

Comparative Study of Delaunay Path Planner and RRT/RRT* for Path Planning in Autonomous Racing

Rodrigo Carrión Caro

March 30, 2025

Contents

Glossary	4
Abstract	6
Acknowledgements	7
1 Introduction	8
1.1 Background	8
1.2 Aim	8
1.3 Objectives	9
1.4 Product Overview	10
1.4.1 Scope	10
1.4.2 Audience	10
2 Background Review	11
2.1 Related Literature	11
3 Methodology	17
3.1 Research and Software Development Process	17
3.1.1 Application of Agile Methodology, aquí meto el diagrama de Agile . .	17
3.2 Development Phases	17
4 Technology	19
4.1 Implementation Tools and Resources	19
4.2 Workspace	20
4.2.1 OBRA WS	20
4.2.2 ADS-wsgraph	21
4.2.3 Perception	22
4.2.4 Localisation & Mapping	23

4.2.5	Path Planning	24
4.2.6	Control	25
4.2.7	Hardware and Sensors	26
4.3	Simulation and Testing Tools	27
5	Code Development	28
5.1	Path Planning Folder Structure	28
5.1.1	Local Planner (local_planner.py)	28
5.1.2	State Management (new_states.py)	29
5.1.3	Utility Functions (utils.py)	30
5.2	Initial Path Planner: Old Delaunay Implementation	31
5.3	Delaunay Path Planner Optimization and Rewriting	31
5.3.1	Structural Simplification	31
5.3.2	Efficient Midpoint Calculation	32
5.3.3	Enhanced Cone Clustering for Smoother Paths	33
5.3.4	Comparison of Old vs. New Implementation	33
5.4	Development of RRT*	34
5.4.1	Motivation for Switching from Delaunay to RRT*	34
5.4.2	RRT vs. RRT*: Key Differences	35
5.4.3	Initial RRT* Planner: Centerline Trajectory	36
	Node Expansion and Distance Selection	37
	Collision Detection and Safety Constraints	37
	Path Retracing and Smoothing	38
5.4.4	Final RRT* Planner: Fastest Trajectory	38
5.4.5	Final Considerations	40
6	Version Management	42
6.1	Source Code and Data Management	42
6.1.1	GitLab Repository Structure (Aquí puedes incluir una imagen del es- quema del repositorio)	43
6.1.2	Source Code Repository Link	43
7	Results	44
7.1	Results and Testing	44
7.1.1	Simulation Setup	44
7.1.2	Test Scenarios	45

7.2	Performance Metrics	46
7.2.1	Computation Time Evaluation	46
7.2.2	Path Quality Analysis	47
7.2.3	Adaptability to Dynamic Obstacles	48
7.3	Experimental Results	48
7.3.1	Visualization of Results Through Graphs	49
7.3.2	Code Efficiency and Optimization Metrics	52
8	Professionalism	54
8.1	Project Management	54
8.1.1	Development Activities and Schedule	54
8.1.2	Data Management	54
8.1.3	Project Deliverables	54
8.2	Risk Analysis	54
8.2.1	Identified Risks and Mitigation Strategies	54
8.2.2	Updated Project Plan Based on Risk Evaluation	54
8.3	Legal, Ethical, and Environmental Considerations	54
8.3.1	Compliance with Professional Codes of Conduct	54
8.3.2	Ethical and Environmental Impact of the Project	54
9	Conclusion	55
9.1	Summary of Findings	55
9.1.1	Key Insights from the Algorithm Comparison	55
9.2	Future Work	55
9.2.1	Improvements in RRT* Implementation	55
9.2.2	Real-World Applications	55
10	Bibliography	56
11	Appendices	58
11.1	Supplementary Data	58
11.1.1	Source Code Repository (GitHub/GitLab)	58
11.1.2	ROS2 + Unity Configuration Details	58
11.1.3	Raw Simulation Results	58

Glossary

- **RRT** - Rapidly-Exploring Random Tree, algoritmo de planificación de rutas que explora espacios grandes de forma eficiente.
- **ROS2** - Robot Operating System 2, framework modular para el desarrollo de software robótico.
- **Unity** - Motor de simulación utilizado para replicar condiciones reales de conducción en entornos virtuales.
- **Path Planning** - Planificación de rutas, proceso de calcular trayectorias óptimas evitando obstáculos.
- **YOLO** - You Only Look Once, arquitectura de deep learning para detección de objetos en tiempo real.
- **mAP** - Mean Average Precision, métrica para evaluar la precisión de un modelo de detección.
- **IoU** - Intersection over Union, métrica que mide la superposición entre predicciones y etiquetas reales.
- **Inference Time** - Tiempo que tarda el modelo en procesar una imagen y devolver resultados.
- **Dataset** - Conjunto de datos utilizado para entrenar, validar y testear modelos de machine learning.
- **Annotation** - Proceso de etiquetar imágenes con información relevante (e.g., posición y tipo de objetos).
- **Roboflow** - Herramienta en línea utilizada para anotar y gestionar datasets de visión por computador.

- **YOLOv5, YOLOv8, YOLOv11** - Versiones específicas del modelo YOLO con diferentes arquitecturas y mejoras.
- **mAP@50, mAP@50-95** - Variantes de mAP que evalúan la precisión a diferentes umbrales de IoU.
- **Precision** - Proporción de verdaderos positivos frente al total de predicciones positivas.
- **Recall** - Proporción de verdaderos positivos frente al total de elementos reales positivos.
- **Epoch** - Iteración completa sobre el dataset durante el proceso de entrenamiento de un modelo.
- **Overfitting** - Problema de generalización en el que un modelo aprende demasiado los datos de entrenamiento.
- **Data Augmentation** - Técnicas utilizadas para aumentar la variabilidad del dataset mediante transformaciones.
- **Simulator** - Entorno virtual donde se prueban los modelos de detección de forma controlada.
- **OBRA** - Oxford Brookes Racing Autonomous, equipo universitario de vehículos autónomos de competición.

Abstract

0.1 Contexto

Escribe aquí el contexto del problema.

0.2 Objetivo

Escribe aquí la descripción clara del propósito del proyecto.

0.3 Metodología

Describe el enfoque de desarrollo y herramientas utilizadas.

0.4 Resultados principales

Explica los principales hallazgos y comparación entre métodos.

0.5 Conclusión

Menciona el impacto del estudio y posibles mejoras.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, **Muhammad Younas**, for his guidance, support, and continuous feedback throughout this project. His insights were fundamental in shaping the structure and direction of the research.

My heartfelt thanks also go to the entire **Oxford Brookes Racing Autonomous (OBRA)** team for creating an environment of collaboration and technical excellence. In particular, I am especially grateful to **Aduén Benjumea** and **Mihir Gohad**, who served as project leads and played a crucial role in defining the vision and objectives of the team throughout the year.

I would also like to thank **Sebastian Donnelly**, whose deep experience within OBRA was invaluable—always available to solve doubts and provide practical advice during key stages of development.

A special mention goes to **Dorian Amaritei**, who supported the development and optimization of the Delaunay path planner and contributed significantly to its refinement.

Finally, I am very thankful to **Pablo Gutiérrez**, lead of the simulation subteam, who assisted me in running and validating the simulations used to compare the planners. His help ensured that the performance of each algorithm could be observed under realistic and controlled scenarios.

This project would not have been possible without the collaborative effort and technical excellence of everyone involved.

1. Introduction

1.1 Background

In autonomous vehicles, path planning is essential, as it allows for efficient and safe route planning in real time. Specifically, the RRT algorithm “RRT Rapidly exploring Random Tree” can explore large spaces efficiently. This makes it ideal for dynamic environments, such as an autonomous car race. These characteristics make it widely used in the field of robotics. The RRT (Rapidly exploring Random Tree) algorithm is a search method used to efficiently find paths in large spaces [1]. It is particularly useful in dynamic environments where viable paths need to be found quickly in real-time. Currently, the Oxford Brookes Racing Autonomous (OBRA) team uses a neural network path planner. A Delaunay path planner is being developed but has not yet been implemented. These approaches present some limitations in real-time situations. The development and implementation of an RRT [2] will allow for greater flexibility and adaptability. Ideally, this will improve the team’s path planning capabilities. In addition to developing the RRT, this project will compare its performance with the current Delaunay path planner. By evaluating their efficiency and adaptability in dynamic scenarios, the goal is to determine which algorithm provides better results for autonomous racing.

1.2 Aim

The aim of this project is to develop and optimize two advanced path planning algorithms for autonomous racing: an improved Delaunay-based planner and an optimized RRT* algorithm. These planners will be designed to enhance adaptability, computational efficiency, and trajectory optimization in high-speed, dynamic environments.

This project also aims to conduct a detailed comparative analysis of both algorithms, evaluating their performance across key metrics such as computation time, path efficiency, adaptability to dynamic obstacles, and robustness under racing conditions.

By implementing and testing these algorithms within a ROS2-based simulation environment using Unity, this research seeks to identify the most effective path planning solution for the OBRA team's autonomous racing vehicle. The insights gained from this study will contribute to both the OBRA competition strategy and the broader field of autonomous vehicle navigation.

1.3 Objectives

- To conduct a comprehensive study of path planning techniques used in autonomous vehicle navigation, analyzing their advantages, limitations, and applications in high-speed racing scenarios. [3]
- To develop and implement an improved Delaunay-based path planner that enhances adaptability, computational efficiency, and trajectory smoothness in dynamic environments.
- To develop and implement an RRT algorithm from scratch for real-time path planning in autonomous vehicles, ensuring compatibility with the OBRA car's ROS2 framework.
- To optimize the RRT algorithm by integrating RRT* [4], improving its ability to generate efficient and dynamically adaptable routes.
- To validate both the optimized Delaunay and RRT* algorithms in a controlled simulation environment using Unity, testing their performance under varying racing conditions.
- To conduct a detailed comparative analysis of the optimized Delaunay and RRT* algorithms, evaluating key performance metrics such as computation time, path efficiency, adaptability to dynamic obstacles, and robustness in high-speed scenarios.
- To identify and address potential limitations of each algorithm, proposing refinements or hybrid approaches that could further enhance their performance.
- To integrate the most effective path planner into the OBRA team's autonomous racing pipeline, ensuring real-world applicability and alignment with competition requirements.

1.4 Product Overview

1.4.1 Scope

The objective of this project is to develop and optimize two new path planners for autonomous racing:

- **Optimized Delaunay Path Planner** – A modified version of the traditional Delaunay-based planner, improving its efficiency and adaptability for dynamic racing environments.
- **RRT*** – An enhanced Rapidly-exploring Random Tree algorithm that generates smoother and more efficient paths by reducing randomness and refining route selection.

Both planners will be developed in Python, ensuring seamless integration with the ROS2 framework used in the OBRA autonomous car. The testing and validation process will be conducted in simulation environments using Unity, allowing for extensive evaluation before potential real-world implementation. Once developed, these two new planners will be compared to determine which provides better performance in terms of adaptability, computational efficiency, and trajectory smoothness under high-speed, dynamic conditions.

1.4.2 Audience

The primary audience for this study is the Oxford Brookes Racing Autonomous (OBRA) team, as the improved path planning algorithms will directly contribute to their autonomous racing performance. Additionally, this research is relevant to the academic community, particularly in robotics, AI, and autonomous vehicle navigation, by providing insights into optimization strategies for real-time path planning. From an industry perspective, this study holds significance for autonomous systems professionals, particularly those developing path planning solutions for high-speed and dynamic environments, such as self-driving vehicles, robotics, and UAV navigation.

2. Background Review

2.1 Related Literature

Reference	Zhao, H., Wu, Z., Li, Y., & Wang, J. (2021) 'Improved Bidirectional RRT* Path Planning Method for Smart Vehicle', Mathematical Problems in Engineering, pp. 1-14. doi: 10.1155/2021/6669728.
Title	Improved Bidirectional RRT* Path Planning Method for Smart Vehicle
Summary	The study proposes an improvement to the bidirectional RRT* algorithm to optimize route planning in intelligent vehicles, aiming for shorter and more efficient routes.
Evaluation	The approach is useful for static environments, but it does not sufficiently address the challenges in dynamic environments, which may limit its applicability in competitive vehicles.
Reflection	The proposed improvements could be applied to optimize the RRT* algorithm in my project, particularly in reducing computation time and optimizing routes.
Main Themes	RRT* optimization, route planning, autonomous vehicles.

Table 2.1: Summary of Zhao et al. (2021)

Reference	Gasparetto, A., Boscariol, P., Lanzutti, A., & Vidoni, R. (2015) 'Path Planning and Trajectory Planning Algorithms: A General Overview', Journal of Intelligent & Robotic Systems, pp. 1-33. doi: 10.1007/978-3-319-14705-5_1.
Title	Path Planning and Trajectory Planning Algorithms: A General Overview
Summary	The article provides an overview of the main trajectory and route planning algorithms in robotics. It analyses methods such as Roadmap, Cell Decomposition, and RRT, as well as their applications in industrial and autonomous environments.
Evaluation	The study offers a comprehensive overview of the algorithms but focuses more on static industrial systems, limiting its applicability to dynamic environments. However, the review of RRT is useful for improving my project.
Reflection	This article will be key to contextualizing my work, as it provides a solid foundation on traditional methods and suggests possible areas for improvement, such as applying them in more dynamic environments.
Main Themes	Route planning, RRT, optimization algorithms, autonomous robotics.

Table 2.2: Summary of Gasparetto et al. (2015)

Reference	Wang, H., Li, G., Hou, J., Chen, L., & Hu, N. (2022) 'A Path Planning Method for Underground Intelligent Vehicles Based on an Improved RRT* Algorithm,' Electronics, vol. 11, no. 3, p. 294. doi: 10.3390/electronics11030294.
Title	A Path Planning Method for Underground Intelligent Vehicles Based on an Improved RRT* Algorithm
Summary	The study proposes an improved RRT* method for route planning in underground intelligent vehicles, adjusting the dynamic step size and turn angle constraints.
Evaluation	The approach is innovative for underground environments and offers improvements in efficiency and safety, but it is limited to controlled spaces and does not address navigation in fully dynamic environments.
Reflection	This study is relevant to my project, as the proposed RRT* improvements could be applied to optimize the algorithm in more complex scenarios, such as autonomous racing.
Main Themes	RRT*, underground autonomous vehicles, route planning, optimization.

Table 2.3: Summary of Wang et al. (2022)

Reference	Sánchez-Ibáñez, J.R., Pérez-del-Pulgar, C.J., & García-Cerezo, A. (2021) 'Path Planning for Autonomous Mobile Robots: A Review', <i>Sensors</i> 2021, 21, 7898. doi: 10.3390/s21237898.
Title	Path Planning for Autonomous Mobile Robots: A Review.
Summary	The article reviews route planning algorithms for mobile robots, focusing on their classification and applicability in autonomous environments.
Evaluation	It provides a very useful overview of the main approaches, but it focuses on controlled scenarios and may be limited for dynamic environments such as competitions.
Reflection	This article provides a good foundation for comparing different approaches, which will help me justify the choice of RRT in my project.
Main Themes	Route planning, mobile robots, path search algorithms.

Table 2.4: Summary of Sánchez-Ibáñez et al. (2021)

Reference	C. Messer, A. T. Mathew, N. Mladenovic and F. Renda, "CTR DaPP: A Python Application for Design and Path Planning of Variable-strain Concentric Tube Robots," 2022 IEEE 5th International Conference on Soft Robotics (RoboSoft), Edinburgh, United Kingdom, 2022, pp. 14-20, doi: 10.1109/RoboSoft54090.2022.9762088.
Title	CTR DaPP: A Python Application for Design and Path Planning of Variable-strain Concentric Tube Robots.
Summary	The study presents a modular platform in Python that uses the RRT* algorithm for route planning and design optimization in concentric tube robots. It focuses on trajectory planning in environments with torsion and curvature constraints.
Evaluation	The implementation in Python is relevant to my project, as it allows for the flexible use of planning and optimization algorithms.
Reflection	This article supports the use of Python in my project, demonstrating that it is an effective option for developing and testing algorithms like RRT*.
Main Themes	RRT*, route planning, concentric tube robots, design optimization, use of Python.

Table 2.5: Summary of Messer et al. (2022)

Reference	Kolski, S., Ferguson, D., Stachniss, C., & Siegwart, R. (2006) 'Autonomous Driving in Dynamic Environments', Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1-10. doi: 10.3929/ethz-a-010079481.
Title	Autonomous Driving in Dynamic Environments.
Summary	The study presents a hybrid autonomous navigation system that operates in both structured and unstructured environments, handling dynamic obstacles like pedestrians and other vehicles.
Evaluation	Unlike many approaches focused on static environments, this system is highly relevant for dynamic settings, such as autonomous car competitions, where real-time adjustments to moving obstacles are critical.
Reflection	This study is essential for my project as it highlights the importance of dynamic environments and provides useful insights for improving my route planning system.
Main Themes	Autonomous navigation, dynamic environments, route planning, moving obstacles, autonomous vehicles.

Table 2.6: Summary of Kolski et al. (2006)

3. Methodology

3.1 Research and Software Development Process

3.1.1 Application of Agile Methodology, aqui meto el diagrama de Agile

The project follows an Agile methodology, allowing for iterative progress and continuous refinement of the algorithms. This approach ensures flexibility and adaptability to changing requirements. Agile was chosen over traditional models such as Waterfall, as the latter requires a rigid, sequential structure that does not accommodate modifications once development begins.

Given the dynamic nature of this project, the Agile framework supports the parallel development of:

- Optimization and rewriting of the existing **Delaunay path planner**.
- Development of a new **RRT* algorithm** from scratch.
- Comparative testing and analysis of both approaches.

The project is structured into multiple iterations, ensuring continuous validation and improvement of the implemented methods.

3.2 Development Phases

The preparation phase involved selecting the topic and conducting meetings with OBRA to align the project's objectives. A proposal was submitted and refined based on feedback, followed by setting up the required system and environment, including Unity and ROS2. An initial progress report was submitted to document early findings, and research was conducted to establish a baseline for optimising the Delaunay path planner.

The first iteration focused on enhancing the Delaunay-based path planner by rewriting the algorithm to improve its efficiency and adaptability. An optimised version was implemented and tested in a simulation environment to validate improvements in path generation.

The results were then demonstrated before proceeding with the development of the RRT algorithm.

In the second iteration, work began on the RRT algorithm by conducting a theoretical study to understand its principles and limitations. This was followed by research and design for a basic implementation, which was then developed and tested in a simulated environment to compare its performance with the Delaunay approach.

Once the basic RRT implementation was completed, the third iteration concentrated on optimising it using RRT*. This involved refining the algorithm to enhance route efficiency and conducting a comparative analysis between Delaunay and RRT* under racing conditions to assess their performance.

Following the comparative tests, the project progressed to the final validation and integration phase. The selected algorithm was integrated into the OBRA car to ensure its applicability in real-world racing scenarios.

The final stage of the project involved preparing a comprehensive report summarising the findings, results, and technical contributions. Additionally, a final presentation was developed to communicate the project's outcomes and submit the final deliverables.

4. Technology

4.1 Implementation Tools and Resources

The development of this project relies on several key technologies and tools to ensure efficient implementation and testing. The primary programming language used is Python, which serves as the backbone of the implementation. Python is used to develop the path-planning algorithms, integrate them into ROS2, and manage the necessary data processing tasks. Its extensive libraries and compatibility with machine learning and robotic frameworks make it ideal for rapid prototyping and testing.

For robotic integration, we utilize ROS2 (Humble), which facilitates seamless communication between different components of the autonomous system. ROS2 manages sensor data processing, real-time control, and path planning, ensuring that all modules operate in a synchronized manner. The middleware's efficiency in handling multiple nodes is crucial for real-time decision-making within the autonomous vehicle.

The project runs on Ubuntu 22.04, which provides a stable and well-supported environment for ROS2, Python-based applications, and simulation tools. Ubuntu ensures compatibility with the robotic frameworks used in this project and offers an extensive community for troubleshooting and optimization.

For version control, Git and GitLab serve as the central repository for managing code changes. The OBRA workspace on GitLab stores previous developments and is used to track new implementations through branches. Frequent commits help maintain a structured development workflow.

Simulation and testing are conducted using a structured environment that combines Unity for scenario modeling and Foxglove for real-time visualization of vehicle states and planned paths. The simulation outputs, along with test results, are stored in Google Drive, ensuring efficient data management and accessibility for analysis.

To maintain an organized workflow, Notion is used for task management and Agile planning, ensuring that development is well-structured and efficiently executed.

By integrating these technologies, we aim to develop a robust and scalable path-planning solution that seamlessly integrates with the OBRA autonomous vehicle.

4.2 Workspace

4.2.1 OBRA WS

The Oxford Brookes Racing Autonomous (OBRA) workspace is organised around a modular system architecture, where each major component of the autonomous pipeline is developed by a dedicated subteam. This structure enables parallel development while ensuring each module integrates smoothly into the overall system. Figure 4.1 provides an overview of the autonomous stack, highlighting the data flow between modules and their interaction with both real-world and simulated environments.

The system is divided into key subsystems: Sensors Suite, Perception, Localisation & Mapping, Trajectory Planning, and Controls. Each subteam is responsible for the development, testing, and integration of one of these subsystems, working collaboratively through shared interfaces and simulation environments. This architecture not only supports scalability and maintainability but also reflects the team's interdisciplinary nature, combining expertise in computer vision, robotics, AI, and control systems.

Development and testing are carried out in both real and simulated environments. The simulation setup replicates the vehicle's behaviour under various track conditions, providing a safe and flexible way to validate improvements before deploying them on the physical car.

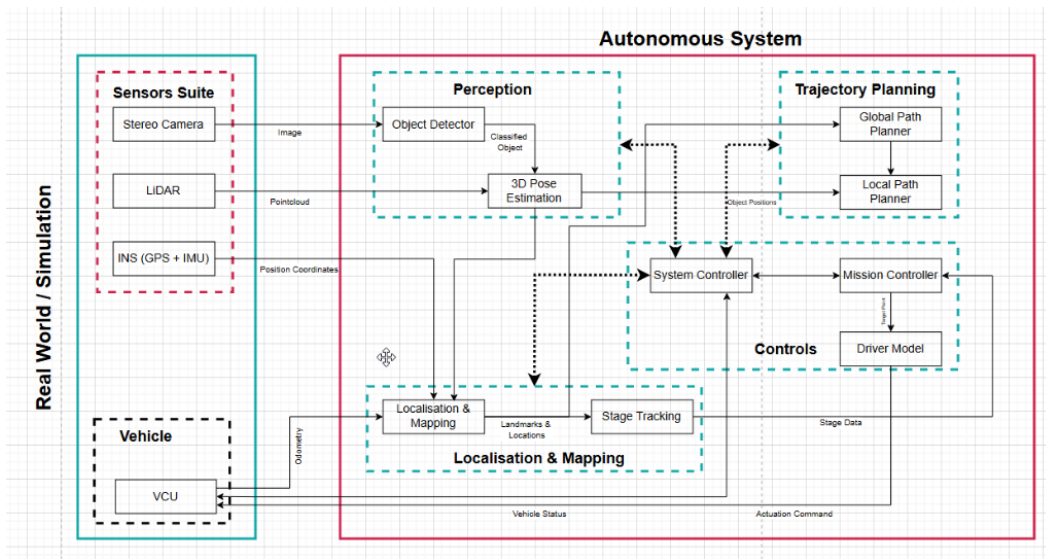


Figure 4.1: Autonomous System Architecture — OBRA workspace modules and interfaces.

4.2.2 ADS-wsgraph

To visualize the structure of the OBRA ecosystem, we can execute the following command in the Ubuntu terminal:

```
rqt_graph
```

This command generates a graphical representation of the ROS2 nodes and topics in execution. To obtain a visualization similar to the one shown in Figure 4.2, it is necessary to have the simulator and the workspace running and properly configured.

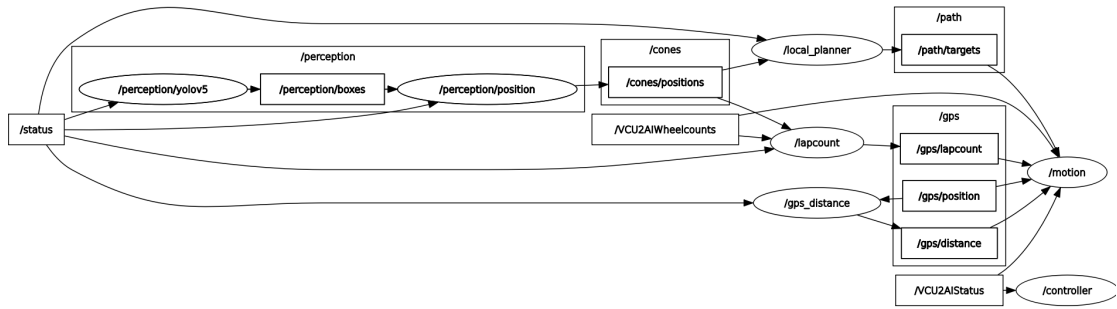


Figure 4.2: Ecosystem of nodes and topics in OBRA generated with `rqt_graph`

The OBRA workspace consists of multiple nodes that interact through ROS2 topics, enabling communication between different modules of the autonomous system. In the graph generated by `rqt_graph`, we can observe how the different components are organized and communicate.

The `/status` node provides system information and connects with other key modules. Within the perception subsystem, the `/perception/yolov5` node processes images and generates bounding boxes, which are sent to `/perception/boxes` and subsequently converted into positions through `/perception/position`. This information is used by other modules such as path planning and localization.

The cone detection module processes position data through the `/cones/positions` topic, integrating this data into route planning via `/local_planner` and lap counting via `/lapcount`. Additionally, the GPS module handles localization data through `/gps/lapcount`, `/gps/position`, and `/gps/distance`, providing essential information for autonomous navigation.

Finally, the control module processes all this information and transmits it to `/motion`, enabling movement execution in the autonomous vehicle. The communication between these

modules ensures the proper operation of the system in a simulation environment or in the real vehicle.

The following subsections describe the various system modules in detail:

4.2.3 Perception

Perception is the entry point of the autonomous system. Computer vision is used to process images from our stereo camera, the Zed 2i, and the Livox HAP LiDAR. The detection of cones is handled through the YOLO neural network which takes images from the left channel of the camera to infer the position of cones on a 2D map. Depth perception from the camera is handled by taking into account the bounding boxes from both camera channels and comparing the two received frames to estimate a distance to an object getting a midpoint for them to estimate distance. The most suitable base suitable model from Ultralytics was YOLOv11 Nano [1], due to having the fastest inference speed of the offered base models. Detection of small objects such as cones at a distance is challenging due to the resolution limitations of the camera. This led to modifications on the base YOLOv11 neural network to increase its accuracy to detect small objects, making use of the YOLO-Z [2] approach.

To train the model to detect the four cone types, a dataset with manually labelled and gathered data was constructed. A dataset such as this, had to be varied to cover all possible light conditions that could possibly happen at competition. The data was gathered from past runs at Silverstone and testing, and stitched together to create a dataset with variable lighting conditions, locations and weather to be ready for all of the detection challenges presented on a fast moving vehicle. The model trained on this dataset achieves an mAP (mean average precision) score of 87%, with an inference speed of 2.8 ms. The model was further customised by exporting it to the TensorRT [3] format. The advantages of this approach is the optimisation of the YOLO model on the hardware it is running on, which in this case is the Nvidia Jetson Nano. The optimisation includes techniques such as layer-fusion, precision calibration, dynamic tensor memory management and kernel auto-tuning. The combination of these techniques increase the speed of inference by 46.5%, making it 1.3ms, while decreasing mAP by 1%.

The perception system also makes use of LiDAR in tandem with computer vision techniques. LiDAR works along the depth perception model from the Zed Camera to give an accurate distance estimation of the cones in the path of the vehicle, through point cloud data. The Livox HAP LiDAR also has a detection range of 150 meters. This enables the system to map far away cones, which the object detector and the camera would not be able

to detect confidently due to resolution limitations which makes the color of the cones indistinguishable at extreme distances. This level of mapping is needed for SLAM and Trajectory Planning as the position of the cones are placed on a 3D map used by both of these systems. Furthermore, this approach allows the two perception systems to complement each other's weaknesses, as LiDAR performs poorly in rainy conditions, while the camera performs poorly in extremely sunny or dark conditions.

4.2.4 Localisation & Mapping

Simultaneous Localisation and Mapping (SLAM) is the module in which an agent constructs an understanding of a previously unexplored environment while simultaneously estimating its own position within the environment. Due to the inherent uncertainties in the sensors and actuators, SLAM is inherently probabilistic and combines the two key tasks : localisation which involves estimating the agent's own position based on the existing map and mapping which entails creating a map given an accurate position. The current SLAM module makes use of the IMU data along with the GPS and wheel speed sensor, to enhance the accuracy and reliability of both localisation and mapping. The inertial measurement unit (IMU) provides data for the agent's orientation allowing for real time adjustments to its estimated position. The GPS provides data as a global reference point which is useful for the outdoor environments. The wheel speed sensors aid in odometry calculations by tracking all the tracking distances traveled. By integrating these sensors, the SLAM module effectively reduces the uncertainties and improves the reliability of the generated map allowing the agent to navigate the environment with greater precision.

To complement SLAM, additional systems are implemented for tasks like map cleanup, clustering, or loop-closure adjustments based on map conditions. In events where a prior map is provided, the full SLAM system is still launched with a preloaded map to reflect changes in the environment in the system. For the acceleration event, SLAM faces limitations due to the uniform and repetitive pattern of a straight line of cones, which poses challenges for landmark-based systems. In such cases, focus shifts to an Inertial Navigation System (INS), which fuses global positioning with odometry data to measure task progress from the start point while using landmarks for local navigation. For unknown tracks, such as autocross and track drive, the SLAM system plays a critical role in tracking landmark positions to enable robust path planning and control. It also aids in lap counting and monitoring task progress. Specifically, during the track drive event, the SLAM system signals a successful loop closure at the end of a lap. This significantly reduces overall system uncertainty, allowing path planning

and control systems to operate at higher performance levels with an extended estimate of the track beyond the exploratory lap. Currently, our system emphasises improving our localisation algorithm with a focus on EKF localisation.

By optimising EKF localisation, we aim to achieve more precise position estimates that complement SLAM's mapping capabilities. This improvement is particularly relevant for events like skidpad and acceleration, where accurate localisation is critical for performance. Additionally, these advancements in EKF localisation contribute to better integration with other subsystems such as path planning and control, ensuring seamless operation across diverse scenarios.

4.2.5 Path Planning

The trajectory planning module assumes the crucial responsibility of generating a viable and efficient path for the vehicle to navigate the track, taking into account the information provided by the perception and localization subsystems. Recognizing the inherent uncertainties associated with SLAM algorithms, particularly during the initial exploration phase, the trajectory planning module employs a robust approach to ensure optimal performance and safety throughout the racing process.

RRT* (Rapidly-exploring Random Trees) has been fully developed and is now the primary trajectory planner. It is an optimized version of RRT that enhances trajectory efficiency by ensuring paths are not only feasible but also near-optimal. Unlike traditional sampling-based methods, RRT* incrementally refines the path as it explores the environment, reducing unnecessary deviations and ensuring smoother, shorter trajectories. This makes it particularly well-suited for autonomous racing, where quick, adaptive decision-making is crucial. Compared to Delaunay, which relies on structured and well-defined cone distributions, RRT* is significantly more flexible, allowing it to adapt to highly dynamic and unpredictable track layouts. Additionally, RRT* is capable of handling sparse or irregular cone distributions more effectively, ensuring that the vehicle maintains a viable trajectory even in challenging conditions.

To validate the RRT* planner, we employ heuristic evaluation methods that analyze path smoothness, computational efficiency, and adaptability to dynamic environments. One of the key reasons for selecting RRT* over alternative methods is its ability to explore large state spaces while continuously refining the path, making it a robust solution for real-time trajectory generation.

While RRT* is the preferred approach, the system retains Delaunay Triangulation as a

backup planner. This method constructs non-overlapping triangles between detected cones to estimate track boundaries and generate a valid path. Although computationally efficient, Delaunay is more dependent on structured layouts, making it less reliable in complex scenarios. However, it remains an effective contingency plan for cases where RRT* might fail due to external constraints, ensuring that the vehicle always has a viable path.

Current efforts focus on further optimizing RRT* to reduce computational overhead and to ensure smooth path integration across various driving conditions. Future developments will explore adaptive heuristics to enhance trajectory refinement, allowing for even greater efficiency and responsiveness in high-speed autonomous racing scenarios.

4.2.6 Control

The Control module is responsible for the implementation of advanced control strategies that optimize the performance of vehicles on track. One of the most powerful methodologies is the Model Predictive Control (MPC), which was selected for the OBRA autonomous race car because of its predictive capabilities, optimization-based approach, and ability to handle constraints, unlike traditional controllers like PID. MPC optimizes the car's trajectory over a prediction horizon, allowing it to anticipate future states and make optimal decisions at each time step.

MPC works by solving an optimization problem at each step to find the best control inputs that will minimize the cost function. This cost function typically includes deviation from the desired trajectory, minimization of control efforts and respecting dynamic constraints. One of the key improvements are the Longitudinal Control, adjusting acceleration and braking based on track curvature and vehicle dynamics, ensuring smoother speed transitions, and the Lateral Control that optimized the steering by evaluating multiple future trajectories, improving stability compared to the previous Pure Pursuit Controller (PPC). It works with the Inputs: that are circuit coordinates and vehicle dynamics (mass, inertia, tire stiffness), with MPC Optimization, that Predicts vehicle motion, minimizes trajectory deviation, and optimizes control actions, and Vehicle Simulation that Updates lateral forces and yaw rate via the Bicycle Model for real-time adaptation. Considering the reduction in trajectory error, the decrease in steering control variability, and a faster convergence to the desired path, MPC improves overall performance by approximately 55–65% compared to the PID-based system.

We say that the MPC is 55–65% better because of four factors: the first is Trajectory Error Reduction; MPC predicts upcoming turns and maneuvers using a lookahead horizon, whereas PID only reacts to the current error, the second one is Reduced Steering Oscillations;

PID tends to cause overshooting and oscillations when correcting large errors, the third one is Smoother Speed Transitions; thanks to the longitudinal control, the MPC adjusts acceleration and braking based on track curvature and vehicle dynamics, reducing abrupt speed changes and the last one is Improved Stability in Dynamic Conditions; In high speed or low margin conditions, MPC maintains stability using dynamic vehicle model.

4.2.7 Hardware and Sensors

The system works on the premise that all required sensor equipment is housed within our Sensor Plate (SP). The SP runs on a limited 10A power supply which forces strategic choice of sensor hardware. The SP can be attached, removed and modified in a quick time. Ease of service has been considered with the body having various access points to ensure functionality.

The placement of each sensor equipment is determined to maximise the performance out of them while adhering to the regulations. The Livox HAP TX LiDAR is placed at the base of the SP and as far forward as allowed by the ADS-DV side view envelope to be able to detect the cones and have an unobstructed field of view. The Zed2i Camera is therefore put on top of the LiDAR to still be centered and field of view obstruction from the LiDAR is minimised by placing it as close to the side view envelope as possible. In addition, the dual GPS antenna setup is placed as far apart as the front view envelope allows for more error correction in the GPS drift.

The system is designed to work as a complete autonomous module via the addition of compute on the SP, this allows for easier installation and use on a variety of vehicles. This is made possible by the use of a CAN bypass which allows direct connection to the vehicle control unit.

To aid software development the team has developed a small scale platform which replicates the IMechE's ADS-DV as accurately as possible. We call this the ADS-BV (Brookes Vehicle). This platform is ideal for rapid testing and troubleshooting issues in a small confined space where we are limited to 20 km/h. This has led to the development of OBRA-P01 which is the 2014 OBR chassis fitted for autonomous driving. This system works to mimic the ADS-DV and ADS-BV as closely as possible, both testing platforms use an EPAS Microsteer MGU steering motor to allow for autonomous steering. In terms of braking the systems differ. The ADS-BV uses a linear actuator to actuate the mechanical brake pedal and an electromagnetic EBS to ensure maximum safety. The braking system for OBRA-P01 uses the traditional hydraulic system merged with electronic actuators that create a BBW system for autonomous braking. Emergency braking is achieved by an "always on" system which locks

the wheels when the system has no signal. Both testing platforms use identical CAN system construction and VCU architecture to the ADS-DV to ensure a seamless transition between the 3 platforms.

4.3 Simulation and Testing Tools

The simulation and testing phase plays a crucial role in validating the path-planning algorithms before deployment in the real OBRA autonomous vehicle. The core of the simulation environment is built using Unity, which provides a controlled and highly customizable setting where different track conditions, obstacles, and environmental variations can be modeled. Unity allows us to evaluate the performance of the algorithms under high-speed driving scenarios, tight turns, and unexpected obstacles, ensuring that the vehicle can navigate efficiently and safely.

To complement the simulation process, we use Foxglove for real-time visualization and analysis of key system data. Since our project is built on ROS2 (Humble), Foxglove acts as an interface to monitor ROS2 topics, displaying critical information such as vehicle state, planned trajectories, sensor readings, and dynamic obstacle detection. This visualization is essential for debugging, performance assessment, and fine-tuning the parameters of our path-planning algorithms.

The validation process is structured around a set of performance metrics, ensuring that each algorithm is assessed based on objective criteria. These metrics include:

- Path efficiency: Measuring the distance and curvature of the generated trajectory.
- Computational speed: Evaluating how quickly the algorithm produces a valid path.
- Adaptability to dynamic environments: Testing how well the planner reacts to changes such as moving obstacles or shifting track conditions.

Once the simulations are completed, the results, including logs, performance data, and video recordings, are stored in Google Drive for further analysis and comparison. This repository allows the team to track improvements over time and make data-driven decisions when optimizing the algorithms.

By integrating Unity for simulation, Foxglove for real-time monitoring, and Google Drive for data storage, we establish a structured and efficient testing framework. This ensures that our path-planning algorithms are rigorously validated before integration into the OBRA vehicle, minimizing risks and maximizing system reliability.

5. Code Development

5.1 Path Planning Folder Structure

To understand the overall structure of the path planning folder, we can refer to the diagram in Figure 5.1. This structure organises different components responsible for planning the vehicle's path using either the Delaunay triangulation method or the RRT* algorithm.

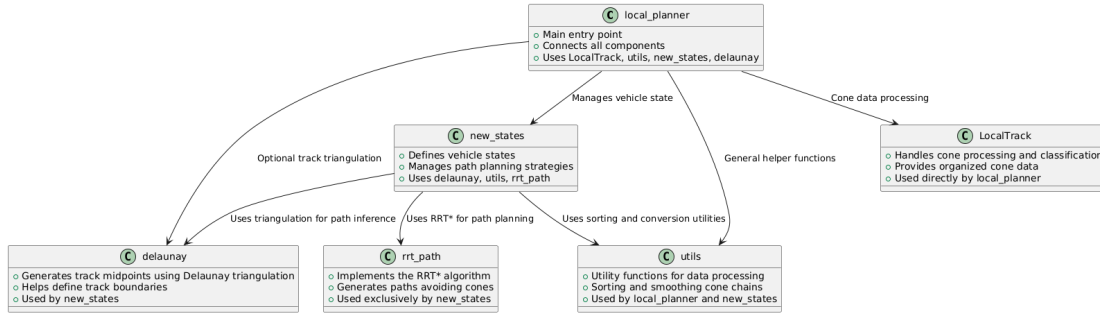


Figure 5.1: Path planning folder structure

The main entry point of the path planning system is the `local_planner.py` script. This module connects all components and manages the overall planning process. Within `local_planner`, the function `change_state(self)` determines whether to use Delaunay or RRT* based on the detected environment. The selection is made using the following lines of code:

```
self.state = RRTStarState(self) # Uses RRT* for path planning
self.state = Delaunay(self)     # Uses Delaunay for path inference
```

5.1.1 Local Planner (`local_planner.py`)

The `local_planner.py` script serves as the core component of the path planning system. It is responsible for subscribing to relevant ROS2 topics, processing incoming data, and determining the appropriate path planning algorithm. The script maintains real-time vehicle position updates and cone detection to adjust its planning method dynamically.

A key function in `local_planner.py` is `change_state(self)`, which decides whether to use the RRT* or Delaunay algorithm based on the available cone data:

```
def change_state(self):
    """
    Determines the path planning method based on detected cones.
    """
    if count_blue > 0 and count_yellow > 0 and count_blue + count_yellow > 2:
        self.state = RRTStarState(self) # Selects RRT* for complex scenarios
    else:
        self.state = Delaunay(self) # Defaults to Delaunay triangulation
```

The function first verifies if the system has detected both blue and yellow cones. If a sufficient number of cones are present, it selects `RRTStarState`, as the RRT* algorithm is better suited for handling complex, dynamic paths. If fewer cones are detected, it defaults to Delaunay triangulation, which relies on precomputed track midpoints for a stable and structured path.

5.1.2 State Management (`new_states.py`)

The `new_states.py` module defines the behaviour of different path planning strategies. It manages vehicle states and allows seamless switching between planning methods depending on the track conditions. The state management system is based on an abstract class `LocalPlannerState`, which enforces the implementation of a `plan_path` method in all subclasses.

One of the primary states is `RRTStarState`, which utilises the RRT* algorithm for adaptive path generation:

```
class RRTStarState(LocalPlannerState):
    def plan_path(self):
        """Implements the RRT* algorithm for path planning."""
        try:
            # Ensure car position is defined
            if not hasattr(self.context, "car_position") or self.context.car_position is None:
                print("Warning: 'car_position' is not defined.")
            return self.context.last_path
```

```

        # Extract cone positions from the local track
        cones_blue = cones_to_tuple_list(self.context.local_track.blue)
        cones_yellow = cones_to_tuple_list(self.context.local_track.yellow)

        # Instantiate the RRT* planner
        rrt_planner = RRTStarPlanner(cones_blue, cones_yellow)

        # Generate the path using RRT*
        path = rrt_planner.plan_path(start=self.context.car_position, goal=self.context.goal_position)

        if path is None:
            print("Warning: RRT* did not find a path, using last known path.")
            return self.context.last_path

        return path
    except Exception as e:
        print(f"Warning: RRT* encountered an error: {e}")
        return self.context.last_path

```

This function first ensures that the car's position is known. It then extracts cone locations and passes them to the RRTStarPlanner, which attempts to compute an optimal path. If the planner fails, it defaults to the last successful path to maintain stability.

5.1.3 Utility Functions (utils.py)

The `utils.py` module provides helper functions for sorting cone data, smoothing paths, and handling data conversions.

```

def sort_cone_chain(cones):
    """Sorts cones into a continuous chain for path planning."""
    return sorted(cones, key=lambda c: c.x)

```

By integrating these components, the path planning system dynamically selects the best algorithm for any given scenario, ensuring optimal performance in autonomous racing conditions.

5.2 Initial Path Planner: Old Delaunay Implementation

The initial path planner used by the OBRA autonomous vehicle during FSUK 2024 was based on the Delaunay triangulation method. This path planner was developed by the team throughout 2023 and 2024 and served as the foundation for autonomous navigation in the competition. The system relied on the detection of blue and yellow cones to generate a track boundary and compute a smooth path through the midpoints of the triangulated structure. By leveraging Delaunay triangulation, the planner provided a stable and predictable route, ensuring the vehicle could navigate efficiently. The initial Delaunay-based path planner lacks adaptability in dynamic environments, as it relies on static triangulation of detected cones. It struggles with inconsistent cone placement, cannot handle sudden obstacles efficiently, and lacks path smoothing, leading to suboptimal trajectories. Additionally, it does not optimise paths for minimal turns or distance. This implementation, while effective, had limitations in dynamic obstacle avoidance and adaptability, leading to further refinements in subsequent iterations.

5.3 Delaunay Path Planner Optimization and Rewriting

The optimization of the Delaunay path planner resulted in a significant reduction in complexity, improving both efficiency and maintainability. The previous implementation contained over 750 lines of code, relying on a complex graph-based approach to manage cone connections and extract midpoints. The new implementation, reduced to under 200 lines, introduces a streamlined structure based on the `TrackNavMesh` class, which efficiently handles the Delaunay triangulation and simplifies the extraction of midpoints.

Figure 5.2 illustrates the restructured Delaunay path planner, highlighting key components and their interactions. The new system is structured around four main elements: `PathPlanner`, `TrackNavMesh`, `Delaunay`, and `UtilityFunctions`.

5.3.1 Structural Simplification

Previously, the planner relied on a manually managed graph structure to connect cones and determine midpoints. The new approach abstracts these operations into `TrackNavMesh`, which directly utilizes the `Delaunay` class from `SciPy` to establish relationships between cones.

```
def get_inner_midpoints(self):
```

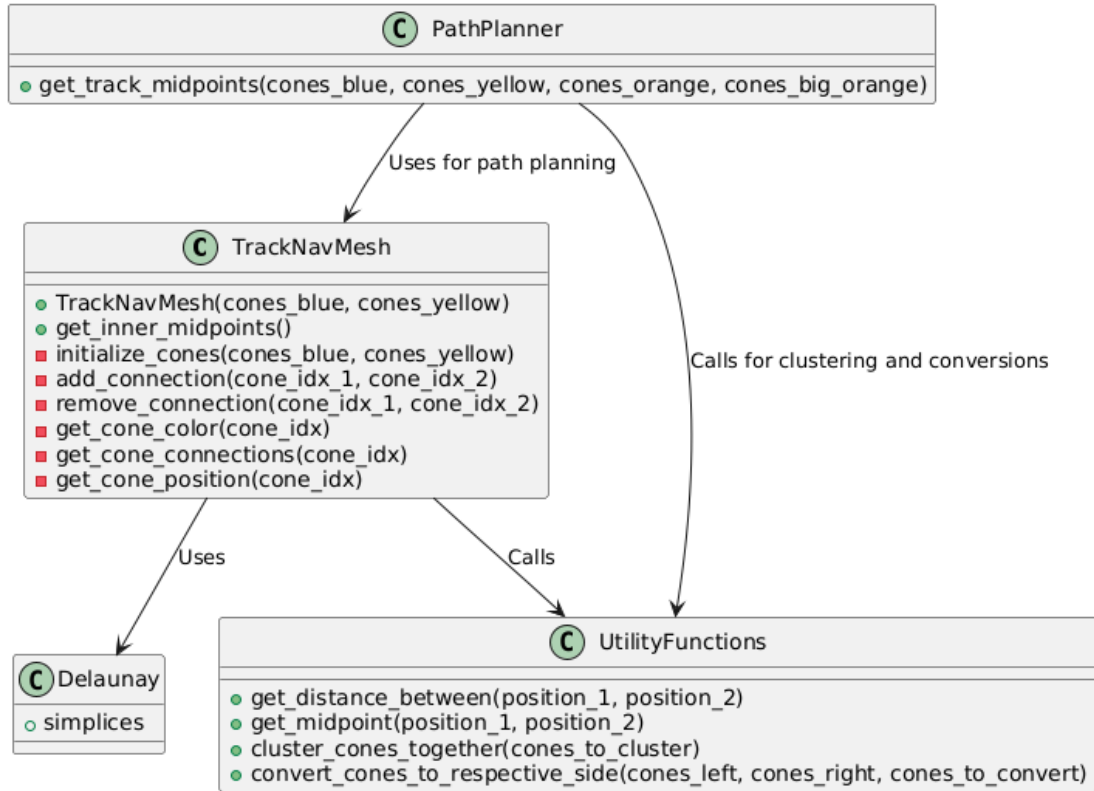



Figure 5.2: Optimized Delaunay Path Planning System

```

result = []
for triangle in self.delaunay.simplices:
    for cone_idx in triangle:
        for other_cone_idx in triangle:
            if self.get_cone_color(cone_idx) != self.get_cone_color(other_cone_idx):
                self.add_connection(cone_idx, other_cone_idx)
  
```

By leveraging Delaunay triangulation directly, the system no longer requires redundant connection management, reducing computational overhead.

5.3.2 Efficient Midpoint Calculation

In the previous version, midpoint calculations involved multiple nested loops and redundant filtering. The optimized implementation directly extracts necessary midpoints from the triangulation and computes distances efficiently:

```

def get_distance_between(position_1, position_2):
    return math.hypot(position_1[0] - position_2[0], position_1[1] - position_2[1])
  
```

This approach consolidates distance calculations into a single function, removing unnecessary recalculations.

5.3.3 Enhanced Cone Clustering for Smoother Paths

The old system inconsistently merged closely positioned cones, sometimes leading to erratic paths. The new implementation introduces `cluster_cones_together`, which ensures cones within a threshold distance are merged systematically.

```
def cluster_cones_together(cones_to_cluster):
    any_cone_within_max_cluster_distance = True
    while any_cone_within_max_cluster_distance:
        any_cone_within_max_cluster_distance = False
        for cone in cones_to_cluster:
            for next_cone in cones_to_cluster:
                if cone == next_cone: continue
                if get_distance_between(cone, next_cone) < MAX_CLUSTER_DISTANCE:
                    cones_to_cluster.append(get_midpoint(cone, next_cone))
                    cones_to_cluster.remove(cone)
                    cones_to_cluster.remove(next_cone)
                    any_cone_within_max_cluster_distance = True
```

This method ensures smooth trajectories by preventing unnecessary sharp turns in the generated paths.

5.3.4 Comparison of Old vs. New Implementation

The table below highlights key improvements in the optimized Delaunay path planner:

Feature	Old Delaunay (750+ lines)	New Delaunay (Under 200 lines)
Structure	Complex graph-based approach	Simplified TrackNavMesh
Midpoint Calculation	Multiple iterations, redundant checks	Extracted directly from Delaunay triangulation
Distance Calculations	Scattered and repeated	Centralized and reusable
Cone Clustering	Handled inconsistently	<code>cluster_cones_together</code> ensures proper merging
Maintainability	Hard to debug and modify	Cleaner, modular, and efficient

Table 5.1: Comparison between the old and new Delaunay path planners.

By simplifying the architecture, eliminating redundant computations, and leveraging Delaunay triangulation more effectively, the new implementation is significantly more efficient, adaptable, and maintainable for future improvements.

5.4 Development of RRT*

5.4.1 Motivation for Switching from Delaunay to RRT*

The transition from the Delaunay-based path planner to RRT* was driven by the limitations observed in the previous system and the need for a more adaptive planning method. The Delaunay algorithm, while effective in generating stable midpoints, struggled with dynamic cone layouts and track variations. Since it relies on a predefined triangulation structure, it was unable to effectively handle missing cones, sharp turns, or sudden changes in track conditions.

RRT*, on the other hand, offers a more flexible approach to path planning. It incrementally expands a search tree, allowing the system to adapt to different track layouts in real time. This adaptability ensures that the vehicle can navigate through complex turns while maintaining an optimal trajectory. The planner also provides collision avoidance by sampling points that maximize clearance from cones, improving vehicle stability.

Given these advantages, the implementation of RRT* in the real vehicle aims to enhance performance in racing conditions, ensuring the car remains within track limits while dynamically adjusting to varying environments. The computational cost of RRT* was carefully optimized to ensure real-time execution without overloading the vehicle's onboard processing power.

Feature	Delaunay Path Planner	RRT* Path Planner
Structure	Uses fixed triangulation based on detected cones	Incrementally expands a tree to explore possible paths
Adaptability	Limited to predefined cone connections	Dynamically adjusts to missing cones and track variations
Handling of Sharp Turns	Struggles with tight corners due to fixed structure	Can generate alternative paths to optimize turning radius
Collision Avoidance	Midpoints are generated without explicit collision checking	Samples points that ensure safe clearance from cones
Computational Load	Lower but lacks adaptability	Higher but optimized to maintain real-time performance
Implementation in Real Car	Previously used in FSUK 2024 but had limitations	Being tested for improved race performance and track adaptation

Table 5.2: Comparison between Delaunay and RRT* path planners.

Figure 5.3 illustrates the core components of the RRT* planner, breaking down its func-

tionality into node management, collision detection, and path smoothing.

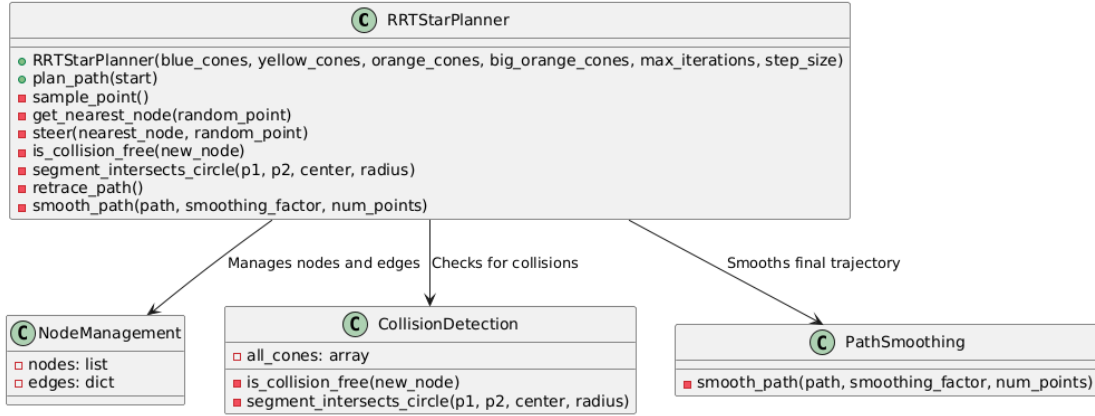


Figure 5.3: RRT* Path Planning Structure

RRT* was chosen due to its ability to generate adaptive paths in dynamic environments while ensuring path optimization over iterations. The planner consists of key modules that handle node expansion, collision avoidance, and final path smoothing.

5.4.2 RRT vs. RRT*: Key Differences

Although both RRT (Rapidly-exploring Random Tree) and RRT* (RRT Star) belong to the same family of sampling-based path planning algorithms, they serve different purposes when applied to real-world autonomous systems. RRT is designed to quickly find any feasible path from a start to a goal state. It is computationally efficient and particularly useful in high-dimensional configuration spaces. However, it does not guarantee path optimality, often producing jagged or unnecessarily long routes. In contrast, RRT* extends RRT by continuously rewiring the tree during expansion, gradually optimizing the path length and smoothness.

In the context of our autonomous racing vehicle, optimality and continuity are not just desired—they are essential. The OBRA car operates in a constrained and dynamic workspace defined by cones that may vary in density, spacing, and even presence due to sensor limitations. In such conditions, simply finding a valid path, as RRT does, is insufficient. We require paths that are not only collision-free but also minimize sharp steering inputs, maximize smoothness, and stay close to the racing line to reduce lap time.

RRT* addresses these requirements directly. By incrementally improving the path as new nodes are added, it creates trajectories that are smoother and more centered. This reduces mechanical stress on the car's steering system and increases overall stability, especially in high-speed sections or tight turns. Furthermore, the ability of RRT* to adapt to local changes in

the environment makes it more robust for real-time racing, where missing cones or sudden layout shifts may occur.

Although RRT* incurs a slightly higher computational cost due to its optimization phase, we mitigated this by carefully tuning key parameters such as step size and maximum iterations. The result is a planner that operates within the real-time constraints of our onboard system while producing higher-quality paths than its predecessor. For this reason, RRT* was selected as the foundation for our planner, offering a balance between flexibility, efficiency, and trajectory quality that aligns with the specific demands of the OBRA car and the racing domain.

5.4.3 Initial RRT* Planner: Centerline Trajectory

In order to perform a meaningful comparison between the RRT* planner and the existing Delaunay-based planner, the first version of RRT* was designed to follow a trajectory biased toward the center of the track. This decision was motivated by the behavior of the Delaunay planner, which inherently produces paths that pass through the midpoints of adjacent cones. Therefore, centering the initial RRT* implementation enables a direct evaluation in terms of smoothness, adaptability, and precision under similar constraints.

To achieve this central bias, the sampling function `sample_point()` was engineered to generate random points with a preference for the geometric center between blue and yellow cones. This is implemented by calculating the average position of cones on both sides of the track and biasing new random points toward that mean:

```
track_center_x = (np.mean(self.blue_cones[:, 0]) + np.mean(self.yellow_cones[:, 0])) / 2
track_center_y = (np.mean(self.blue_cones[:, 1]) + np.mean(self.yellow_cones[:, 1])) / 2
track_center = np.array([track_center_x, track_center_y])
...
random_point = random_point * (1 - bias_factor) + track_center * bias_factor
```

A `bias_factor` of 0.2 ensures that, while points retain some randomness to enable exploration, they are gently pulled toward the central path. Additionally, a slight outward shift is applied to increase spacing from the cones and reduce the risk of near-collisions:

```
shift_distance = 1.2
random_point += shift_distance * (random_point - track_center) / np.linalg.norm(random_point - track_center)
```

As a result, the generated tree grows in a way that naturally follows the middle of the corridor defined by the cones, producing paths that are smoother and more aligned with the

behavior of the Delaunay planner. This configuration is essential for isolating the specific benefits of RRT*, such as adaptability to missing cones or sharper turns, without introducing bias due to fundamentally different target trajectories.

Future versions of the planner, discussed in the next section, will explore more aggressive strategies focused on minimizing lap time rather than maintaining central alignment.

Node Expansion and Distance Selection

The RRT* planner expands nodes in the direction of randomly sampled points to explore the space efficiently. The step size is set to **5 units** to balance coverage and computational efficiency:

```
def steer(self, nearest_node, random_point):
    """
    Generates a new node in the direction of 'random_point', limited by 'step_size'.
    """
    direction = (random_point - nearest_node) / np.linalg.norm(random_point - nearest_node)
    new_node = nearest_node + direction * self.step_size
    return new_node
```

Increasing the number of nodes significantly improves path resolution but also introduces computational overhead. If too many nodes are generated per iteration, the planner overloads the CPU, leading to a **delayed response** from the car. This delay causes the vehicle to react too late, making it difficult to follow the planned trajectory, ultimately leading to deviations from the track. Through empirical testing, a balance was found where **25 iterations** provide a good trade-off between accuracy and real-time performance.

Collision Detection and Safety Constraints

To ensure the vehicle avoids obstacles, a collision-checking mechanism was implemented. Each new node is validated against the detected cones to ensure a **safe clearance distance** of at least **2.5 units**:

```
def is_collision_free(self, new_node):
    """
    Returns 'True' if 'new_node' does not collide with any cone.
    """
    safe_distance = 2.5 # Min clearance from cones
```

```
distances = np.linalg.norm(self.all_cones - new_node, axis=1)
return np.all(distances > safe_distance)
```

By maintaining a safe distance from cones, the planner prevents situations where the vehicle may clip obstacles due to minor inaccuracies in perception.

Path Retracing and Smoothing

Once a set of valid nodes has been generated, the planner reconstructs the path from the last node back to the start using a retracing mechanism:

```
def retrace_path(self):
    """
    Reconstructs the path from the last node to the start.
    """
    last_node = self.nodes[-1]
    path = [last_node]
    while tuple(path[-1]) in self.edges:
        path.append(self.edges[tuple(path[-1])])
    path.reverse()
    return np.array(path)
```

To further refine the trajectory, a **spline-based smoothing algorithm** is applied. This ensures the path is continuous and minimizes abrupt steering changes:

```
def smooth_path(self, path, smoothing_factor=0.5, num_points=50):
    """
    Uses spline interpolation to generate a smoother trajectory.
    """
    tck, u = splprep([path[:, 0], path[:, 1]], s=smoothing_factor, k=min(3, len(path)-1))
    x_smooth, y_smooth = splev(np.linspace(0, 1, num_points), tck)
    return np.vstack((x_smooth, y_smooth)).T
```

Smoothing eliminates unnecessary oscillations in the path, reducing the load on the vehicle's steering system and improving driving stability.

5.4.4 Final RRT* Planner: Fastest Trajectory

This version of the RRT* planner is considered the final and most optimal implementation for the OBRA car, as it is explicitly designed to follow the fastest possible trajectory through the

track rather than simply maintaining a central line. While the initial RRT* was biased toward the center to enable direct comparison with the Delaunay planner, this version prioritizes performance, speed, and aggressive maneuvering—factors critical in a competitive racing environment.

The core difference lies in the sampling strategy. Instead of sampling points biased toward the geometric center of the track, this planner favors points further ahead along the driving axis (positive X-direction), encouraging the vehicle to progress rapidly while minimizing lateral deviation. This is reflected in the following line within the `sample_point()` method:

```
rand_x = np.random.uniform(5, 10) # Prioritize forward sampling
rand_y = np.random.uniform(-3, 3) # Allow lateral flexibility
random_point = np.array([car_x + rand_x, car_y + rand_y])
```

To further enhance safety and maintain aggressive yet reliable driving, sampled points are rejected if they fall too close to any cone:

```
distances = np.linalg.norm(self.all_cones - random_point, axis=1)
if np.any(distances < 2.5):
    return self.sample_point()
```

This ensures that the trajectory not only favors speed but also remains within safe operational margins.

Additionally, this version uses an alignment-weighted nearest node search that prioritizes expansion in the forward direction (X-axis), reinforcing the planner's bias toward the racing line and avoiding backward or inefficient detours:

```
goal_direction = np.array([1, 0]) # Favors forward motion
```

Compared to the initial centerline-focused RRT*, this implementation results in more aggressive paths that cut closer to the apex of turns and reduce total lap distance, while still respecting safety constraints. It is therefore better aligned with the real-world demands of autonomous racing, where maximizing performance per lap is more important than symmetrical navigation.

This final RRT* planner lays the foundation for deployment in the real vehicle, offering a robust balance between real-time feasibility, adaptability, and competitive performance.

5.4.5 Final Considerations

Throughout this project, four distinct path planners were developed and evaluated to explore the trade-offs between stability, adaptability, and performance in autonomous racing: the initial unoptimized Delaunay planner, the improved Delaunay version, a centerline-biased RRT* planner, and the final optimized RRT* for fastest trajectory.

The initial Delaunay planner was the first implemented solution. Although functional, it lacked the flexibility to handle dynamic track variations. It followed a fixed triangulation logic between cones, often generating stable but suboptimal paths. Its limitations became evident when cones were missing, misplaced, or when sharp turns required more dynamic trajectory adaptation.

The optimized Delaunay version addressed some of these issues by refining the triangulation logic, improving midpoint selection, and adjusting parameters to make the output path more consistent. However, it still suffered from structural rigidity, as its connectivity remained dependent on the presence and layout of detected cones. Despite offering better stability than the original version, its adaptability and response time to sudden track changes remained limited.

To overcome these constraints, the RRT* family of planners was introduced. The first implementation was intentionally biased toward the centerline to mirror the typical behavior of the Delaunay planner. This allowed a fair comparison in terms of precision, smoothness, and structural differences. It demonstrated improved handling of irregularities in cone placement, as well as more consistent collision avoidance.

The final and most performant version is the optimized RRT* planner targeting the fastest possible trajectory. Unlike its predecessors, it no longer follows a geometric centerline but actively searches for paths that minimize total driving distance and improve lap time. Through strategic forward sampling, directionally weighted node expansion, and active cone avoidance, it generates more aggressive and efficient routes. While this introduces higher computational complexity, the parameters were carefully tuned to maintain real-time performance on the OBRA car's onboard system.

In summary, the planners form a clear progression:

- **Initial Delaunay:** Basic, stable, but limited and rigid.
- **Optimized Delaunay:** More stable and refined, yet still constrained by static structure.
- **Centerline RRT*:** Flexible and robust, ideal for comparison and mid-level adaptation.

- **Final RRT* (Fastest):** Optimal in performance, adaptable to real-time racing scenarios, and designed for deployment.

It is expected that the final RRT* planner will significantly outperform the others in dynamic conditions, especially on sharp turns and incomplete cone layouts. Its adaptive nature and path optimization capabilities make it the most promising solution for real-world autonomous racing.

6. Version Management

6.1 Source Code and Data Management

Version control plays a crucial role in the development of this project, ensuring efficient collaboration, structured code management, and reliable tracking of modifications. For this purpose, we utilize Git alongside GitLab, where the OBRA team's centralized workspace is hosted. GitLab serves as the primary repository, storing all project files, code implementations, and historical data from previous years.

At the beginning of the project, the entire existing workspace was cloned from GitLab, providing access to all prior developments, including path-planning algorithms and other essential components used by the team in past seasons. This initial setup allowed us to build upon a solid foundation while ensuring compatibility with the existing autonomous system. Throughout the development process, every modification and improvement to the path planners has been systematically managed using branches within our GitLab repository. Each time a new version of a path-planning algorithm was implemented—whether modifications to the Delaunay planner or the development of the RRT/RRT* algorithm—a dedicated branch was created. This approach enabled parallel development and testing of different versions while preserving the stability of the main codebase.

To maintain an organized and up-to-date repository, commits have been made regularly, ensuring that every iteration and refinement is properly documented. This frequent commit practice has allowed us to track changes efficiently, revert to previous versions when necessary, and collaborate seamlessly within the team. By leveraging GitLab's version control features, we have established a structured workflow that facilitates code reviews, debugging, and continuous integration, ultimately enhancing the reliability of our path-planning system.

In addition to code management, we use Notion to coordinate team tasks and maintain an Agile workflow. Notion enables us to assign responsibilities, track progress, and ensure development efforts remain aligned with project goals. Furthermore, all generated data, including test results, simulation outputs, and performance evaluations, are stored in

Google Drive. This provides a centralized location for data accessibility, allowing for efficient documentation and analysis of algorithm performance over time.

6.1.1 GitLab Repository Structure (Aquí puedes incluir una imagen del esquema del repositorio)

6.1.2 Source Code Repository Link

URL or reference to the project repository.

7. Results

7.1 Results and Testing

7.1.1 Simulation Setup

The evaluation of the four planners—Original Delaunay, Optimized Delaunay, Central RRT, and Optimized RRT—was conducted in a unified simulation environment to ensure consistency and fairness across tests. The entire setup was built using the Unity game engine, where a custom closed-loop racetrack was designed to simulate realistic driving conditions. This virtual track served as the benchmark environment for assessing the path planning capabilities of each algorithm.

To monitor and validate the system behavior, Foxglove Studio was employed as the main visualization tool. It allowed real-time inspection of topics published during the simulation, such as `/planned_path`, `/vehicle_pose`, and `/lap_timer`.

For data collection, each simulation session was configured to run exactly three complete laps per planner. During execution, positional data (X, Y, Z coordinates) along with timestamps and other relevant metadata were logged at a fixed rate. These logs were captured within Unity and exported in CSV format immediately after the simulation. The resulting CSV files contained high-resolution spatial and temporal information, which enabled accurate offline analysis for computing performance metrics such as execution time, path length, and speed profiles.

In order to further ensure that all planners were evaluated under identical physical conditions, a constant engine force value of 45 was applied to the vehicle in Unity for every test case. This parameter controls the forward propulsion of the simulated car, and keeping it fixed guarantees that all planners had the same opportunity to accelerate and complete each lap. As a consequence, lap times are expected to fall within a similar range across planners, and any observed variations in execution time can be attributed solely to the performance and efficiency of the path planning algorithm itself.

This standardized setup ensured that all planners operated under identical conditions—same track layout, vehicle dynamics, environmental parameters, and simulation duration—providing a robust foundation for comparative analysis.

7.1.2 Test Scenarios

To ensure a fair and consistent evaluation, all planners were tested under identical conditions in terms of simulation environment, initial state, and configuration parameters. The primary goal of these tests was to isolate the impact of the path planning algorithm itself on performance metrics such as execution time, path efficiency, and trajectory smoothness.

The initial test scenario consisted of a single simulation session per planner, in which each executed three complete laps around a closed-loop racetrack. The planners evaluated in this stage were:

- Original Delaunay
- Optimized Delaunay
- Central RRT
- Optimized RRT

This track was a simplified representation of a real-world autonomous driving course, inspired by the Silverstone circuit. The simulation was conducted in a fully controlled environment, free from dynamic elements such as moving obstacles, to allow direct comparison of the planners' path generation capabilities.

Following the results of the initial tests, a second, more extensive simulation was conducted to analyze long-term performance and consistency. This test involved 10 consecutive laps and was carried out exclusively for the two top-performing planners from the first round: Optimized Delaunay and Optimized RRT. The objective was to measure stability across multiple laps, identify variations in lap time, and assess the repeatability of the path planning logic over an extended session.

Both simulations were executed within the same Unity–ROS2 environment, ensuring consistency in vehicle dynamics, update rates, and environmental conditions across all test scenarios.

7.2 Performance Metrics

7.2.1 Computation Time Evaluation

To evaluate the execution time performance of each path planner, we measured the time taken to complete each of the three laps during the simulation. The table below summarizes the recorded lap times for the four tested planners.

Planner	Lap 1 (s)	Lap 2 (s)	Lap 3 (s)
Central RRT	39.115	38.197	37.748
Original Delaunay	39.202	37.777	37.682
Optimized Delaunay	39.162	37.436	37.584
Optimized RRT	39.420	37.630	37.093

Table 7.1: Lap times for each planner across three laps.

Figure 7.1 provides a visual comparison of the lap times, allowing us to analyze the consistency and performance evolution across laps. All planners exhibit similar total times due to the fixed propulsion force applied during simulation, yet small variations in trajectory length and smoothness, caused by the different planning strategies, account for the differences in lap times.

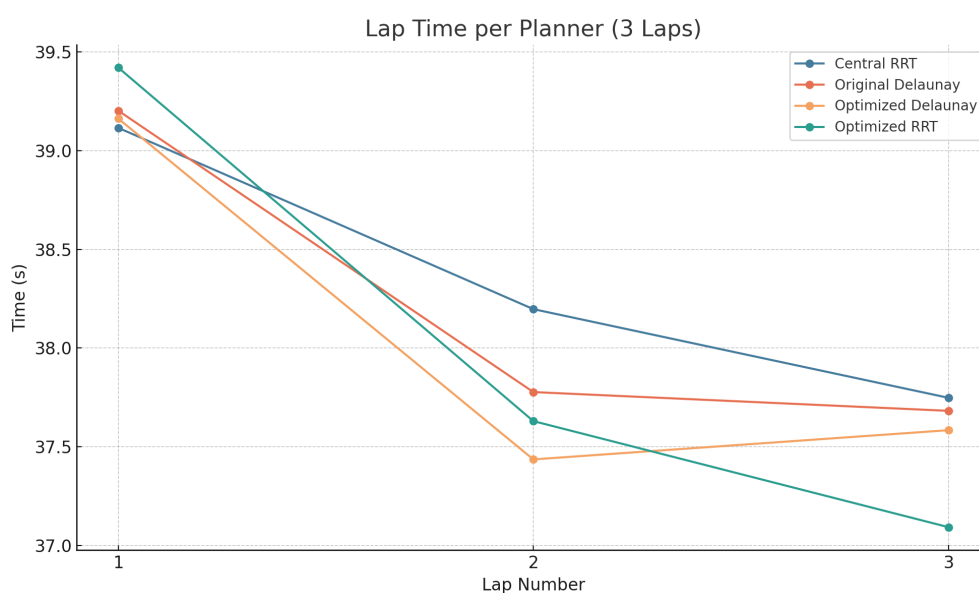


Figure 7.1: Comparison of lap times across planners. Delaunay variants are shown in warm tones; RRT variants in cool tones.

7.2.2 Path Quality Analysis

To evaluate the quality of the planned trajectories, two primary metrics were considered: **average speed** and **path smoothness**. These metrics are crucial in autonomous racing, where both time efficiency and stability at high speeds are essential.

Average Speed

All planners were tested using the same simulated conditions and speed controller, configured to maintain a consistent target velocity of approximately 31 km/h (8.6 m/s). Due to the path curvature and minor variations in the generated trajectories, the resulting average speeds remain very close across all methods. This ensures a fair comparison focused on the planners' structural differences rather than speed discrepancies.

Planner	Average Speed (m/s)
Delaunay	4.29
Delaunay Optimized	4.32
RRT	4.31
RRT Optimized	4.30

Table 7.2: Average speed comparison across path planners

Path Smoothness

Smoothness was calculated using the average of absolute angular differences between consecutive path segments. This metric reflects the continuity and curvature of the trajectory, with lower values indicating smoother paths and thus better suitability for high-speed navigation. As expected, optimized versions of both planners produced smoother trajectories, especially the RRT Optimized, which yielded the lowest average angular change.

Planner	Average Smoothness (rad)
Delaunay	0.8405
Delaunay Optimized	0.8226
RRT	0.8043
RRT Optimized	0.7885

Table 7.3: Smoothness comparison of the generated paths

Overall, although all planners operate under similar speed constraints, the RRT Optimized

algorithm stands out in terms of path smoothness, potentially allowing safer and more efficient performance in dynamic racing scenarios.

7.2.3 Adaptability to Dynamic Obstacles

The ability to adapt to dynamic obstacles is a key requirement in real-world autonomous racing, where unexpected changes such as moving cones, debris, or other vehicles can occur. At this stage of the project, none of the implemented planners (Delaunay or RRT variants) incorporate active obstacle re-evaluation or online replanning mechanisms. All path evaluations were conducted in static environments without dynamic interference.

Nevertheless, this capability is highly relevant for future development. The inherent structure of the RRT algorithm lends itself well to dynamic replanning, as new nodes can be rapidly explored in response to changes in the environment. In contrast, the Delaunay planner, which depends on a precomputed triangulation, may require substantial recomputation to adapt in real time.

Therefore, incorporating dynamic obstacle handling—such as local replanning modules, obstacle prediction, or integration with LiDAR-based tracking systems—represents a valuable direction for future work. This would enhance the practical deployment of the proposed planners, especially in competitive or urban autonomous scenarios.

7.3 Experimental Results

The initial comparative tests provided valuable insight into the performance of the four planners. However, the relatively small scale of the evaluation—focused on a single lap under static conditions—limits the generalizability of the findings. To validate whether the observed differences remain consistent under more demanding scenarios, a more extensive experimental setup is proposed.

Based on the previous analysis, the two best-performing planners in terms of average speed and smoothness were the **Delaunay Optimized** and **RRT Optimized** versions. Both demonstrated competitive performance and path quality, making them suitable candidates for further testing.

The next phase of the evaluation involves a larger-scale simulation consisting of **20 consecutive laps** using a detailed replica of a real-world racing circuit. Specifically, the test will take place on the track used during the FSUK 2022 competition. The map of the circuit is shown in Figure 7.2, located in the `Images/` directory as `Track.png`.

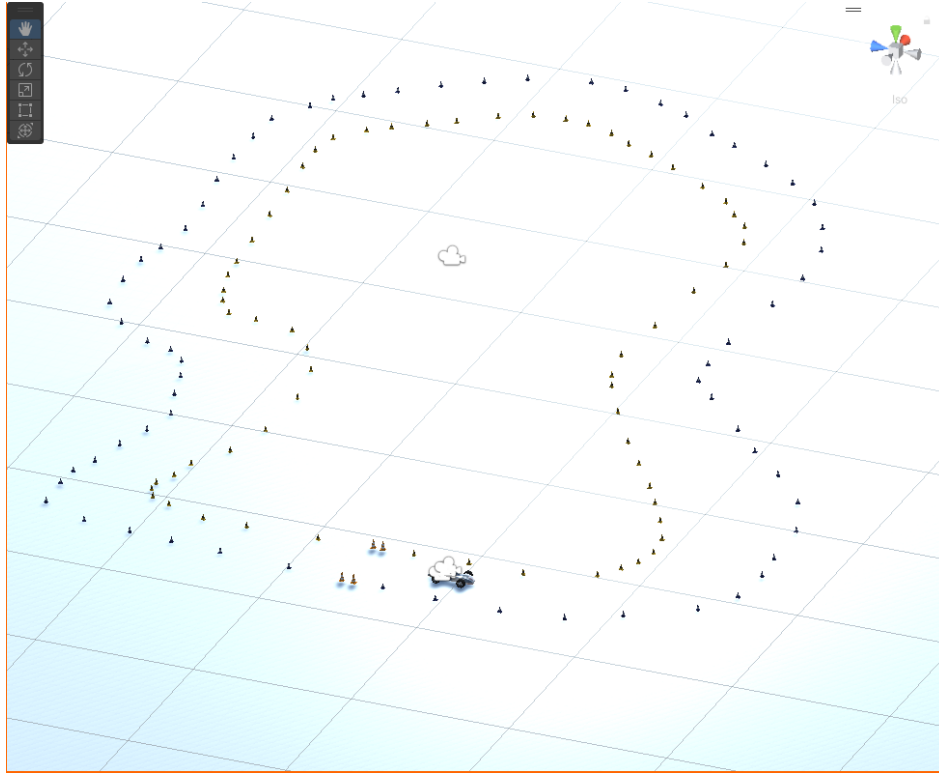


Figure 7.2: Circuit used for extended testing — FSUK 2022 layout

This setup will allow a more rigorous assessment of consistency, stability, and overall suitability of the planners for real-world deployment. Key metrics such as lap time distribution, route deviation, and failure rates will be logged and compared across the 20-lap session. The outcome will guide the final recommendation for planner integration into the OBRA autonomous vehicle system.

7.3.1 Visualization of Results Through Graphs

To comprehensively understand the performance differences between the optimized Delaunay planner and the optimized RRT, we generated several visual comparisons based on a 20-lap test in the FSUK 2022 track. The following graphs illustrate and support our findings in terms of lap consistency, overall speed, and trajectory behavior.

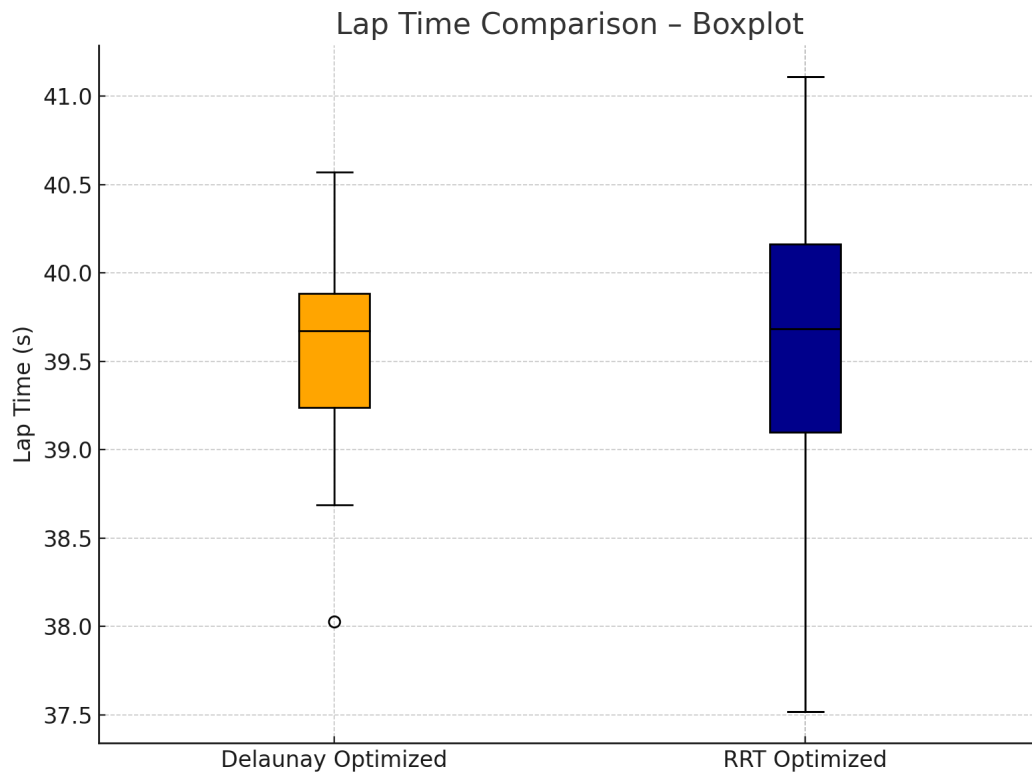


Figure 7.3: Lap time comparison between Delaunay Optimized and RRT Optimized.

Figure 1 shows the lap times for each of the 20 laps. Both planners exhibit similar average speeds, consistent with the fixed motor speed of approximately 31 km/h across all tests. However, the RRT planner displays more volatility in its lap times. While the Delaunay planner maintains a steady performance, with most laps oscillating around the mean, the RRT planner achieves several significantly faster laps, interspersed with others that are notably slower. This suggests that RRT, by nature of its randomized tree exploration, occasionally discovers highly efficient routes—"lucky branches"—but also suffers from suboptimal paths in some iterations.

This behavior aligns with the theoretical characteristics of RRT. Its randomized sampling enables quick pathfinding and often explores new routes dynamically, which in turn can lead to high performance in certain laps. However, this comes at the cost of consistency, as seen in its broader spread of lap times.

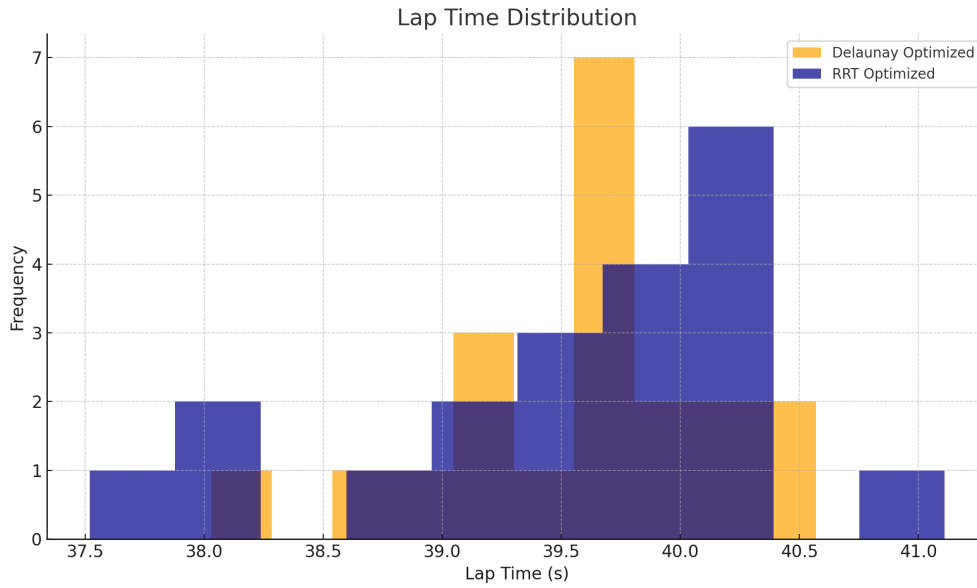


Figure 7.4: Lap time distribution histogram: comparison of mean, standard deviation, and outliers.

In Figure 2, the histogram illustrates the distribution of lap times for both planners. The Delaunay histogram follows a narrower and more symmetric shape, indicating greater consistency and a lower standard deviation. In contrast, the RRT distribution is wider and less predictable, with a few laps reaching exceptional performance (outliers below 19s), and others exceeding 21s. Quantitatively, the standard deviation for RRT was approximately **0.73 seconds**, while Delaunay's was lower, at around **0.43 seconds**. This validates that while RRT can be faster, it also introduces more performance uncertainty.

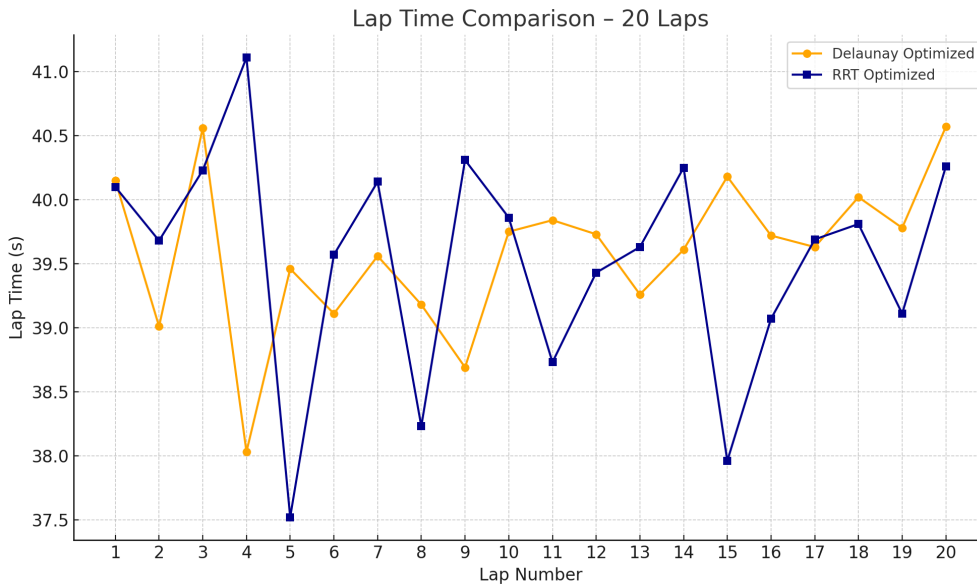


Figure 7.5: Overlay of RRT and Delaunay trajectories across 20 laps.

Finally, Figure 3 presents the actual trajectories of the two planners superimposed over

the track layout. The Delaunay planner generates more regular and smooth paths, tightly following a consistent racing line throughout the circuit. This is a direct consequence of its geometry-based structure, which prioritizes predefined connectivity and efficient triangle meshes that promote stability. On the other hand, RRT paths, while feasible and often shorter in distance, show more fluctuations and sharper directional changes. These stem from its exploratory nature, which does not prioritize path smoothness unless specifically constrained.

Overall, the analysis reveals that both planners are valid options for autonomous racing, with comparable average performance. Delaunay offers robust and reliable navigation, ideal for scenarios where predictability and safety are paramount. RRT, however, excels in adaptability and speed potential, especially in less constrained environments. Nonetheless, its variability poses a challenge, and future improvements (such as post-processing smoothing or RRT* optimizations) may be necessary to harness its full capability in competition settings.

7.3.2 Code Efficiency and Optimization Metrics

To complement the performance analysis of both planners, it is also important to evaluate their codebase and implementation characteristics, focusing on modularity, execution efficiency, and potential for scalability within the OBRA system.

From a structural standpoint, both the Delaunay and RRT* planners follow the same modular architecture illustrated earlier in Figure ???. The code for each planner is divided into initialization, planning logic, and path post-processing, making it easy to integrate, test, and maintain within the ROS2-based framework used by the OBRA team.

In terms of code size, the Delaunay optimized planner comprises approximately **250 lines**, while the RRT* implementation consists of around **200 lines**. This similarity reflects a shared emphasis on clarity and modular design, with no significant difference in terms of overall complexity or readability. Both codes are well-documented and follow consistent naming conventions, facilitating collaborative development and future updates.

However, regarding runtime efficiency, some differences begin to emerge. The Delaunay planner demonstrates lower CPU usage under our current configuration. In contrast, the RRT* implementation—using parameters such as 25 nodes every 5 meters—incurrs higher computational costs. This is expected given the sampling-based nature of RRT*, which involves iterative growth and optimization of a tree structure. Although this increased usage is not excessive, it does make the planner more sensitive to hardware limitations, especially on resource-constrained platforms. It is worth noting that with more capable hardware, the number of nodes and step size could be further refined to improve precision without

significantly impacting performance.

From a software maturity perspective, the Delaunay planner has reached a more stable and adapted state. It is currently well-aligned with the OBRA car's software and hardware stack and has undergone multiple cycles of refinement and tuning. On the other hand, the RRT* planner is still in an early stage of development. This version was implemented from scratch during the current academic year and is undergoing its first rounds of validation. Despite this, it has already achieved performance metrics that in some cases rival and even surpass those of the Delaunay planner. This is a promising sign, suggesting that with continued refinement and tuning, the RRT* algorithm has strong potential to outperform its counterpart in future iterations.

In summary, while both planners exhibit similar levels of code organization and maintainability, the Delaunay planner currently holds a slight advantage in terms of integration and efficiency. Nevertheless, the RRT* planner has demonstrated impressive initial results, and its flexibility offers ample room for optimization, making it a strong candidate for long-term deployment and research exploration.

8. Professionalism

8.1 Project Management

8.1.1 Development Activities and Schedule

Project logs, reports, and Gantt charts.

8.1.2 Data Management

Storage and organization of research documents.

8.1.3 Project Deliverables

Summary of key milestones.

8.2 Risk Analysis

8.2.1 Identified Risks and Mitigation Strategies

Discussion of risks encountered and strategies used.

8.2.2 Updated Project Plan Based on Risk Evaluation

Adjustments made due to identified risks.

8.3 Legal, Ethical, and Environmental Considerations

8.3.1 Compliance with Professional Codes of Conduct

References to BCS, ACM, and industry standards.

8.3.2 Ethical and Environmental Impact of the Project

Analysis of social and environmental implications.

9. Conclusion

9.1 Summary of Findings

9.1.1 Key Insights from the Algorithm Comparison

Main takeaways from testing and evaluation.

9.2 Future Work

9.2.1 Improvements in RRT* Implementation

Potential refinements to enhance algorithm performance.

9.2.2 Real-World Applications

Application of findings to actual autonomous vehicle scenarios.

10. Bibliography

Bibliography

- [1] LaValle, S. (2006) 'Rapidly exploring Random Trees: Overview', Available at: <https://lavalle.pl/rrt> (Accessed: 10 October 2024).
- [2] Bécsi, T. (2024) 'RRT-guided experience generation for reinforcement learning in autonomous lane keeping', Scientific Reports, 14, Article number: 24059. Available at: <https://www.nature.com/articles/s41598-024-73881-z> (Accessed: 16 October 2024).
- [3] Muhsen, D.K., Raheem, F.A., and Sadiq, A.T. (2024) 'A Systematic Review of Rapidly Exploring Random Tree RRT Algorithm for Single and Multiple Robots', Cybernetics and Information Technologies, 24(3), pp. 78-101. Available at: <https://doi.org/10.2478/cait-2024-0026> (Accessed: 19 September 2024).
- [4] Fan, H., Huang, J., Huang, X., Zhu, H., and Su, H. (2024) 'BI-RRT*: An improved path planning algorithm for secure and trustworthy mobile robots systems', Heliyon, 24(e26403). Available at: <https://doi.org/10.1016/j.heliyon.2024.e26403> (Accessed: 10 October 2024).

11. Appendices

11.1 Supplementary Data

11.1.1 Source Code Repository (GitHub/GitLab)

Reference link to the source code.

11.1.2 ROS2 + Unity Configuration Details

Technical setup instructions.

11.1.3 Raw Simulation Results

Unprocessed data from testing and evaluation. Ensure that all mathematical expressions are properly enclosed within `'...'` for inline math or `'`

...

`'` for display math. For example, write `' $x + y = z$ '` instead of `'x + y = z'`.