Squadra Corse Driverless

Autonomous System
Documentation

Politecnico di Torino

October 23, 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 [3] 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.

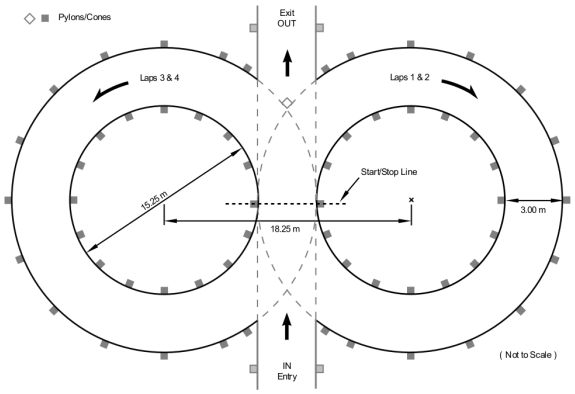

Figure 1: Skidpad track specifications

- **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.

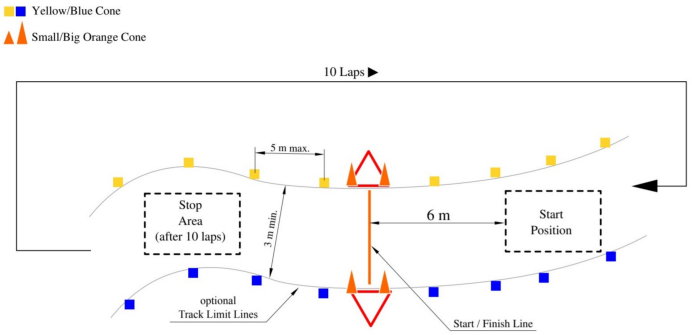

Figure 2: Autcross/Trackdrive track specifications

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.

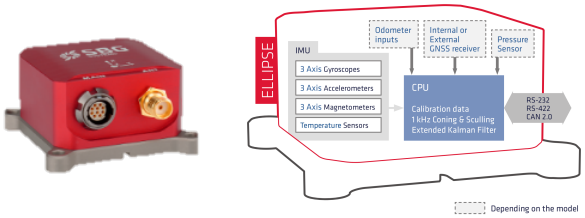

Figure 5: IMU SBG Ellipse-N

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

3.5 GSS

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.

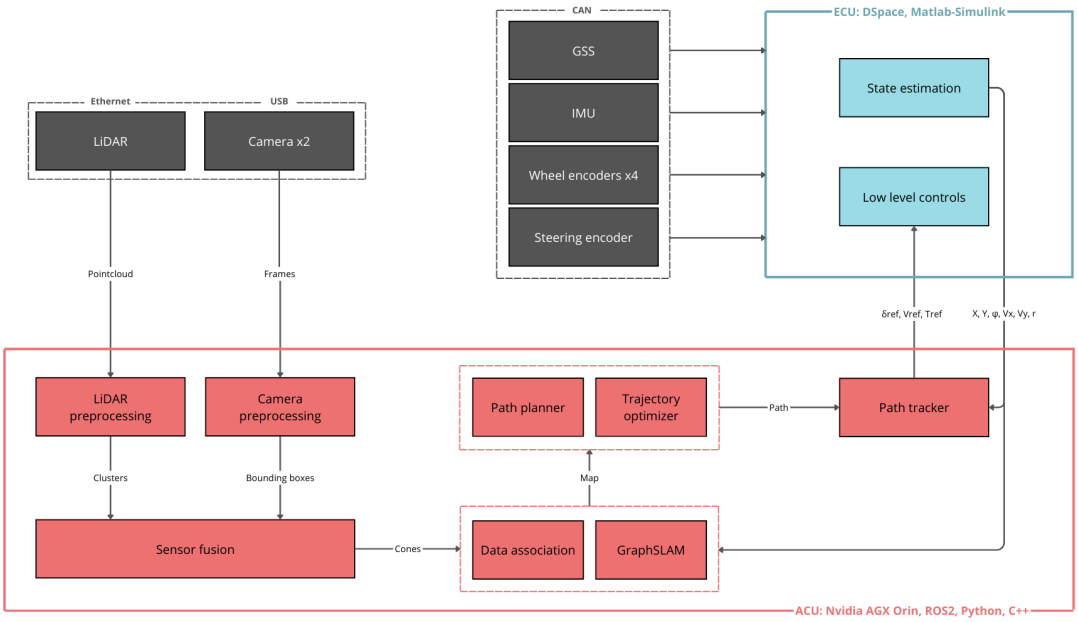

Figure 8: Autonomous system overview

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.

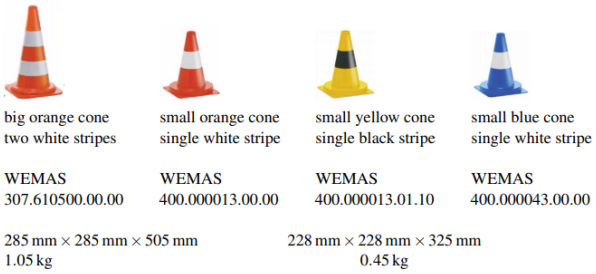

Figure 9: Cones rule specifications

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.

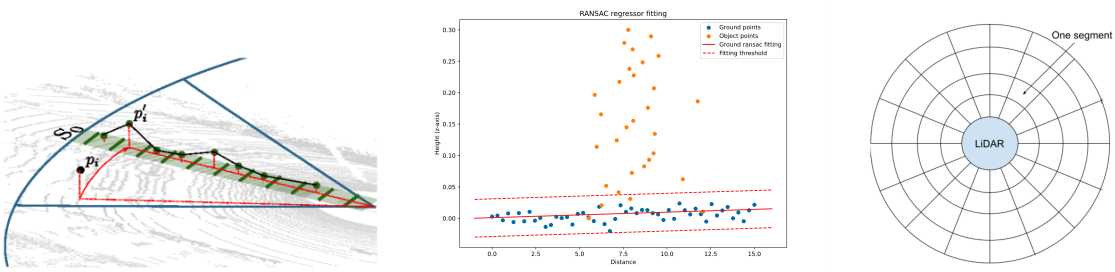

Figure 10: Ground Plane Removal

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.

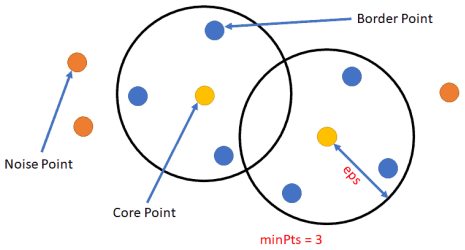

Figure 11: DBSCAN

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.

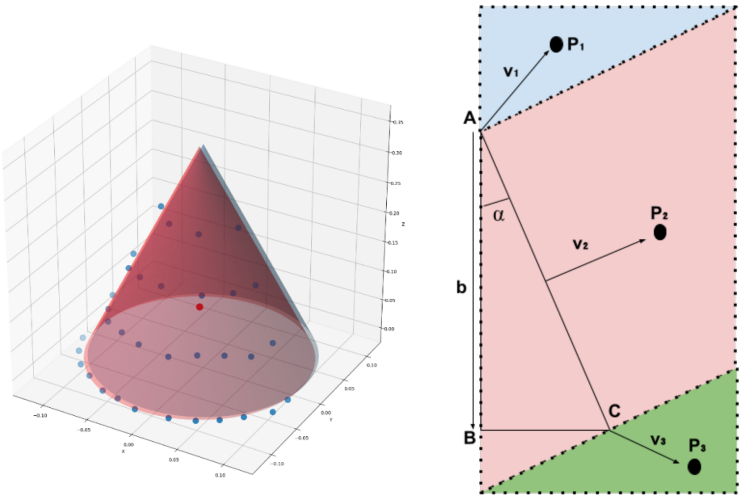

Figure 12: Shape Recognition

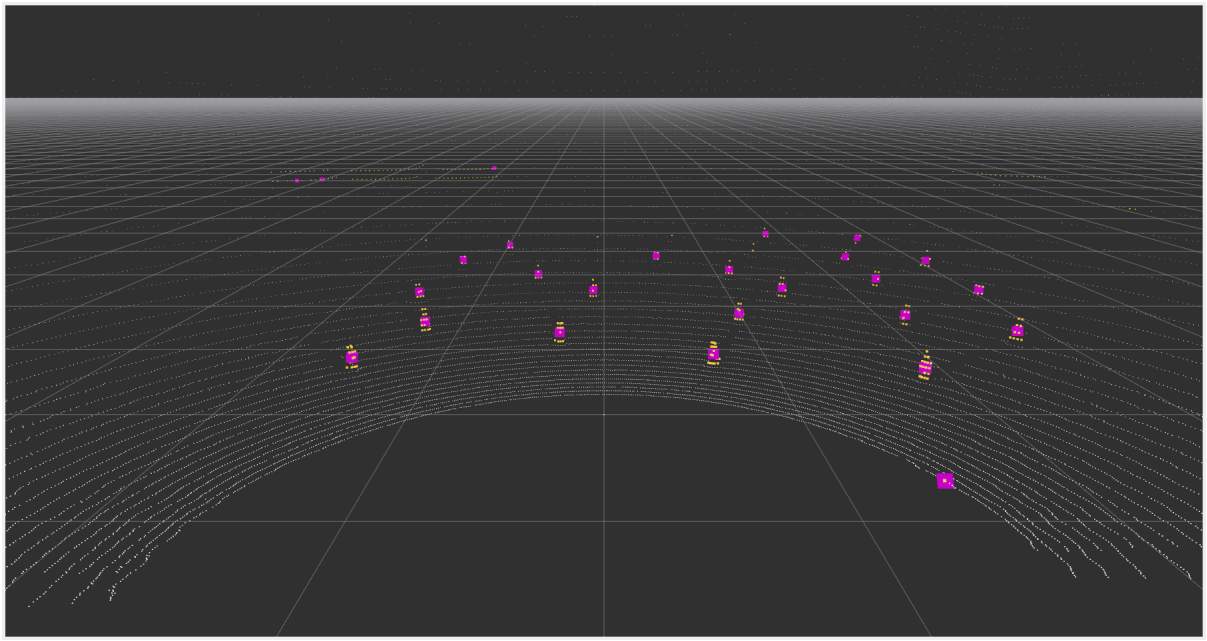

Figure 13: LiDAR preprocessing: point cloud in white, obstacle point cloud in yellow, centroids in violet

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 [2] and YOLOv10 [1] 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

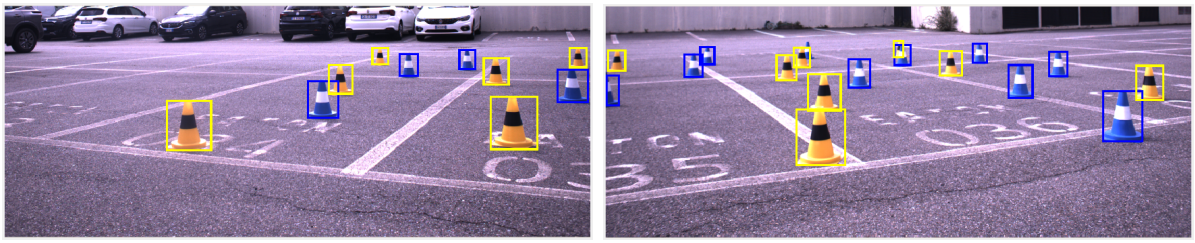

Figure 14: YOLO

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.

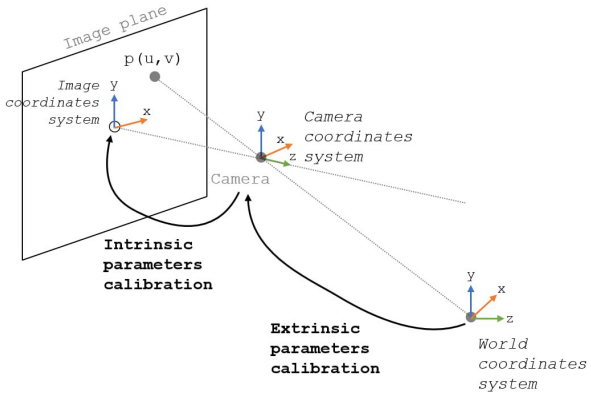

Figure 15: Extrinsic and intrinsic matrices

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.

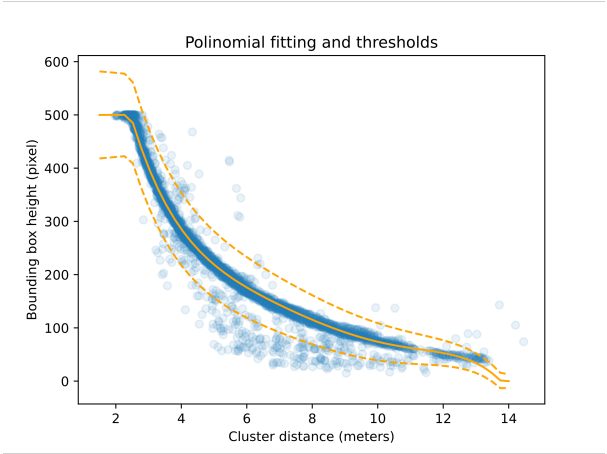

Figure 16: Polinomial fitting of distance to bounding box height function

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.

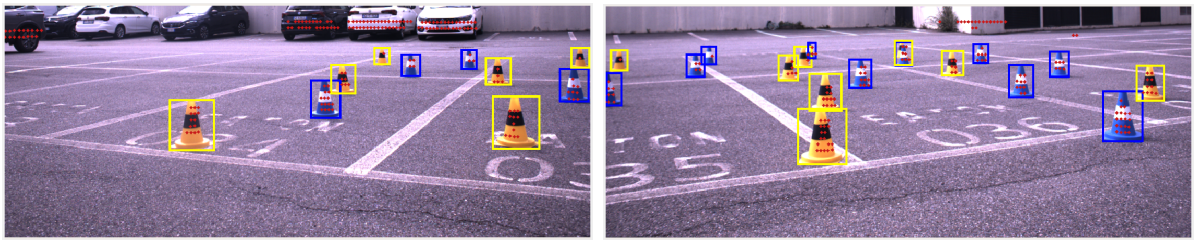

Figure 17: Sensor Fusion obstacle point cloud mapping onto processed images

4.2.5 Results

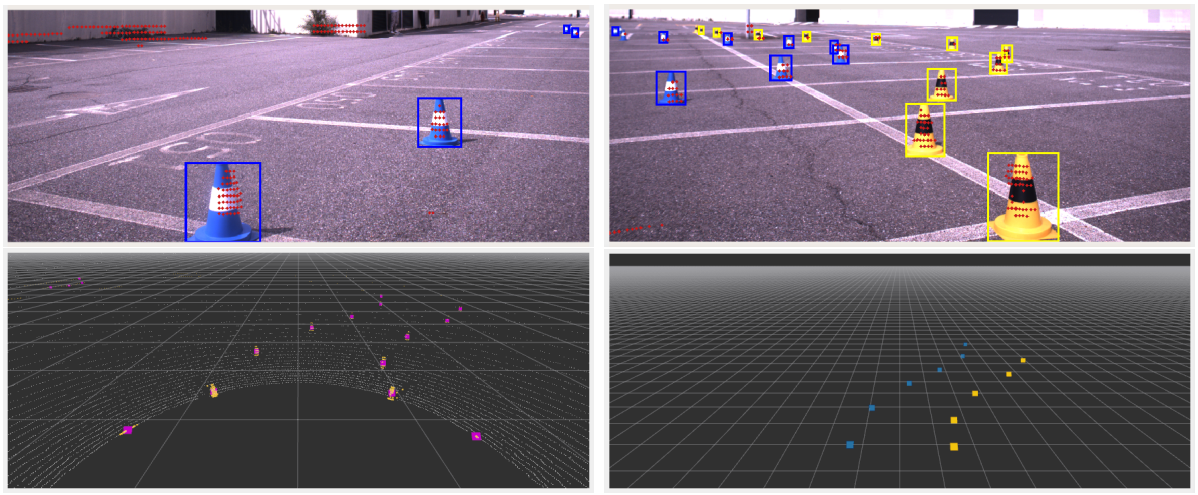

Figure 18: Perception track results

4.3 Simultaneus Localization And Mapping

4.3.1 Data Association

4.3.2 Optimization

4.3.3 Lap counter

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.

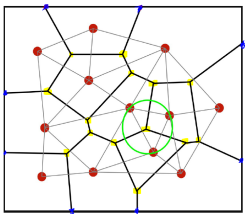

Figure 19: Delaunay triangulation and Voronoi diagram

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.

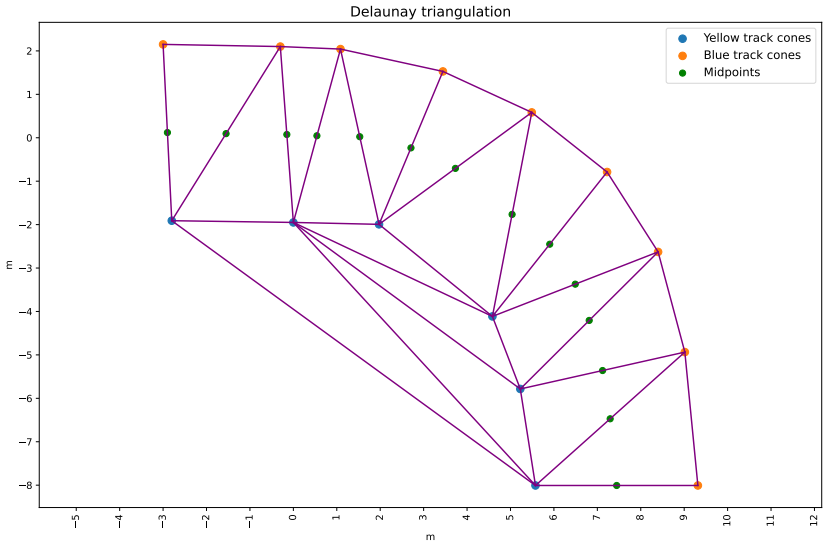

Figure 20: Delaunay waypoints

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.

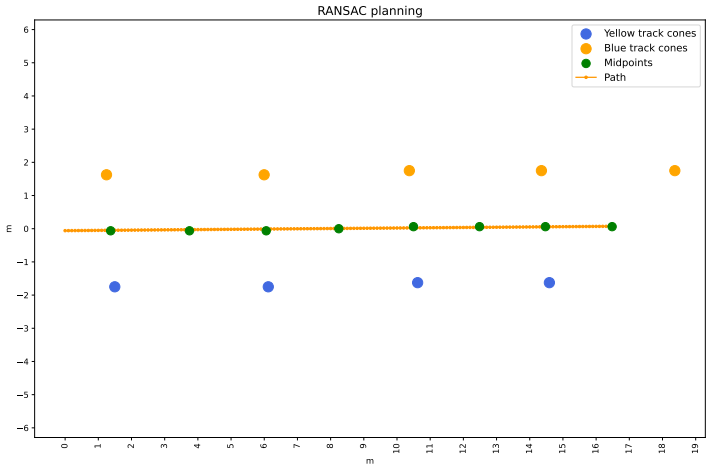

Figure 21: RANSAC path

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.

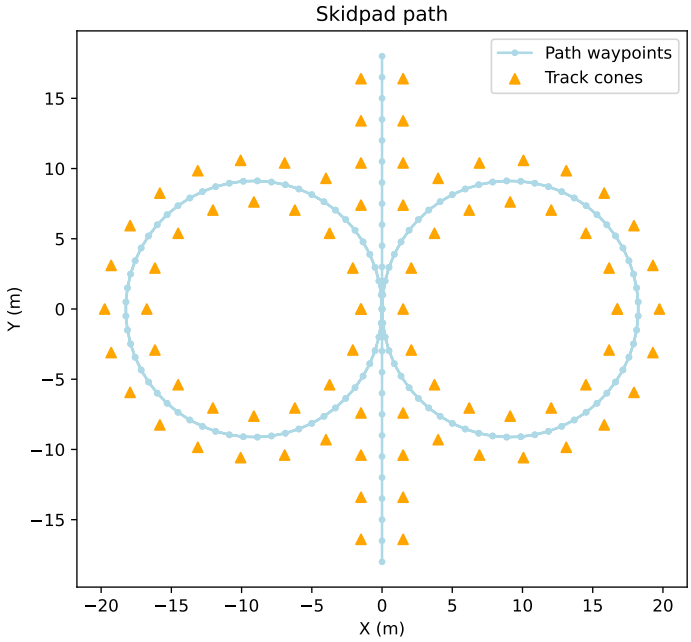

Figure 22: Skidpad path

4.4.3 Local Path Planning

RRT Planner

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.

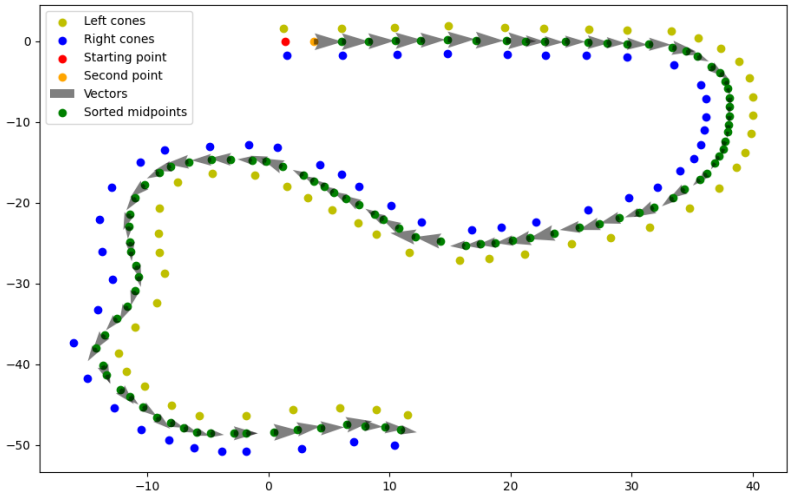

Figure 23: Path Searching

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.

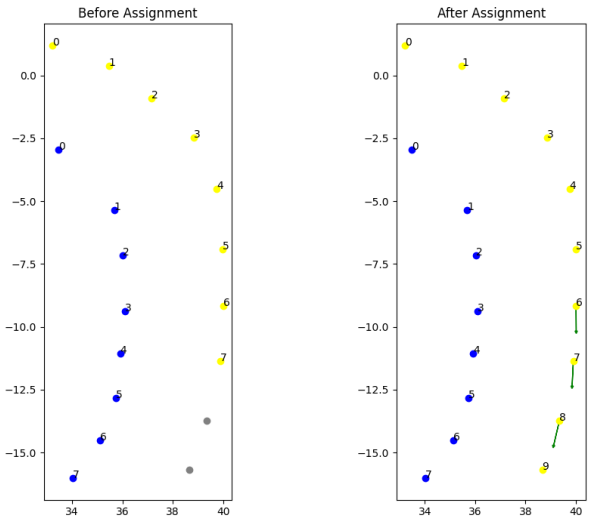

Figure 24: Unknown cones color assignment

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.

4.4.5 Trajectory Optimization

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:

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

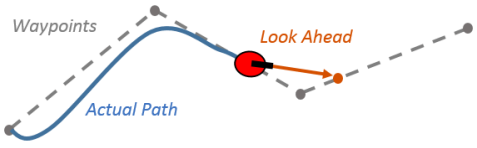

Figure 25: Pure Pursuit

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

4.5.4 Model Predictive Contouring Control

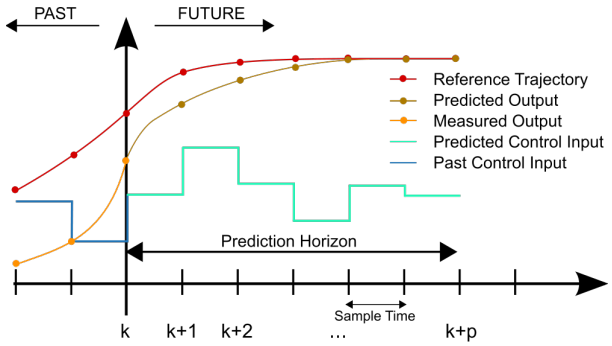

Figure 26: MPC receding horizon

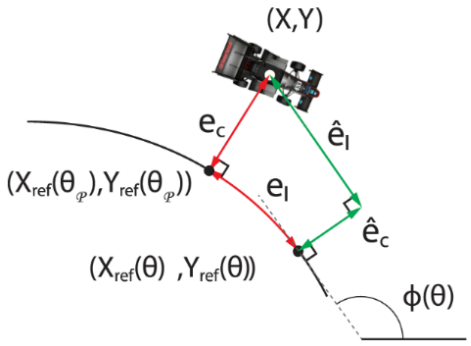

Figure 27: MPC contouring formulation

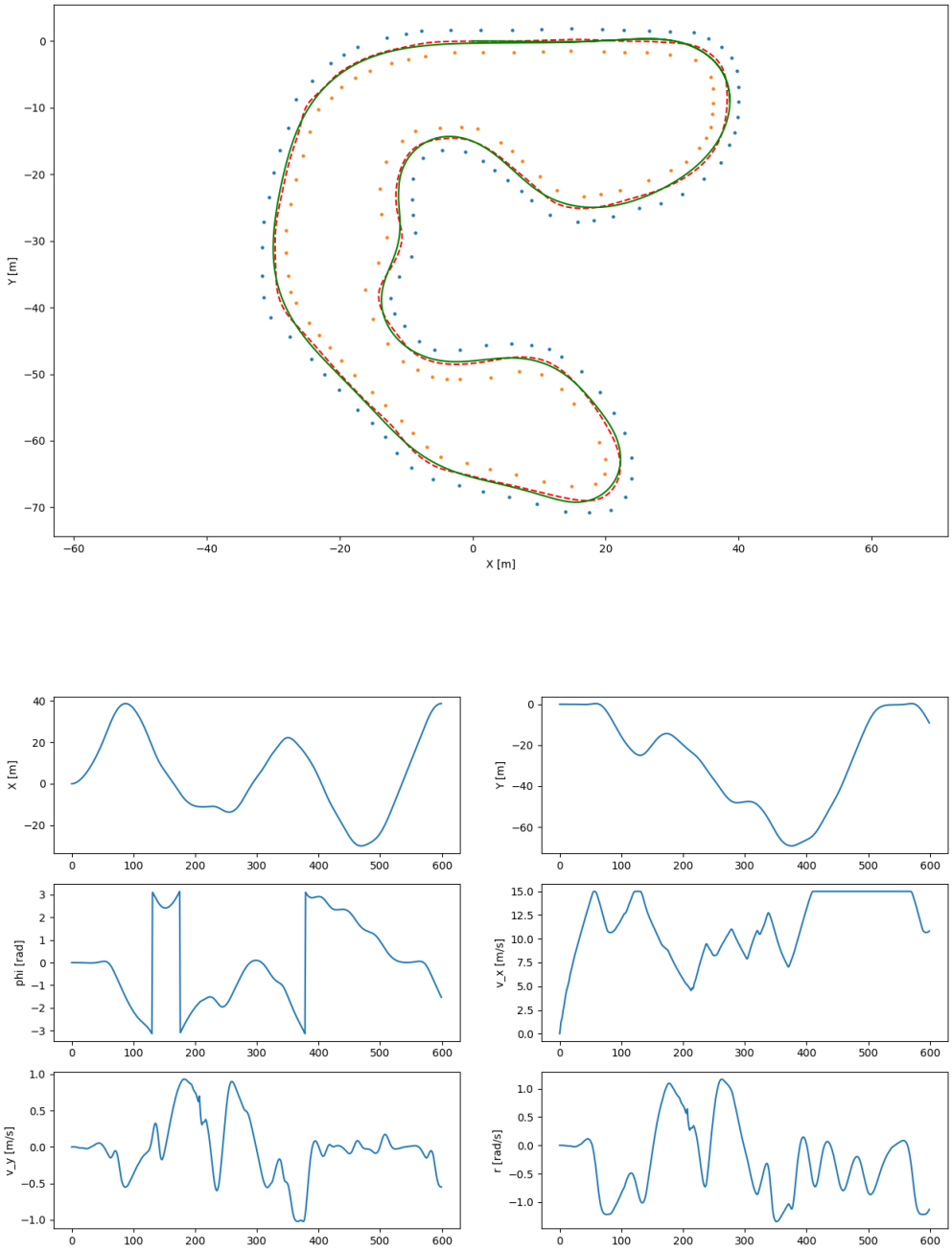

Figure 28: MPCC simulation results

5 Project configuration

5.1 ROS2 framework

Figure 29: 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 30: SCD Simulator

Figure 31: 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 Vehicle model

The vehicle model employed in the simulation 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.

The state vector is 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.

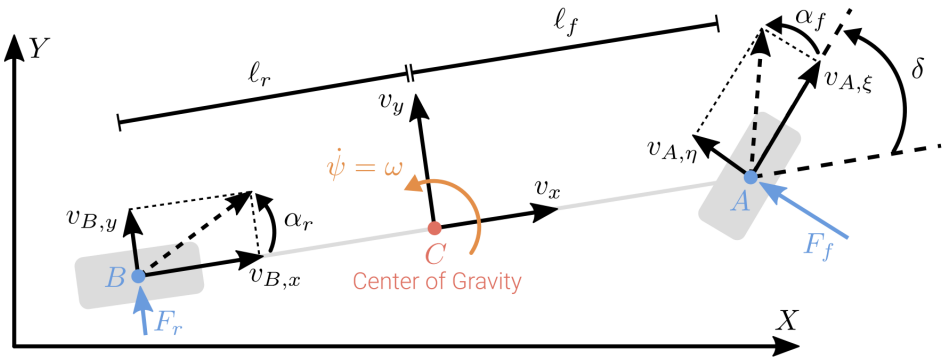

Figure 32: Bicycle model

To be replaced with a more complex solution to emulate rality gap.

6.3 Unity application

This simulator communicates with a Unity application via a TCP-based inter-process communication (TCP IPC) system, which is used to communicate in loopback with Unity application. The Unity application handle visualization, simulation of the environment and the simulation of sensor data (LiDAR and cameras). A percetion_simulation node allow also a simulation of path planning SLAM and path tracking algorithms, without perception simulation.

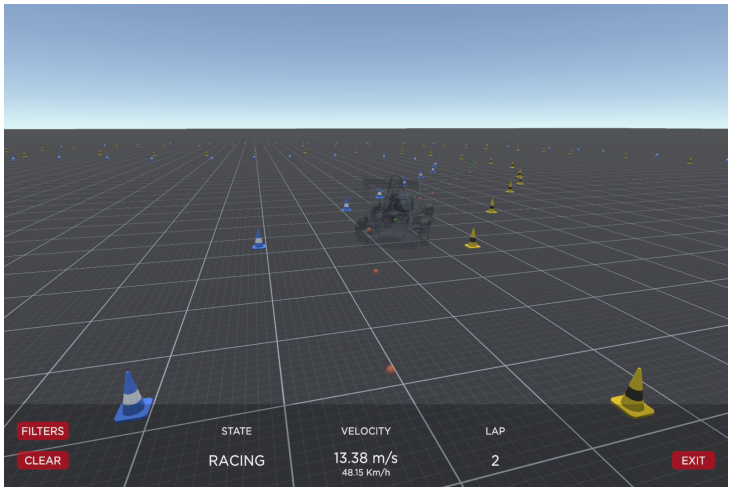

Figure 33: SCD Simulator interface

6.4 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.

6.4.1 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 $\phi_f = e^{2\pi i r}$, 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:

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

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.

6.4.2 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.

6.4.3 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.

6.4.4 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.

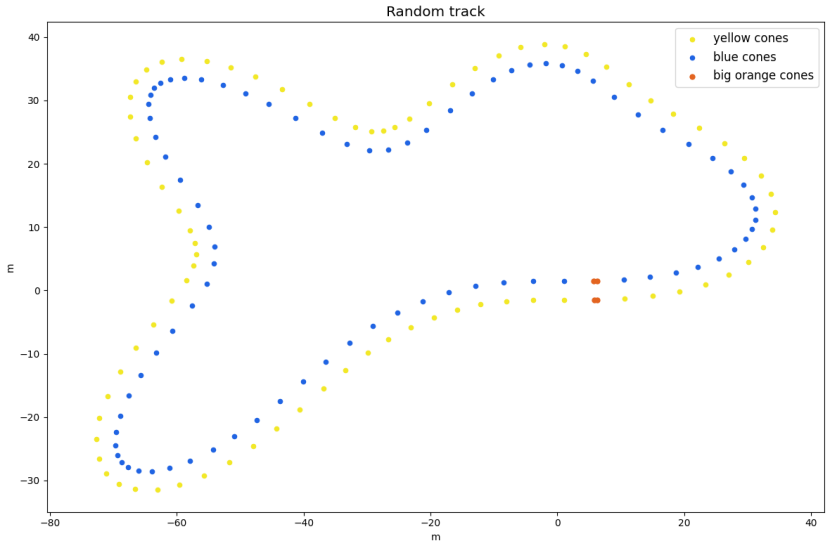

Figure 34: Random generated track

7 ROS2 Pipeline

Figure 35 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**: Finite state machine, logger, TCP inter-process communication, and visualization packages, which are accessible by all nodes.
- **Yellow**: Perception simulation and vehicle model packages, primarily used for simulation purposes.
- **Light blue**: ECU tasks, managing low-level vehicle control and state estimation.

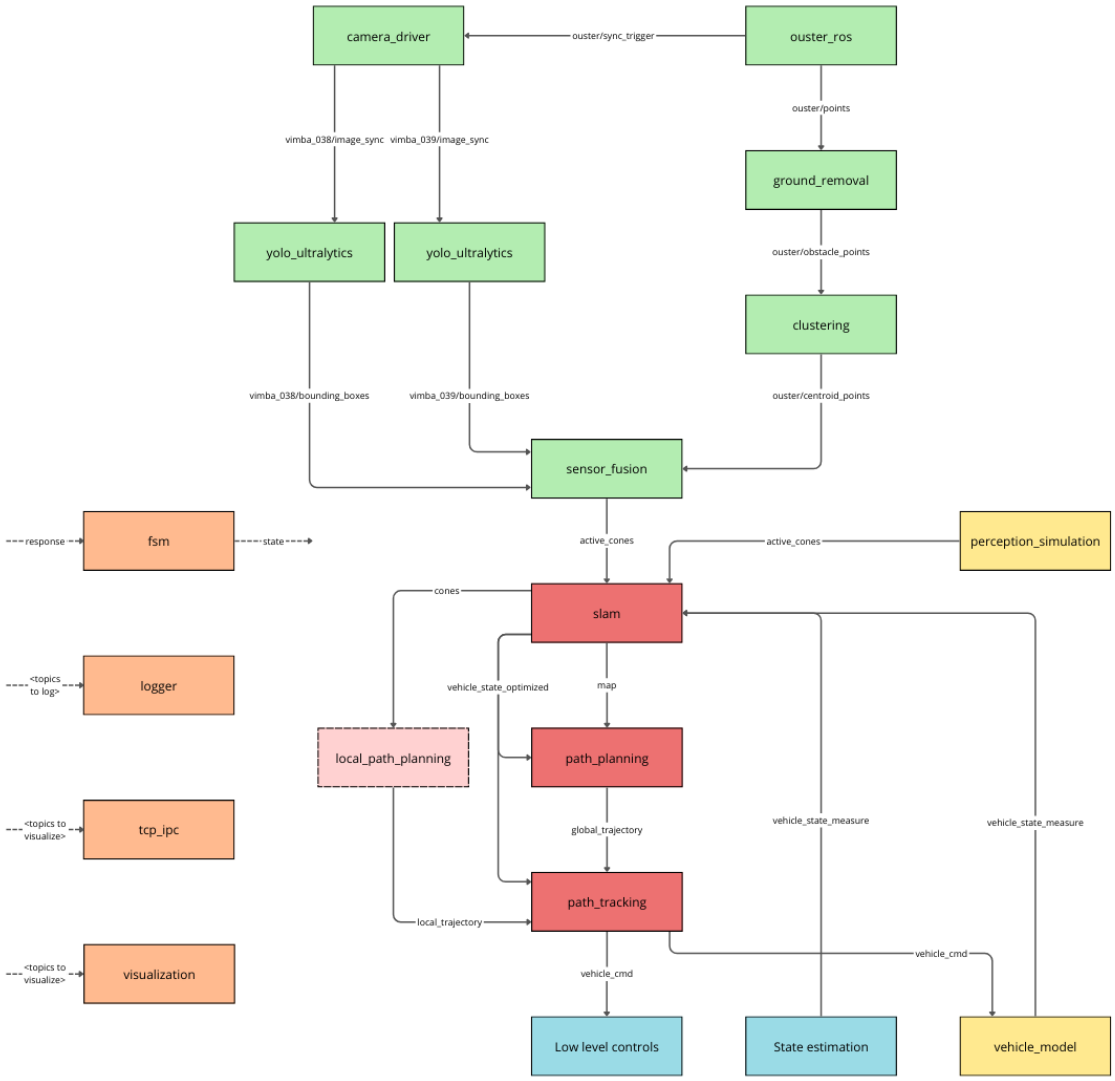

Figure 35: ROS2 packages pipeline

References

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

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

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