAM, path planning, and path tracking, which handle mapping, trajectory generation, and control.
- **Orange**: Auxiliary packages. Finite state machine, logger, TCP inter-process communication, and visualization packages, which are accessible by all nodes.
- **Yellow**: TCP endpoint to retrieve unity simulated sensors data, perception simulation and vehicle model packages, primarily used for simulation purposes.
- **Light blue**: ECU tasks, managing low-level vehicle control and state estimation.

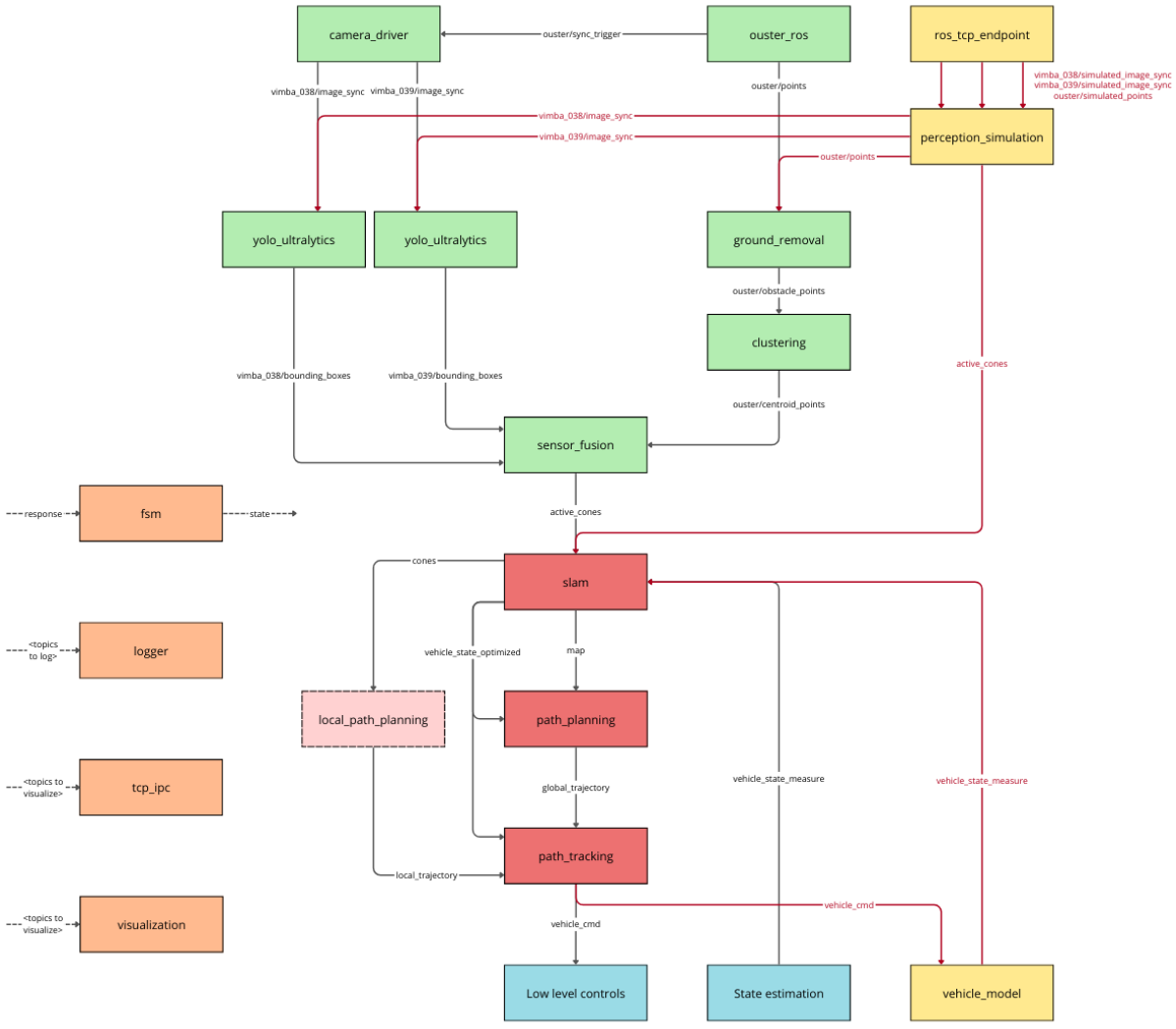

Figure 43: ROS2 packages pipeline

References

[1] Leonhard Hermansdorfer Johannes Betz Markus Lienkamp Alexander Heilmeier, Alexander Wischnewski and Boris Lohmann. Minimum curvature trajectory planning and control for an autonomous race car. *Vehicle System Dynamics*, 2020.

[2] Lihao Liu et al. Ao Wang, Hui Chen. Yolov10: Real-time end-to-end object detection. *arXiv preprint arXiv:2405.14458*, 2024.

[3] Frank Dellaert and Michael Kaess. Square root sam: Simultaneous localization and mapping via square root information smoothing. *International Journal of Robotics Research*, 2006. To appear.

[4] Gianluca Frison and Moritz Diehl. Hpipm: a high-performance quadratic programming framework for model predictive control. *arXiv preprint arXiv:2003.02547*, 2020.

[5] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based slam. *IEEE Transactions on Intelligent Transportation Systems Magazine*, 2010.

[6] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics yolov8, 2023.

[7] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A general framework for graph optimization. In *Proceedings of the 2011 IEEE International Conference on Robotics and Automation*, 2011.

[8] Denise Lam, Chris Manzie, and Malcolm Good. Model predictive contouring control. 2010.

[9] Alexander Liniger, Alexander Domahidi, and Manfred Morari. Optimization-based autonomous racing of 1:43 scale rc cars. *arXiv preprint arXiv:1711.07300*, 2017.

[10] SAE International. *Formula SAE 2024 Rules*. Society of Automotive Engineers, 2024. Available online: `https://www.fsaeonline.com`.

[11] X. Shen, E. Frazzoli, D. Rus, and M. H. Ang. Fast joint compatibility branch and bound for feature cloud matching. In *Proceedings of the 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1757–1764, 2016.