Technical Design Report

GBDP

By

Theodore Bedos – tb777

30/05/2022

Executive Summary

The following paper describes the design and development process of a Local Planning algorithm to be implemented on the autonomous software pipeline of Team Bath Racing electric (TBRe) autonomous race car. The development of this pipeline is part of a partnership between Bath Dynamics and TBRe. The aim of this partnership for Bath Dynamics was to develop a prototype software for their project of autonomous robot – Paletron – while considerably reducing the cost of acquiring hardware for testing. In exchange TBRe would obtain for free a team of seven engineers working full time on developing their autonomous pipeline. For both team it was primordial that a working pipeline was developed as early as possible. Thus, the main requirement of the Local Planning algorithm design process was to build a system that is integrable with the rest of the systems and react to dynamic conditions, regardless of the algorithm performances.

TBRe autonomous race car will compete in Formula Student UK (FSUK) driverless. The competition is made of different events, each having specific characteristics and rules. Local Plannning has a major importance in the Autocross/Sprint and Trackdrive events. These events favour reliable performance and safety trajectory over fast race time. The design requirements were based on these factors.

Multiple design alternatives were investigated. Including different discretization methods, path finding algorithms and implementation methods. The design selected consist in discretizing the environment using Delaunay Triangulation and using characteristics of wrong paths to constrain an A* algorithm.

Two algorithms were developed to test the algorithm. *TrackReader* to easily produce track setups from drawing and *Simulator* to reproduce conditions. The algorithm was then tested in static conditions given ideal and unideal inputs. And via *Simulator* to confirm if the algorithm can be implemented on the car.

The algorithm was able to build a valid path in every type of curve and most unideal inputs. It was also able to complete one lap in dynamic conditions. The average computational time of 0.2 sec allowed a maximum car speed of 4.2 m/s in a complex circuit.

In regard to the requirement, the design was considered successful. However, the performance of the algorithm would not allow TBRe to obtain a high score in the competition. Major improvements could be made to improve reliability and computational speed. Implementing a multiple branched algorithm such as RRT* would probably improve the performance of the design.

Content

1 – Overview

1.1 – Background

Bath Dynamics is a company that specializes in developing autonomous robots for industry applications. The differentiation factor of the company is the capacity of its robots to autonomously evolve in unknown environments without the help of guiding infrastructure. Thanks to this feature, Bath Dynamics' goal is to provide solutions for the many industries where implementing guiding infrastructure is impossible due to the temporary nature of the applications, vast outdoor spaces, or dynamic environments. Industries, such as the military and disaster relief pop-up warehouses with autonomous fork-lift (Paletron), skiing resorts with autonomous snowcats and agriculture with autonomous farming vehicles are the primary targets of the company.

The cornerstone of the project is Bath Dynamics' autonomous software pipeline. The pipeline will have the ability to detect obstacles and build a safe trajectory around them. The pipeline will keep a memory of the obstacles' position and create a map of the environment. Then, the map will allow the pipeline to build optimized trajectories through the sections of the environment that have previously been explored.

Due to the multiple similarities between Bath Dynamics' pipeline and the pipelines of autonomous racing cars for the Formula Student UK (FSUK) competition, Bath Dynamics will work on developing its pipeline in partnership with the student racing team Team Bath Racing electric (TBRe). This partnership will allow Bath Dynamics to cut the cost of acquiring hardware for testing the pipeline as the company will have access to TBRe's racing car. Thanks to this partnership, Bath Dynamics will develop the prototype of the pipeline that will be implemented on Bath Dynamics's future autonomous robots. With such a working prototype, the company expects to convince potential investors to invest in the company's next project Paletron. On the other hand, TBRe will obtain a high performing pipeline for free. Therefore, as part of the partnership between Bath Dynamics and TBRe, the following paper focuses on the development of TBRe's autonomous software pipeline.

1.2 - FSUK Driverless:

FSUK is an international student racing car competition hosted every year in the UK at the Silverstone circuit. There are four categories of competition, thermic, hybrid, electric and autonomous vehicles. The competitions are divided into two disciplines, Static and Dynamic. The Static events assess the teams' design, manufacturing, and business and the Dynamic events assess the car performances on the track. Bath Dynamics will be involved in the development of TBRe autonomous car and will focus only on Dynamic events.

In every event, the track's borders consist of cones, blue for the right border, and yellow for the left border. A pair of orange cones represent the start/finish line. The track has a minimum width of 3m and the maximum distance between two border cones is 5m [2].
There are four Dynamics events. Each event has a different track layout, rules and scoring points that have to be considered when designing the pipeline. The events consist in:

- Skidpad: Two laps in two pairs of concentric circles in a figure of eight patterns. The width of the track is fixed, and the exact radius of the circles and the number of cones are known in advance.
- Acceleration: 75m straight-line acceleration. The width of the track and the number of cones are fixed and known in advance.
- Autocross/Sprint: Two attempts to complete one lap of an unknown track.
- Track drive: One attempts to complete ten laps of an unknown track.

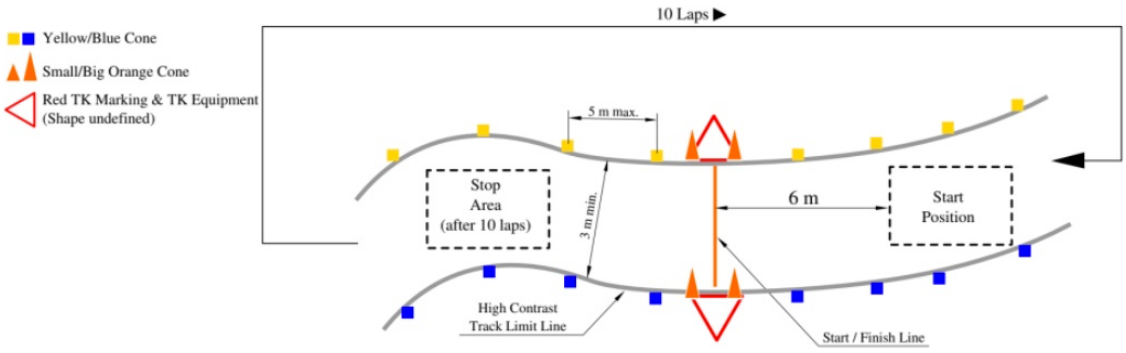


Figure 1: Track layout of a TrackDrive event [1]

1.2 – The Design Task

The software pipeline's role is to find a path/trajectory around obstacles. In TBRe's racecar, the software pipeline receives information about the environment from the sensors and sends the path found to the control system. The control system will then determine the required sequences of orders to be sent to the actuators in order to follow the path.
Depending on the event, the software pipeline has more or less impacts on the performance of the car. Indeed, in the Skidpad and Acceleration events, the tracks are precisely defined and known in advance. The optimal trajectory is known, and the challenge consists of designing a control system and hardware capable of following it. While the software pipeline's role is limited to detecting the endline. Thus, the pipeline software design will be focused on the Autocross/Sprint and Trackdrive events where the track is unknown.
To operate in these events, the pipeline has to be capable of three things – perception, localization and planning:
e

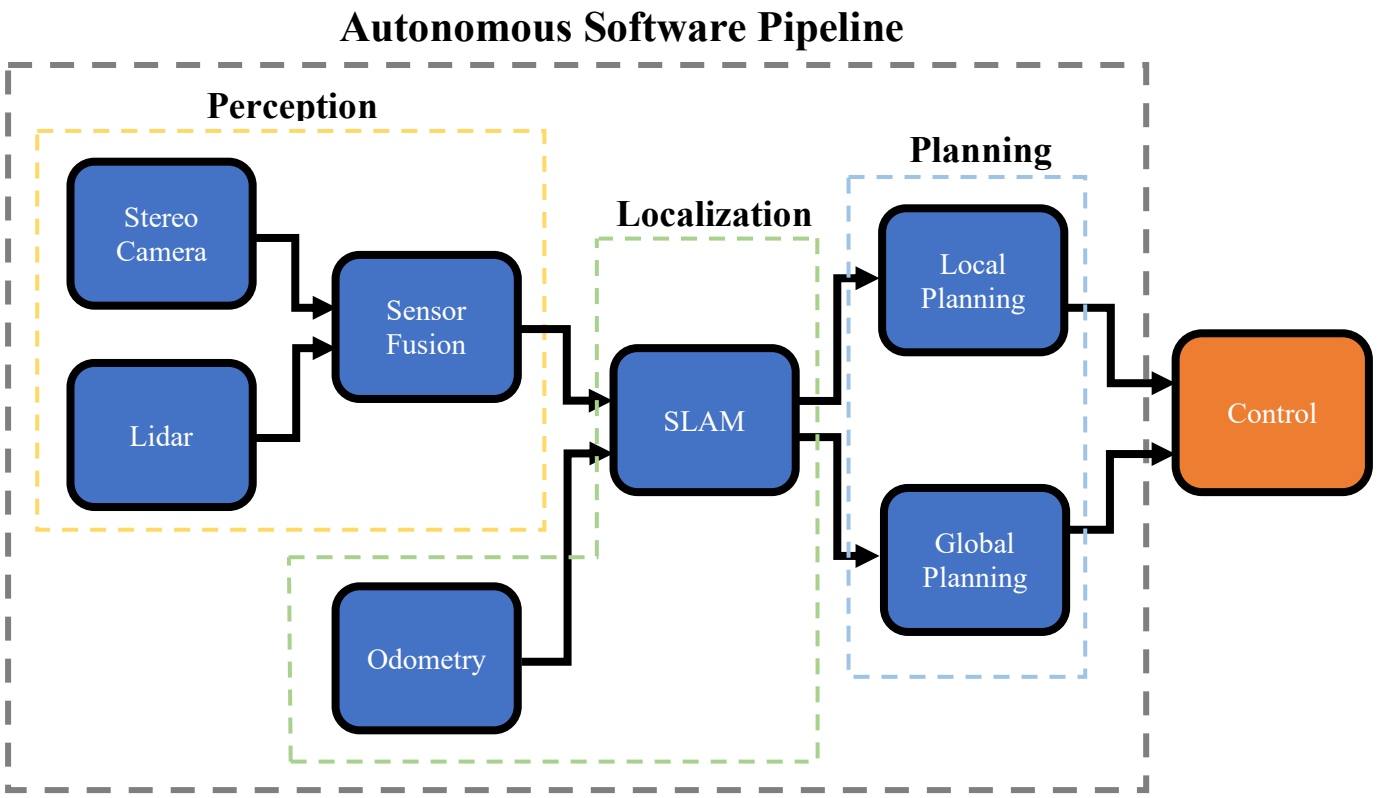
- Perception: The software pipeline needs to be able to detect the position and colour of the cones. To do so, the car is equipped with two different sensors, a LiDAR and a Stereo Camera. The data obtained from the sensors is treated by the so-called LiDAR and Stereo Camera sub-system and then merged by the Sensor Fusion sub-system.

- Localization: To produce the most optimized trajectory, the software pipeline must obtain a complete map of the track and be able to position itself in it. Using the information on the local environment from the perception systems, the SLAM sub-system will progressively build a map. Then, given the car orientation obtained from the Odometry system and the distance relative to cones that have previously been encountered, SLAM will locate the car within the map.

- Planning: According to the specificities of the Autocross/Sprint and Trackdrive events, the pipeline will have two different modes of operation Local Planning and Global Planning. The first one is used in the Autocross event and in the first lap of the Trackdrive event. When the tracks are unknown, only local information about the environment is available. So as the car moves, the uncomplete SLAM map is constantly being updated and a new trajectory needs to be found. The second one is used after the completion of the first lap of Trackdrive. At that point, the track is no longer unknown and so a faster trajectory can be found throughout the whole map.

The different subsystems are integrated together using ROS2. The input and output of each sub-system are presented in the following table

Roles	Sub-Systems	Input			Output		
		Source	Type	Content	Destination	Type	Content
Perception	Stereo Camera	Stereo Camera sensor	Image	Environment data	Sensor Fusion	.csv file	Cones coordinates (x, y, z), cones colour
	Lidar	LiDAR sensor	Point cloud	Environment data	Sensor Fusion	.csv file/Point cloud	Cones distance to the car and colour
	Sensor Fusion	Stereo Camera, LiDAR	.csv file	Cones coordinates (x,y,z), cones colour	SLAM	.csv file	Cones distance to the car, cones coordinates, cones colour
Localization	Odometry	IMU sensor	SbgImuData, SbgGpsPos, SbgGpsVel	Car data	SLAM	Geometry_msg	Car orientation, velocity
	SLAM	Sensor Fusion, Odometry	.csv file/ Point cloud	Cones distance to the car, cones coordinates, cones colour, car orientation. May include the variance of the cone coordinates, colour probability.	Local Planning, Global Planning	.csv file	Cones coordinates (x,y), cones colour, car position, car orientation
Planning	Local Planning	SLAM	.csv file	Cones coordinates (x,y), cones colour, car position, car orientation	Control System	.csv file	Trajectory points, borders points, car position, car orientation
	Global Planning	SLAM	.csv file	Cones coordinates (x,y), cones colour, car position, car orientation	Control System	.csv file	Trajectory points, borders points, car position, car orientation



2 – Design Specification

2.1 - Problem statement

According to the segmentation of the whole pipeline described previously, the Local Planning section of the pipeline is in charge of finding a valid path during the first race lap of the Trackdrive event and during the totality of the Autocross/Sprint event as it consists of only one lap. During this lap the environment is unknown, and data is obtained from the SLAM system.
According to the scoring rules of the two events [ref]. Points are awarded based on the car's time, but also on the ability to finish laps consistently and safely. Penalties are given when the car hits cones or goes out of track. Thus, in general, greater scores are obtained when the designs favour reliability over speed. It is confirmed by the results of the edition of the competition held in 2018, where the best scores were obtained by the only three teams who completed the Trackdrive event [ref].

Moreover, for Bath Dynamics and TBRe, it is primordial that a working pipeline is developed as rapidly as possible. In the unlikely event that Bath Dynamics is unable to produce a working pipeline by the end of the 3 months partnership or does produce it lately, TBRe will not be able to compete or will have poor results due to insufficient testing. Thus, Bath Dynamics will have no or poor results to present to potential future investors and no guarantee that its next project – Paletron - is feasible. Therefore, the major requirement of this design is the ability to rapidly produce a design that can be implemented on the pipeline. The Minimum Viable Product (MVP) is based on this requirement.

2.2 – Requirements

Based on the problem statement and the whole pipeline requirements, the following requirements for the Local Planning system were defined:

Number	Category	Weight	Requirement	Validation Criteria	Reason
1	Design	5	Working MVP by the 30/05/22	The system can be implemented successfully on the car by 30/05/22. The performance does not matter	Obtain a prototype to present to investors
2	Design	4	Reliability	The system consistently produces a valid path. The system does not plant	Points are awarded for being able to complete the event
3	Design	3	Prefer safe trajectory	The trajectory closely follows the centerline of the track	The time lost during a safe 1st lap is less damaging than the penalties for hitting cones.
4	Design	3	High computational speed	The design is optimized to improve computational speed	Fast computational speed allows a faster reaction time. Thus, the car can go faster and is quicker to recuperate a bad trajectory.

5	Design	4	Smooth trajectory	The system outputs a trajectory that can be followed by the car. The angle change between the point of the trajectory is small	If the trajectory has to be changed too much by the control system to match the car capacities. It might not avoid cones anymore
6	Technical	5	Use the SLAM output as Input of the system	The system operates using information contained in the .csv file outputted by the slam system	To be implemented on the car, the system must comply to the other systems requirements. Implementation on the car is the main requirement
7	Technical	5	Output a .csv file	The system writes its outputs in a .csv file	To be implemented on the car, the system must comply to the other systems requirements. Implementation on the car is the main requirement
8	Technical	5	Must output a trajectory as set of points defined by x and y coordinates	The system outputs the x and y coordinates of each point of the trajectory on a different line of the output file	To be implemented on the car, the system must comply to the other systems requirements. Implementation on the car is the main requirement
9	Technical	3	Output the x and y coordinates of the border cones, and inform to which border they belong	The system outputs the x and y coordinates of each border cone on a different line of the output file. The cones have a colour tag to differentiate them.	To be implemented on the car, the system must comply to the other systems requirements. Implementation on the car is the main requirement
10	Technical	5	Output the car position and orientation	The system outputs the last car position and orientation.	To be implemented on the car, the system must comply to the other systems requirements. Implementation on the car is the main requirement
11	Technical	5	Operate on ROS2	The system is design using Python 2.7 or C++	To be implemented on the car, the system must comply to the whole system requirements. Implementation on the car is the main requirement
12	Technical	4	Responsible of its own refresh	The system autonomously takes the decision to compute a new trajectory	To operate in dynamic conditions

2.3 – Tasks and Prioritization:

As a first approach, it was determined that the system will require the following functionalities. Not all functionalities are needed for it to operate. Some would allow better performance or reliability. The order of implementation is determined by the required effort and necessity of each functionalities.

1- Generate a valid path inside a track where every cone has been clearly identified
2- Generate a valid path inside a track despite missing cones
3- Generate a valid path given dynamic conditions
4- Read/extract data from .csv file from SLAM
5- Write in a .csv file for the control system
6- Compute car orientation through the trajectory
7- Take count of cones' position variance
8- Take count of cones' colour likelihood
9- Control the refresh frequency
10- Ouptut right and left borders
11- Detect when reaching the starting point again

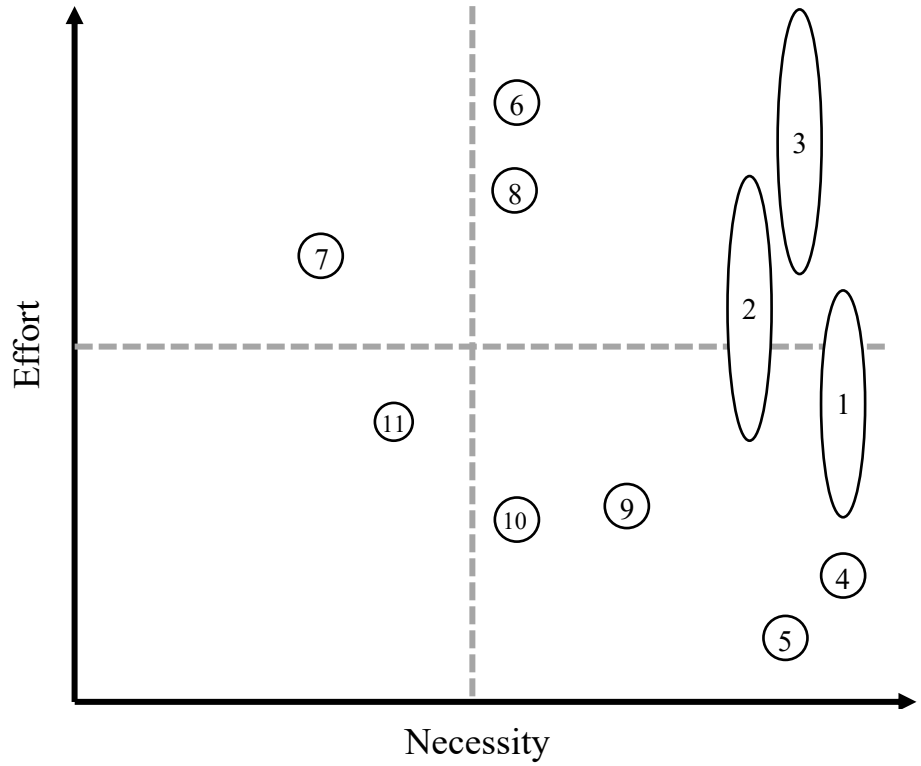


Figure 1: Prioritization chart of the functionalities of the system

Using the prioritization chart shown above, it was determined that the first functionalities to implement are 1, 2, 4 and 6. They will allow testing the algorithm in static conditions where the input can be controlled. In a second time, functionalities 3, 5, 9, 10 and 11 will allow the program to interact with the other systems. If implemented correctly, it will be sufficient to complete the MVP requirement. Finally, functionalities 7 and 8, will be implemented to improve the performance an reliability of the system.

3– Design Alternatives

3.1 – Alternatives review

As mentioned previously, Local Planning aims to find a path/trajectory around obstacles given an incomplete map of the environment. As the SLAM map is incomplete the path produced is relatively short compared to the total length of the track. Also, the car is moving, thus, Local planning needs to frequently generate new trajectories based on the updated map. The algorithm will have to be fast, reliable and realistic.

Considering the specificity of the challenge, many algorithms have been investigated. The following

3.1.1 - Discretization

The space around the car consists of an infinite number of points and each of these points is a potential future position. Considering each point requires a tremendous amount of time and computational power, therefore, the space can be discretized to obtain a finite number of points. The level of discretization has a direct impact on the precision of the path and the computational time. The smaller the spatial step between points is, the smoother the trajectory will be, however, the longer it will take to link each of the points that make the path. Inversely, with large spatial steps, potential paths are investigated faster but the angle changes in the trajectory might become unfeasible for the control system.

i) Delaunay Triangulation

Given a set of discrete points, the Delaunay Triangulation method generates triangles with the points being the vertices, so that no vertex lies inside the circumcircle of any triangle. As the triangle vertices are cones, the centers of the triangle edges can represent safe points where the car can travel without hitting cones. The centers can be used to create a new set of discrete points that are potential future spatial states of the car.
This considerably limits the number of potential trajectories. Given a limited amount of points the best trajectory possible will be sub-optimal, however, as cones are placed closer to each other in curves than in straight lines, the spatial step between each point variates. Thus, fast computational efficiency is obtained in straight lines, while relatively smooth trajectories can be achieved in curves.

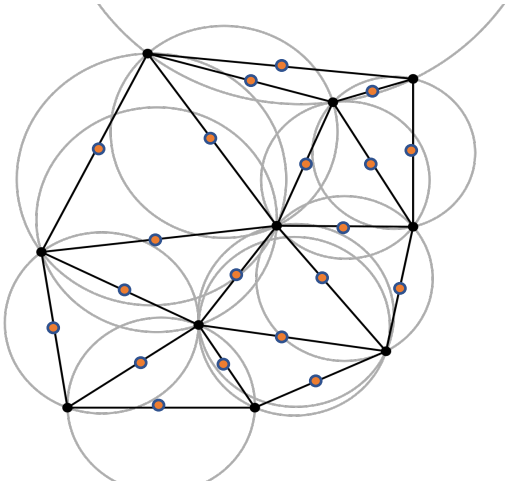


Figure 2: Delaunay triangulation applied on the black points. Red points representing the centers of the triangles' edges

ii) Cones Pairing

Linking each cone from one border with the closest cone on the other border creates multiple segments whose centers are on the centerline of the track. This new set of points has the advantage to be exclusively inside the track boundaries. And as for the Delauney Method, the spatial step variate depending on the curvature of the track. However, there is no guarantee that each border will have the same number of cones, nor that the same cones will be visible at the same time. This could be overcome, by extrapolating each border with the visible cones and linking the same number of pseudo points from each of the extrapolated borders. Nevertheless, if cones from other segments of the track are detected, it will false the extrapolation. Also, this highly depends on whether the color data is 100% reliable when realistically the cone color is associated with a likelihood percentage.

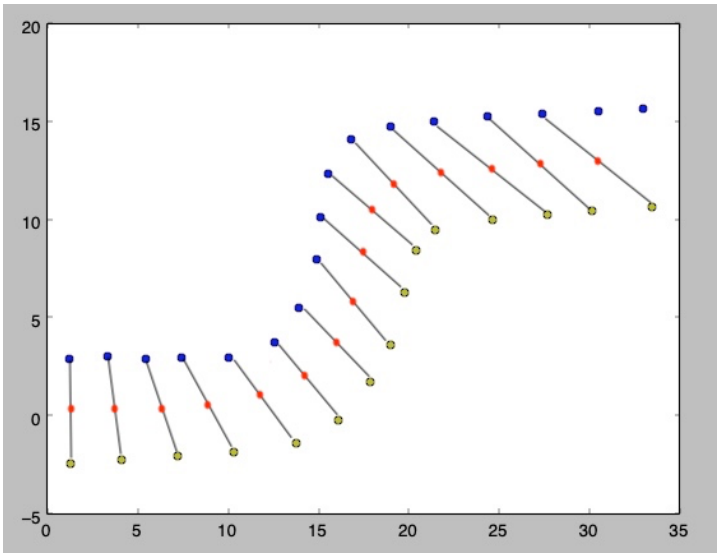


Figure 3 : Cone pairing discretization method. Blue and yellow points are cones. Red points are ce center of the pairs.

iii) Gridding

A grid can be used to divide the space into multiple cells, the coordinates of each cell being its row and column. Each cell contains a cone filled with a 1 and each empty cell that represents a potential future state is filled with a 0. As explained previously, the spatial step is correlated to the computational speed of the algorithm, it is particularly impactful for this method. The main issue is the progression from one cell to another which can lead to a 90° angle change. As the control is affected by the smoothness of the path [3], such angle changes will have to be smoothened using extra work.

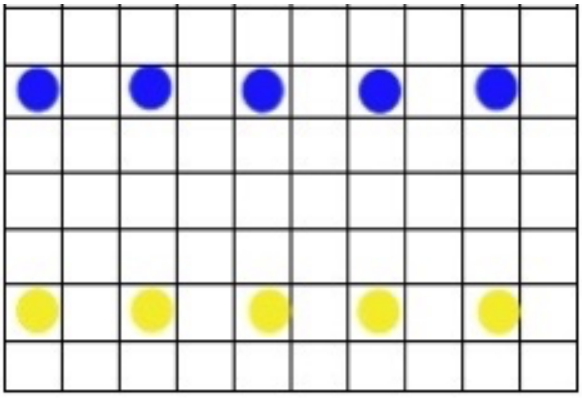


Figure 4 Grid discretization method. Blue and yellow points are cones

3.1.2 – Path Finding

*i) A**

The A* algorithm considers the point surrounding it. Each point is weighted depending on criteria, generally the sum of its distance from the start-point and from the end-point. The point with the lower weight is then selected and the process is repeated with this new point until the end-point is reached. The trajectory found is optimal, but if the algorithm is stopped before the end, an incomplete path is obtained. The path might be just about to reach a dead, thus there is no guarantee that the path leads to the end-point until the algorithm is completed.

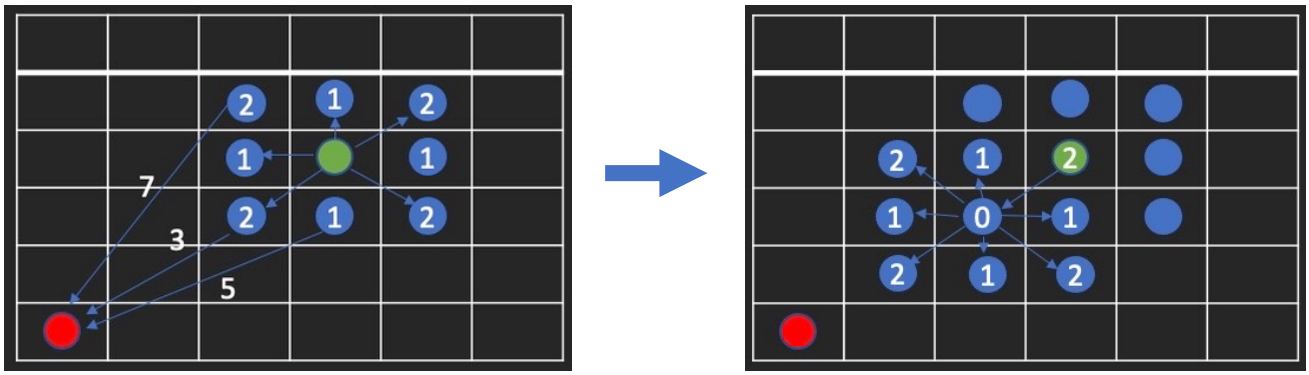


Figure 5: A algorithm using the distance to the end point as the selection criteria*

*ii) Rapid Random Tree**

The RRT* algorithm consists of randomly selecting points within a defined parameter. The point is added to the graph if it is possible to link it to another point without crossing an obstacle. The point is linked to the closest point in the graph so that progressively a tree of potential trajectory is built (see figure 6). Eventually, the endpoint is randomly selected and added to the graph. The trajectory to the end-point is initially sub-optimal, but as more points are added to the tree, the trajectory is refined. RRT* allows to quickly obtain a viable root, however, it requires a significant memory to store every branch of the tree.

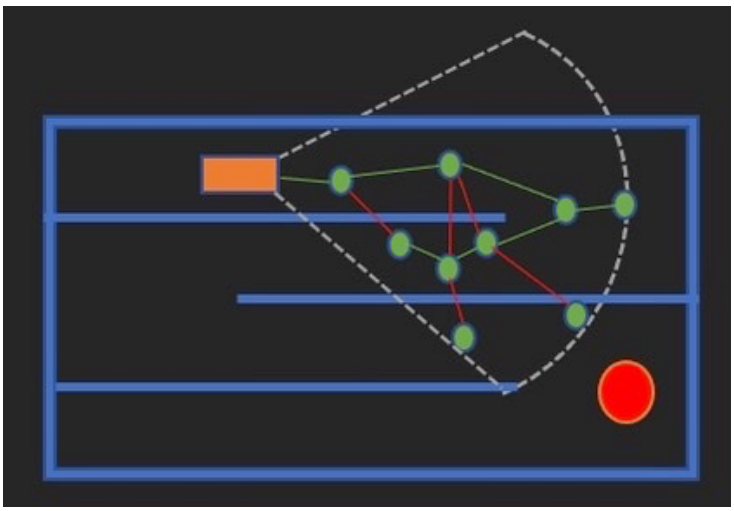


Figure 6: RRT algorithm in a maze. The red lines represent a connection that is not possible due to the borders. Green line are valid branches.*

3.1.3 – Implementation Method

One of the main challenges of building a path in an unknown environment as the racetrack is the absence of an endpoint. Most path-finding algorithms build a path between a start point A and an endpoint B. In the absence of an endpoint, new criteria, or rules other than reaching the endpoint need to be found. Two main strategies have been investigated to overcome this issue. The following alternatives are tuned variations of A* and RRT* in order to suit the strategy.

i) Constrained research

Some characteristics of wrong paths have been identified and can be used to constrain the search algorithm. Starting from the position of the car, potential future position points are filtered out according to these criteria. When a point meets all criteria, it is added to the trajectory. The following criteria are used as filters:

- *Distance from the car*: To avoid traveling too far in the wrong direction, the local planning algorithm is frequently run at fixed intervals. Considering a maximal speed of 15m/s according to the control system [ref], and a running interval of 100ms (discussed in section), the car will travel 1.5m until a new trajectory is obtained. Thus, generating paths that are too long is a waste of effort as they will be updated before the car has entered a different type of curve. On the other hand, paths that are too short are too similar to each other to allow the control system to adequately adapt to the speed. Therefore, potential points are rejected if they are more than 10m away from the car.

- *Angle to the car*: As it is very unlikely that the desired trajectory is directed in the opposite direction than where the car is heading, it is preferable to filter out points that are not in a 180° range in front of the car. A smaller angle would correspond better to the car's field of vision but in the eventuality of a pinhead curve, it is desired to consider points that are not just straight in front of the car. As it can be seen in figures XX and XX it allows for building a path that more accurately describes the situation for the control system.

- *Crossing a border*: The ideal path must stay within the border of the track. If reaching a potential point makes the path cross a border, then this point should not be added to the trajectory.

As every potential point has to be tested through multiple criteria, this method can be very slow if the number of potential points is high. The following methods work particularly well with the Delaunay Triangulation method to reduce the number of point.

Delaunay/Constrained A:*
This method is in fact a tuned version of the A* algorithm. Every surrounding point is investigated but instead of using the distance to the end-point to select the next trajectory point, it is the point that passes all the filters that is selected. If multiple points pass the filters, the closest point is selected. The selected point becomes the new reference point, and the process is repeated.

Delaunay/ Constrained RRT:*
This method can also be used with RRT*. Points are randomly selected and tested through the filters. Contrary to A*, not all the points need to be tested before adding a new point to the trajectory. When a point passes all the filters it is directly added to the trajectory. As more points are added to the trajectory, points that did not pass the *crossing a border* criterion can now be connected to the trajectory points without crossing a border.

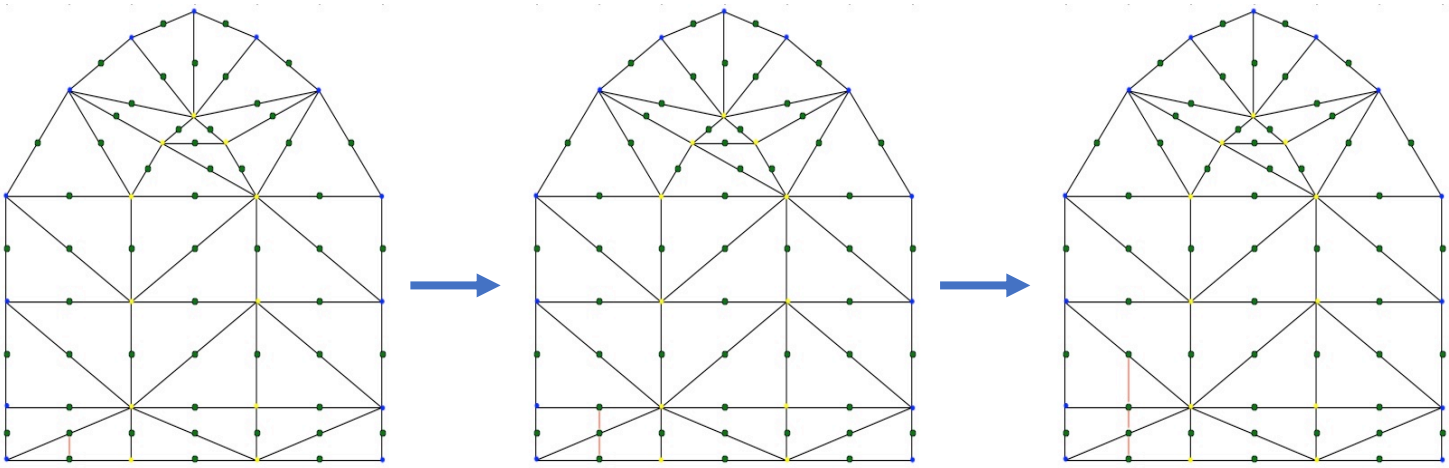


Figure 7: Step by step construction of a path using the "Delaunay/constrained A" method*

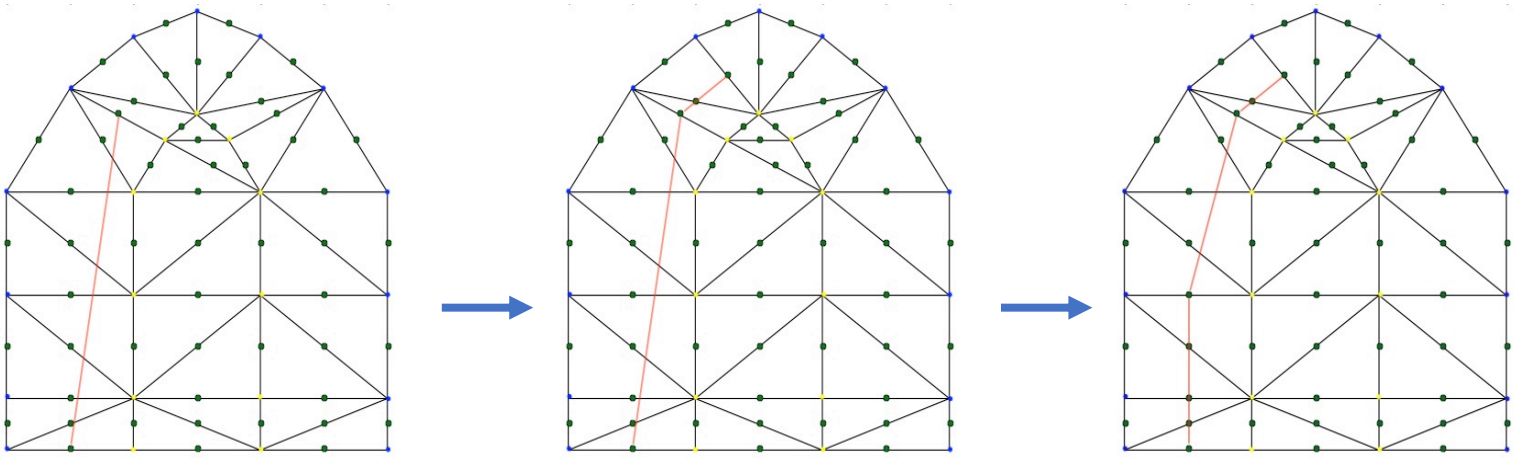


Figure 8: Step by step construction of a path using "Delaunay/constrained RRT" method*

3.3.2 - Weighted selection

In the previous method, a unique path was built, assuming that every point that is added to the path is valid. In this method, the strategy consists of building multiple paths and selecting the best alternative. The paths are weighted according to criteria based on the characteristics of valid paths, then the lowest weight is selected. The criteria are:

- *Distance from borders*: For safety in the first lap, it is desired to stay close to the center of the track. The deviation from the centerline can be assessed using the average of the difference between the distance to the closest blue and yellow cone at each point on the trajectory.

- *Cones on both sides:* The deviation from the centerline can also be assessed by checking for each trajectory point if cones are present on both sides of the car.

- *Maximum angle change:* Unless the car is trying to recover from an unwanted position, it is very unlikely that the car will need to execute aggressive changes of direction to follow the track. Trajectories with small maximum angle change are more likely to be valid.

- *Length of the path:* The trajectory length should be relatively close to the perception range. Too long and too short paths are more likely to be invalid and should be weighted higher.

To build the multiple paths, the following strategies were considered:

Angle constrained RRT:*
This method is used with a continuous space. The first constrain consists of generating random points only in a 10m radius semi-circle parameter in front of the car. Then, when a point is generated, rather than checking if the point can be linked to an existing one without crossing a border, the algorithm checks the angle change between the two points. If linking the point, create an angle change superior to 20° in the branch, the point is not added (see figure 9). This strategy quickly generates potential paths. However, has the algorithm is meant to be restarted at fixed intervals, even the best path can meander. Also, storing all branches of the exploring tree can slow the algorithm if the restart interval is too long.

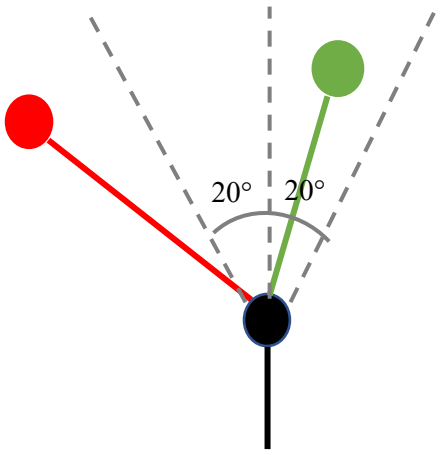


Figure 9: Angle constrained RRT method. The red point is not added to the tree. The green point is added to the tree*

Artificial End-point A:*
The endpoint can be simulated by placing multiple points at fixed intervals on a 10m radius semicircle (see figure 10). One of the end-point is very likely to be within the track. The A* star algorithm is used to build a path for each of the end-points. On each iteration, a point is added to each of the branches. This can be used with a grid discretisation or the Delauney Method. With A* the smoothness of the paths is highly dependent on the level of discretisation. And as multiple paths are built in parallel the effect on the computing speed is multiplicated. It is likely that the paths will be incomplete when the algorithm needs to be restarted.

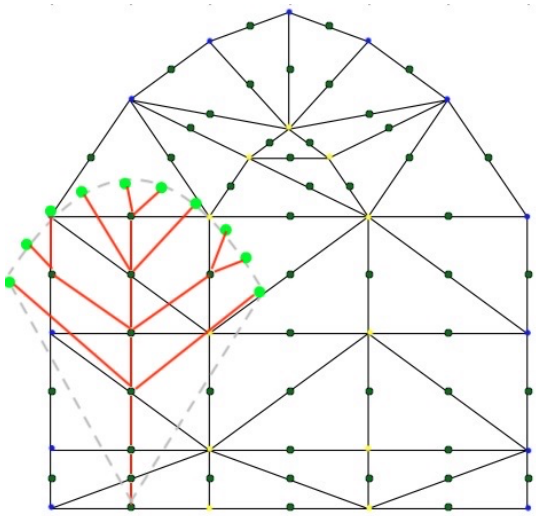


Figure 10: Artificial end-point with A and Delaunay discretization. The end-points are represented by the bright green points*

Systematic evaluation:
If the Delaunay Triangulation Method is used on the centres of Delaunay triangles' edges, it creates a new set of triangles whose edges are the transition between every centre point (see figure 11). Paths can be built relatively easily using Delaunay's library in python to find the connection between vertices. The path stops being built when a vertex has less than three connected vertices or when it returns to a previous vertex. As the number of centre points surrounding the car is relatively small, it is possible to quickly investigate most of the path if not all. When a path is generated, it is weighted, and if the weight is inferior to the minimum weight previously calculated it is stored as the best path. Thus, only one branch is stored at a time. If no history of the previous paths is created, it is possible that the best route is never found. However, it is very likely that the lowest weighted path will be good enough for the car to travel safely until the algorithm is rerun. If this becomes an issue, previous paths can be stored but investigating them every time a new path is built will slow the algorithm.

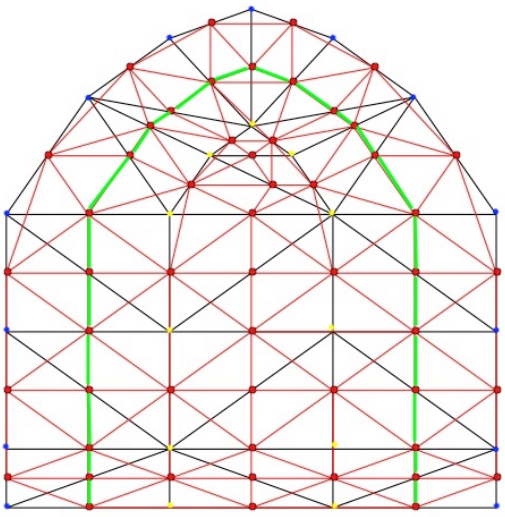


Figure 11: Systematic evaluation method with Delaunay discretization. All the possible paths are represented by red lines. The green line is the lowest weight path.

3.2 - Alternatives Selection

As mentioned previously, the main requirement of the design is to develop a working pipeline that can be implemented on the car as quickly as possible. The performance of the pipeline is of secondary importance. Therefore, the major criteria to assess alternative designs is the ability to rapidly produce the MVP.

In terms of the requirements defined in section 2.2, the weighted selection method seems optimal, especially the constrained RRT* and systematic evaluation algorithm. They are consistent, if the algorithm fails to select the best path, a path is always produced and so the car is never stuck. And they are fast as a sub-optimal trajectory is very quickly obtained. However, finding the formula to incorporate all the weighting criteria and tunning each of them to find the right weight requires a lot of testing. It is not guaranteed that the right combination will be found quickly or found at all in the absence of real testing. Therefore, according to the main requirement which is to rapidly produce an MVP this method can't be selected.

The constrained research method requires considerably less testing and as there is only one trajectory to deal with it is less complex, thus an MVP is likely to be obtained faster. On the other hand, these algorithms take longer to output a path. Also, they can be inconsistent due to the multiple assumptions that they rely on. For instance, it is assumed that the car will follow relatively closely the trajectory. But if the car deviates as it is represented in figure 12, then none of the potential points around the car will be within 10m, in front of the car and can be reached without crossing a border. This situation will result in no path being built and the car being stuck in this position. However, this position is unlikely to happen, smaller deviations can easily be retrieved, and it is reasonable to make such assumptions when delivering the MVP is the priority.

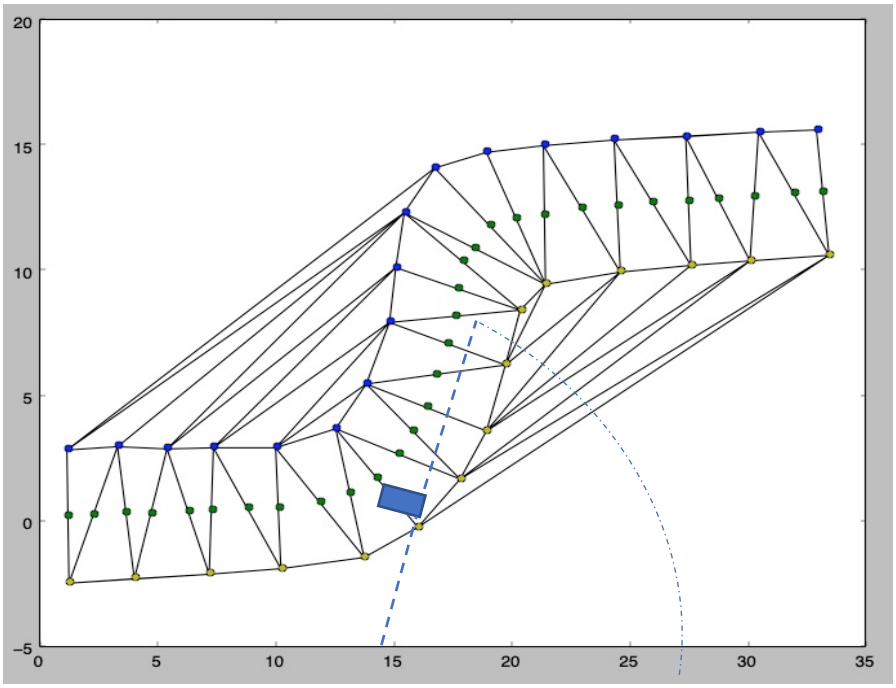


Figure 12: The car being stuck in position after deviating from the path. The potential trajectory points are the green points

Among the constrained research algorithms, the cone pairing/A* algorithm is discarded. It relies on unrealistic assumptions that can be tolerated only with basic track layouts. The two remaining algorithms Delaunay/A* and Delaunay/RRT* are equivalent in regard to the requirements. However, as the Delaunay/RRT* algorithm comports random elements, it will complicate the process of testing and debugging as the same sequence cannot be reproduced twice. Therefore, the Delaunay/A* algorithm is selected.

3.3 – Design Validation

Sub-systems of the software pipeline have been built separately and it has not been possible to integrate and test them together. Therefore, it is required to develop a testing method specific to the Local Planning sub-system to assess the viability and performance of the design.
Two programs were created in order to reproduce the input from SLAM. The first program called *TrackReader* transforms a track drawing into a .csv file similar to a complete SLAM input. Complete means that the file contains all cones' information as if the first lap had been completed. However, to closely reproduce the SLAM input during the first lap, the input file should only contain the cones that have been seen. Thus, based on the car position and orientation, the second program called *Simulator* uses *TrackReader*'s output to progressively add in a different .csv file the cones that have already been seen.

3.3.1 - TrackReader

Thanks to *TrackReader*, it is possible to quickly generate different track layouts by simply drawing them. Thus, the viability of the design can be tested through many different types of curves and tracks. The following figure 13 illustrates the result obtained with the program. The track drawing consists of blue and yellow dots representing the cones, a white dot for the start position of the car and a green line used as a scale. The length of the green line in meters is the only dimension known on the drawing as it is the width of the track. The length of the green line is thus hardcoded, generally as 5m, and allows to determine the dimension of the rest of the track.
The main operations of the *TrackReader* program are represented by the flowchart in figure 14. The legend can be found in figure 17 and the detailed code in appendix C.

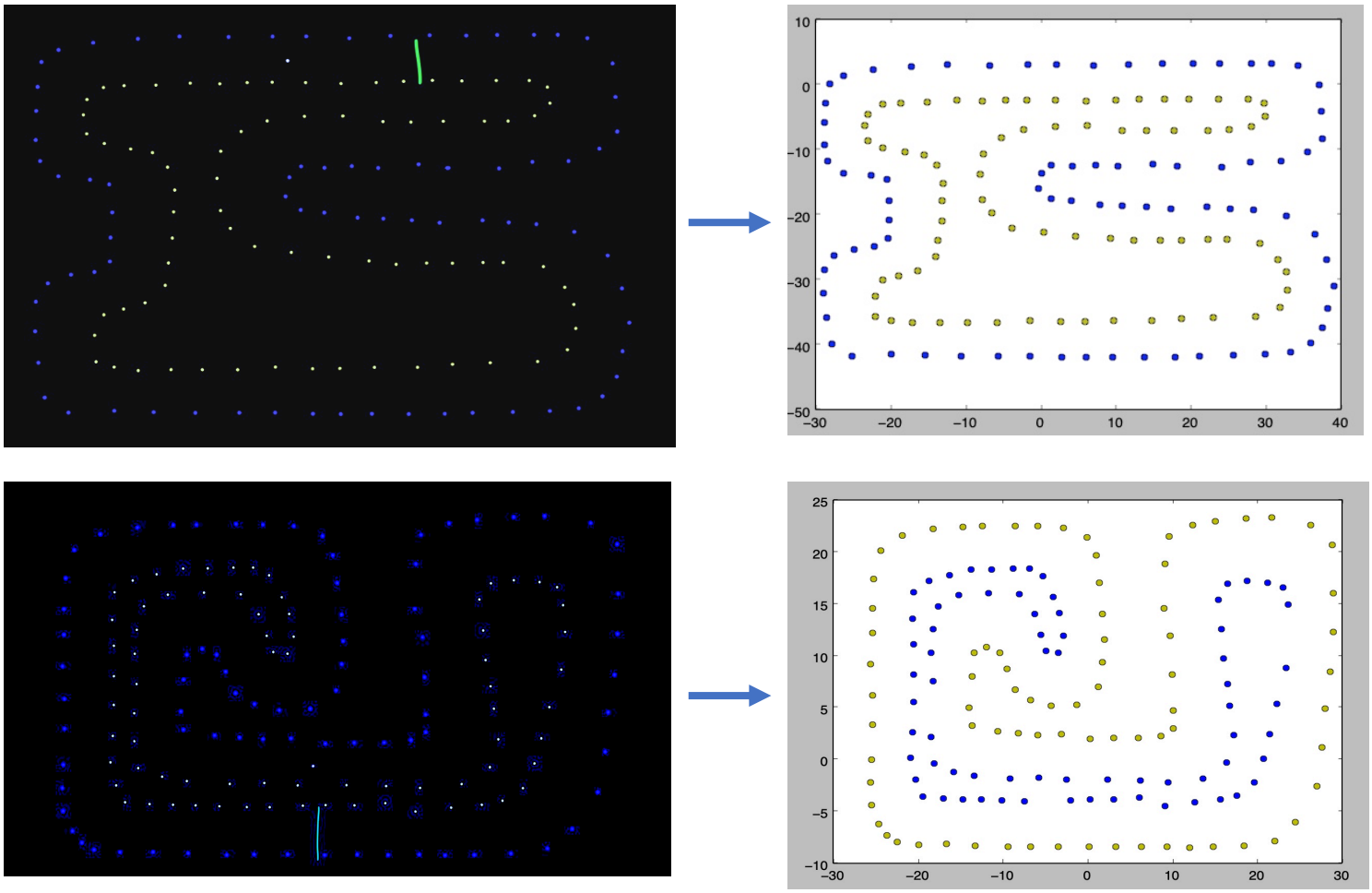


Figure 13: Track drawings converted into SLAM map using the program TrackReader

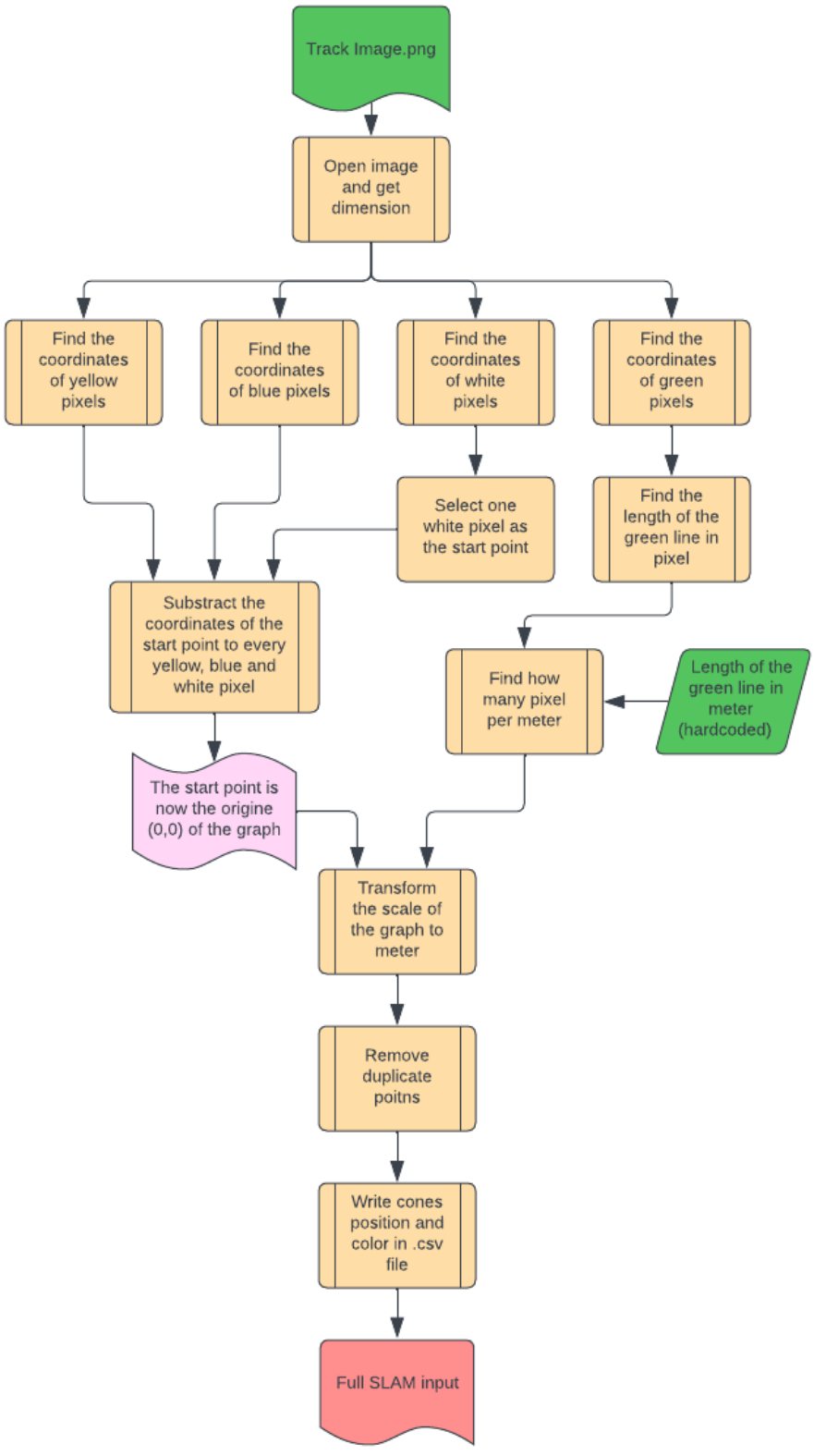


Figure 14: TrackReader program flowchart

3.3.1 - Simulator

The *Simulator* program allows testing of the algorithm's performance. By setting different car speeds, it makes it possible to see if the local planning algorithm is responding fast enough. And, by setting different levels of error of the car position and orientation, it makes it possible to test the algorithm's reliability.

The *Simulator* can be set to perform two types of tests. In the first configuration, the local planning algorithm is restarted at a fixed frequency. A restart frequency is defined, and a clock is used in the Local Planning algorithm to stop the algorithm when the time limit is reached. If the path isn't completed before the time limit, a shorter version of the optimal path is obtained. This version is still a valid trajectory and might be sufficient as the algorithm will be restarted before the car had travelled such a distance. Therefore, this setting allows testing of different restart frequencies and minimum path lengths. However, this configuration doesn't give feedback on the speed of the local planning algorithm. Thus, the second configuration consists of letting the local planning finish and recording the running time. That way, it is possible to analyze the algorithm efficiency in the different sections of the track and optimize the local planning algorithm accordingly.

The simulations produced by the *Simulator* program are shown in figure 15. The main operations of the *Simulator* program are represented by the flowchart in figure 16. The legend can be found in section 17 and the detailed code in appendix B.

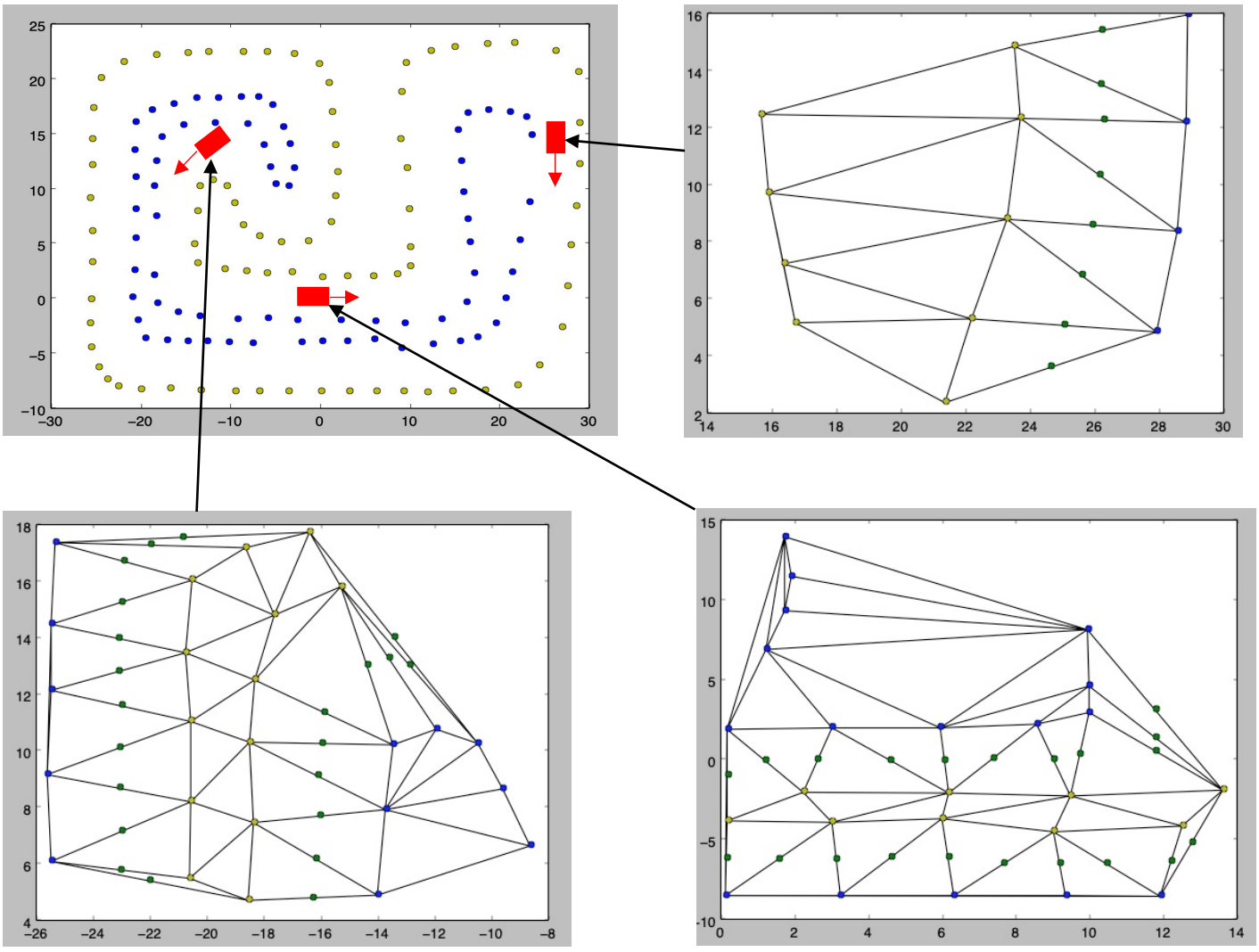


Figure 15: Local Map send to the Local Planning algorithm by Simulator at different point on the track

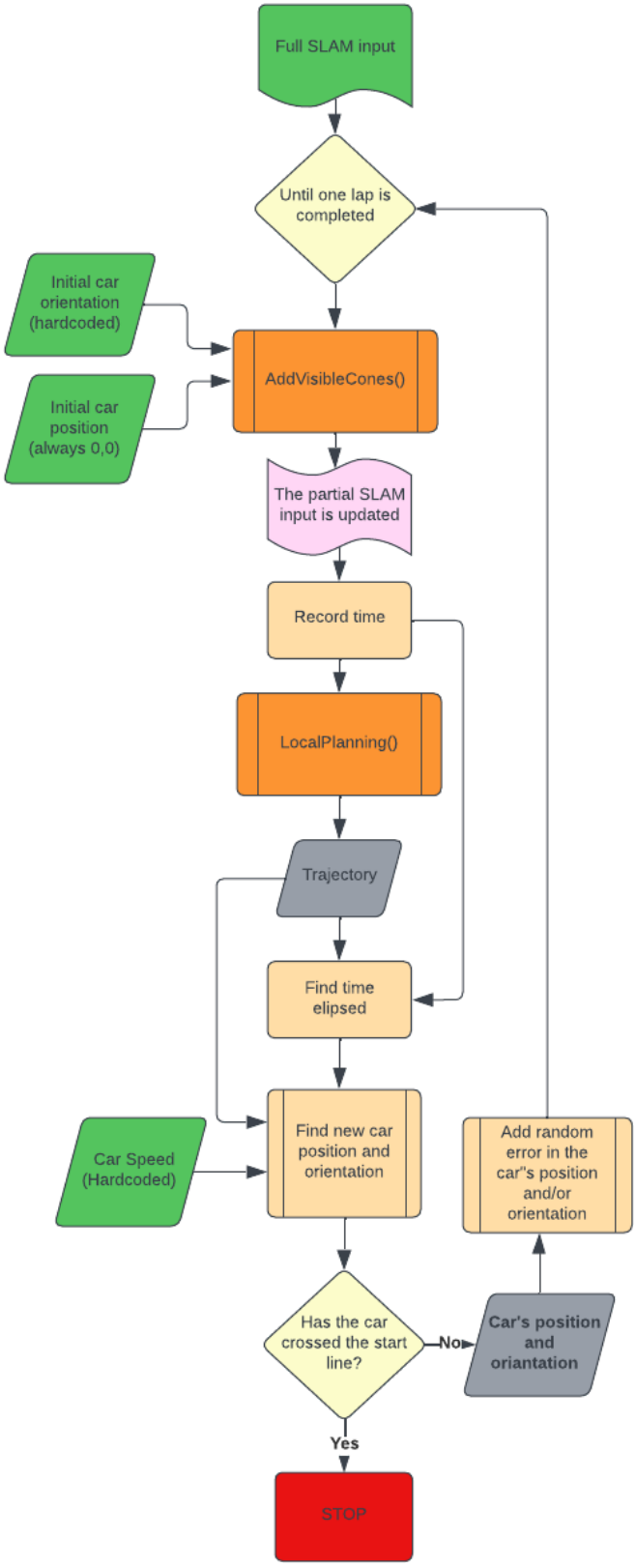


Figure 16: Simulator program flowchart

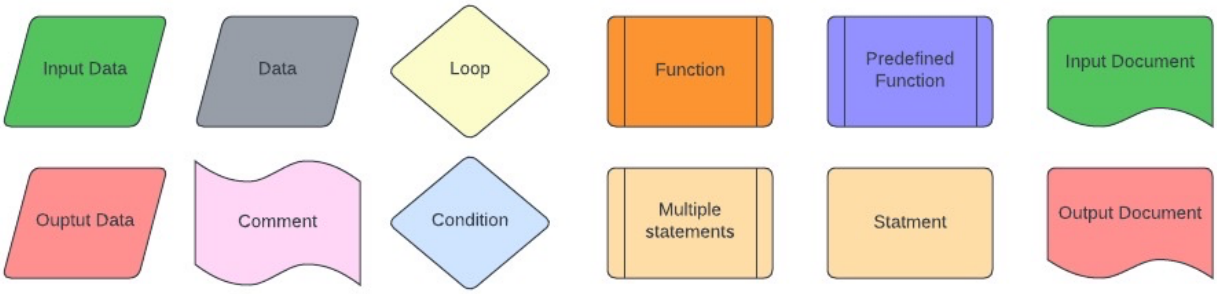


Figure 17: Flowcharts legend

3.4 - Test Protocol

According to the task prioritization of the sub-system defined in section 2.3, the design process consists of three phases. Different battery tests will be performed for each phase.

The first phase is the construction of a valid path in a static environment regardless of the computational speed. The environment could be ideal or with missing or incorrectly identified cones as it can be seen in figure 18, 19 and 20. To validate this phase, the program will be tested through different types of curves, such as straight lines (figure 19), hairpins (figure 7) and chicanes (figure 12). Different radii of the same type of curve, different track widths and different cone spacing will be tested. The algorithm will also be tested given a different full map of the track (figure 13, 21), to evaluate different sequences of curves. The tracks will be generated using *TrackReader*.

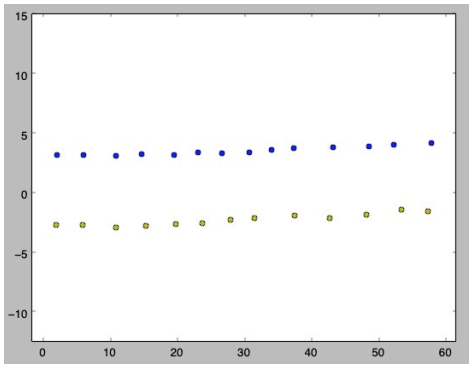


Figure 18: Straight line correctly identified

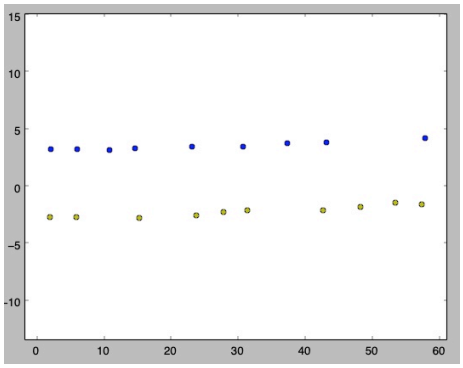


Figure 19: Straight line with missing cone

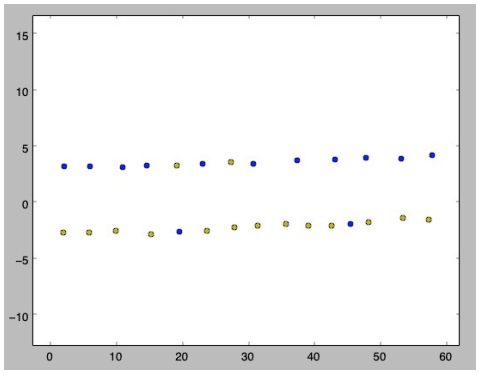


Figure 20: Straight line with cone of the wrong colour

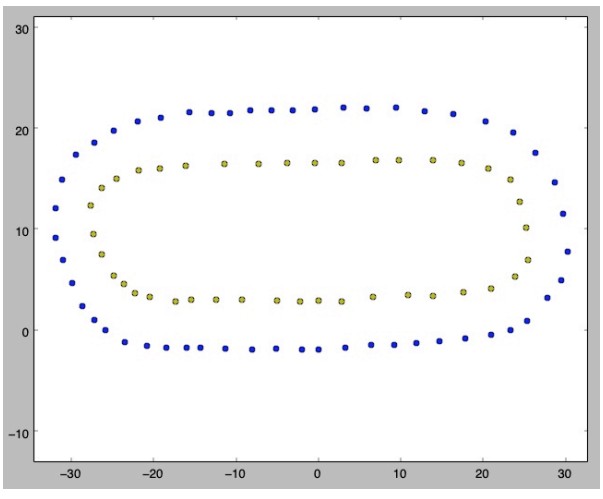


Figure 21: Complete track to assess a sequence of hairpins

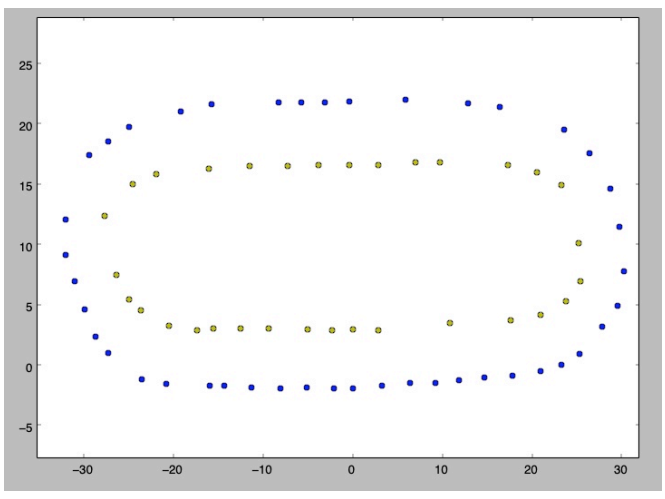


Figure 22: Complete track with missing cones

The second phase consists in making the program usable on the pipeline regardless of the performance. This means that the program is able to produce the right output and is able to operate in dynamic conditions. To validate this phase, the program has to be able to consistently complete one lap on different tracks using *Simulator*. *Simulator* is set on the "no frequency" mode so that the local planning algorithm can run completely without being interrupted. Then, car position and orientation are updated to correspond to the next position on the trajectory previously computed. The process is repeated until the car completes one lap or gets stuck in a position.

The final phase consists in improving the program performance. Using *Simulator*, the program's computational speed and reliability will be tested using different car speeds, restart frequencies, and levels of errors (position and orientation).

4 – The Final Design

4.1 - Implementation

This section describes the implementation of the "Delaunay/constrained A*" method selected in Section 3.2.

4.1.1 – Reading Input File

To be able to build a path around obstacles, the program needs to know where the obstacles are and where the car is. This information - the cone's position and colour, the car's position and orientation – is contained in the SLAM output file (see Figure 23 for details of the file). Thus, the first step in designing a Local Planning algorithm is to create a function capable of reading the file and extracting these data.

TextToArray() is the function that has been designed for this purpose. It opens the .csv file and reads it line by line. If the line contains the string "blue", "yellow" or "orange", the line describes a cone. If it contains "car", the line describes the car. The position of the commas in the line is used to determine where the information starts and ends. The different pieces of information are then stored in three different arrays. The first element of the output is a 2D array containing the x and y coordinates of the cones, the second element is a list containing the colour of the cones and the third is a list containing the car's actual XY coordinates and orientation.

Full_SLAM_Input.csv

```
tag,x,y,angle
car, 0, 0, 90
yellow,-0.588235294118,16.6666666667
yellow,-4.06862745098,16.568627451
yellow,-11.6666666667,16.5196078431
yellow,-7.45098039216,16.5196078431
yellow,-16.1764705882,16.3725490196
yellow,-19.1666666667,16.1274509804
yellow,-21.9607843137,15.7352941176
yellow,-24.5588235294,15.0980392157
yellow,-26.568627451,13.9705882353
yellow,-27.6960784314,12.3529411765
yellow,-27.5,9.46078431373
yellow,-26.6666666667,7.45098039216
yellow,-25.0490196078,5.39215686275
yellow,-23.9215686275,4.50980392157
blue,-0.490196078431,21.8137254902
blue,-3.23529411765,21.7647058824
blue,-8.33333333333,21.7156862745
blue,-5.78431372549,21.7156862745
blue,-15.8333333333,21.5196078431
blue,-13.0882352941,21.4705882353
blue,-10.8333333333,21.4705882353
blue,-19.2156862745,20.9803921569
blue,-22.1568627451,20.637254902
blue,-24.9019607843,19.7549019608
blue,-27.4019607843,18.5294117647
blue,-29.5098039216,17.3039215686
blue,-31.1764705882,14.9019607843
blue,-32.0098039216,12.0588235294
blue,-32.0098039216,9.11764705882
```

Figure 23: SLAM input file.csv

One advantage of the SLAM map is that it allows considering cones that have been seen but aren't visible when building the path. This is particularly useful in hairpins where very few cones are visible as it can be seen in Figures 25 and 26.

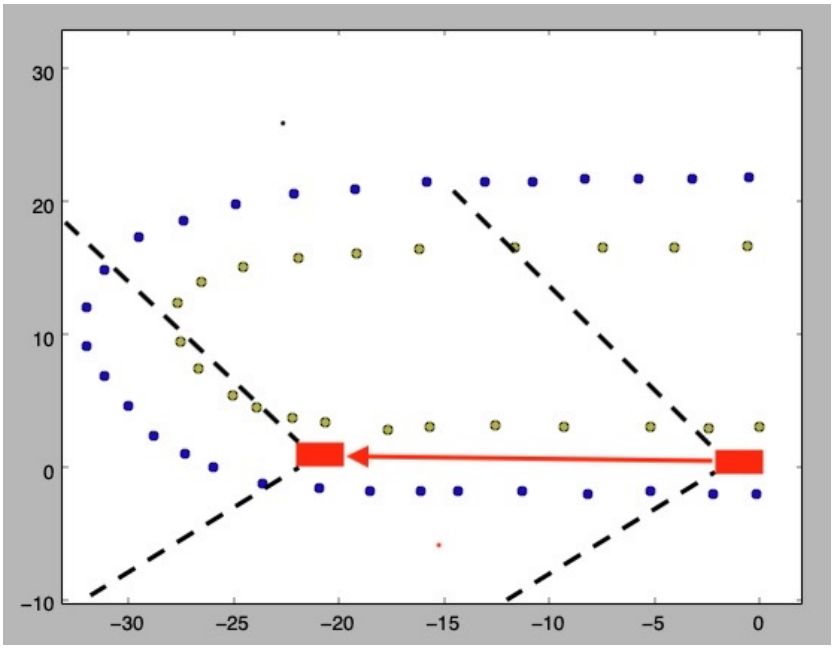


Figure 24: Shows which cone have already been seen by the car before attacking a hairpin

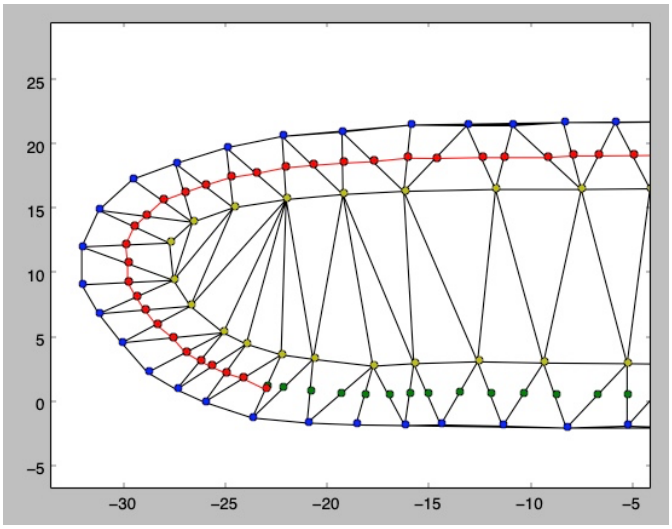


Figure 25: Path found using every cone that has been seen in an hairpin curve

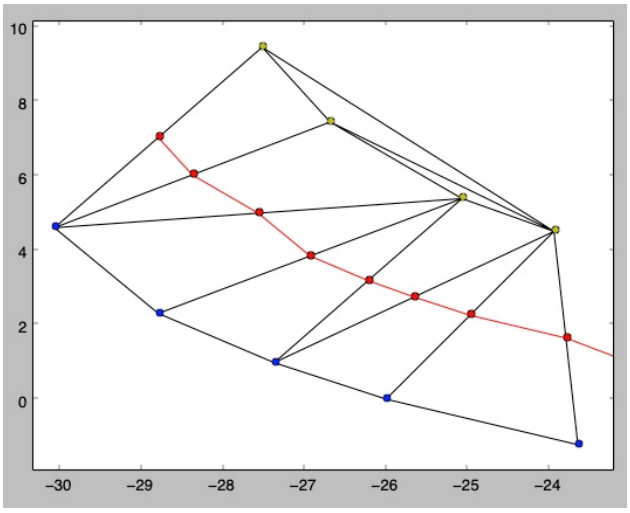


Figure 26: Path found using only the cones actually visible in an hairpin curve

However, as the car progresses on the track, more cones are discovered and added to the SLAM map. Thus, due to the nature of the constrained A* algorithm, it results in increased computational efforts of the path planning algorithm to consider every cone. Therefore, it was decided that *TextToArray()* would add only cones that are less than 10m away from the car and in front of it. This was achieved by designing the function *DistanceFinder()* and *AngleChange()*.

DistanceFinder() finds the absolute distance between two points. If the distance found when using the car position and the position of a cone in the SLAM input is superior to 10m, the cone is not added to the array. The distance is $\sqrt{x^2 + y^2}$, where x and y are respectively the difference of the points' coordinates on the X-axis and Y-axis.

AngleChange() finds by how many degrees the car orientation has to be changed to reach a point in a straight line. If the angle change is superior to 90°, it means that the point is behind the car and it not added to the array. How the angle change is calculated is illustrated in figure 27. In this system, the car orientation is 0° when the car is heading north.

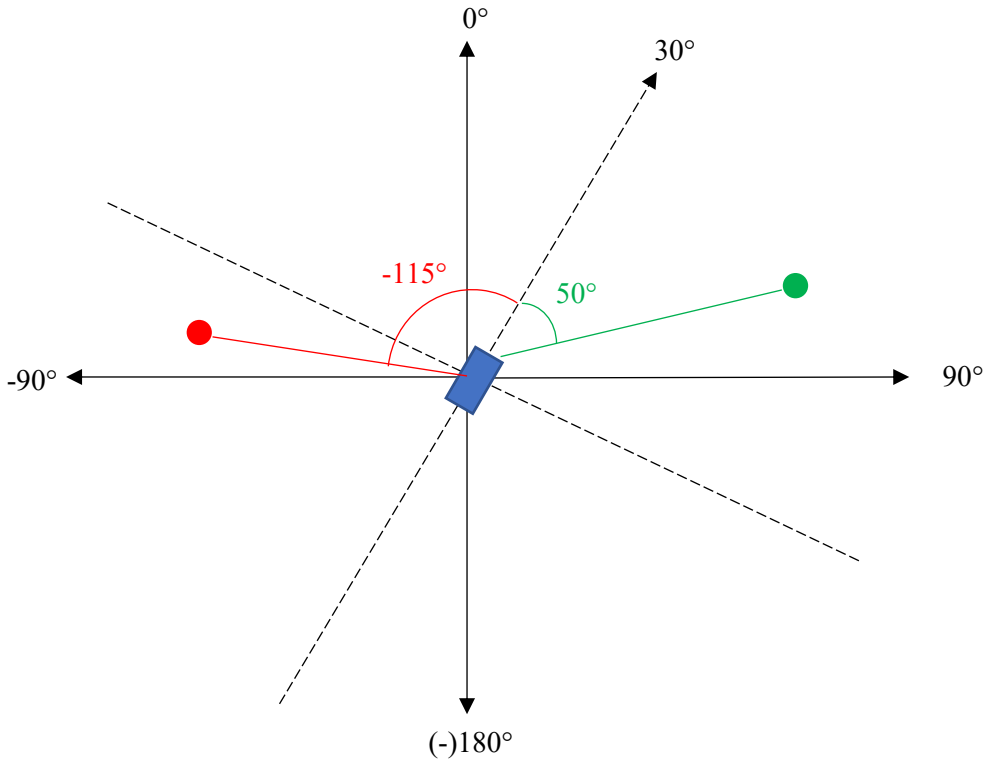


Figure 27: Shows how the AngleChange() function take count of the car orientation to find the angle change. The red angle compare to the car is inferior to -90° so it is not valid. The green point angle compared to the car orientation is between -90° and 90° so it is valid.

4.1.2 – Discretization

The second part of the program consists in discretizing the environment in order to create a finite number of possible future positions for the car. As explained previously the Delaunay Triangulation method was selected to achieve this task. It was implemented using the pre-build *Delauney()* function from the *spicy.spatial.Delauney* library in Python. The function receives the array containing the cones' coordinates and creates triangles according to the method – the cones being the vertices of the triangle.
The next step consists in finding the coordinates of the centres of the triangles' edges. These centres are the potential position points. But prior to this, there are two characteristics of the track that can be used to reduce the number of edges considered. Edges on the exterior of the track have no neighbor simplices and only edges that connect a yellow and blue cone have a centre that is in the track. Taking this characteristic into consideration, the *MidPointFinder()* function was designed to generate position points only within the track - which will effectively reduce the computational time. The function was optimized using the *.neighbors* attribute of the *Delauney* library to avoid considering multiple time the same edges.

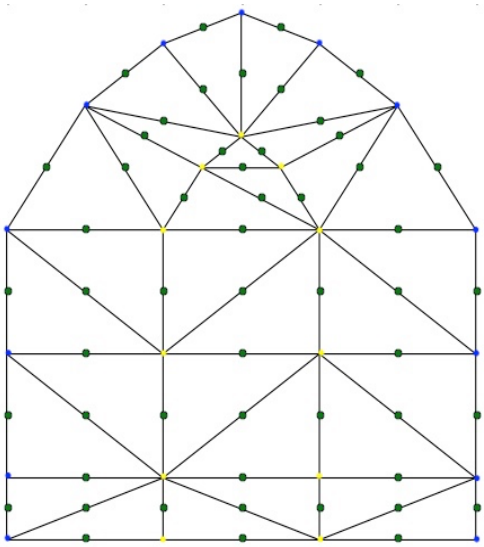


Figure 28: MidPointFinder() output when every center is added.

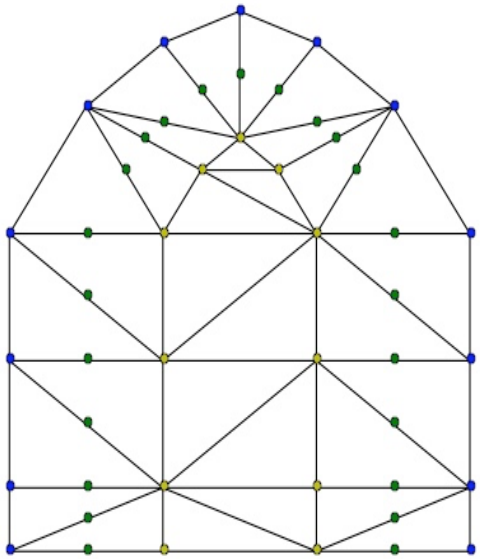


Figure 29: Final MidPointFinder() output. Only centers within the track are added.

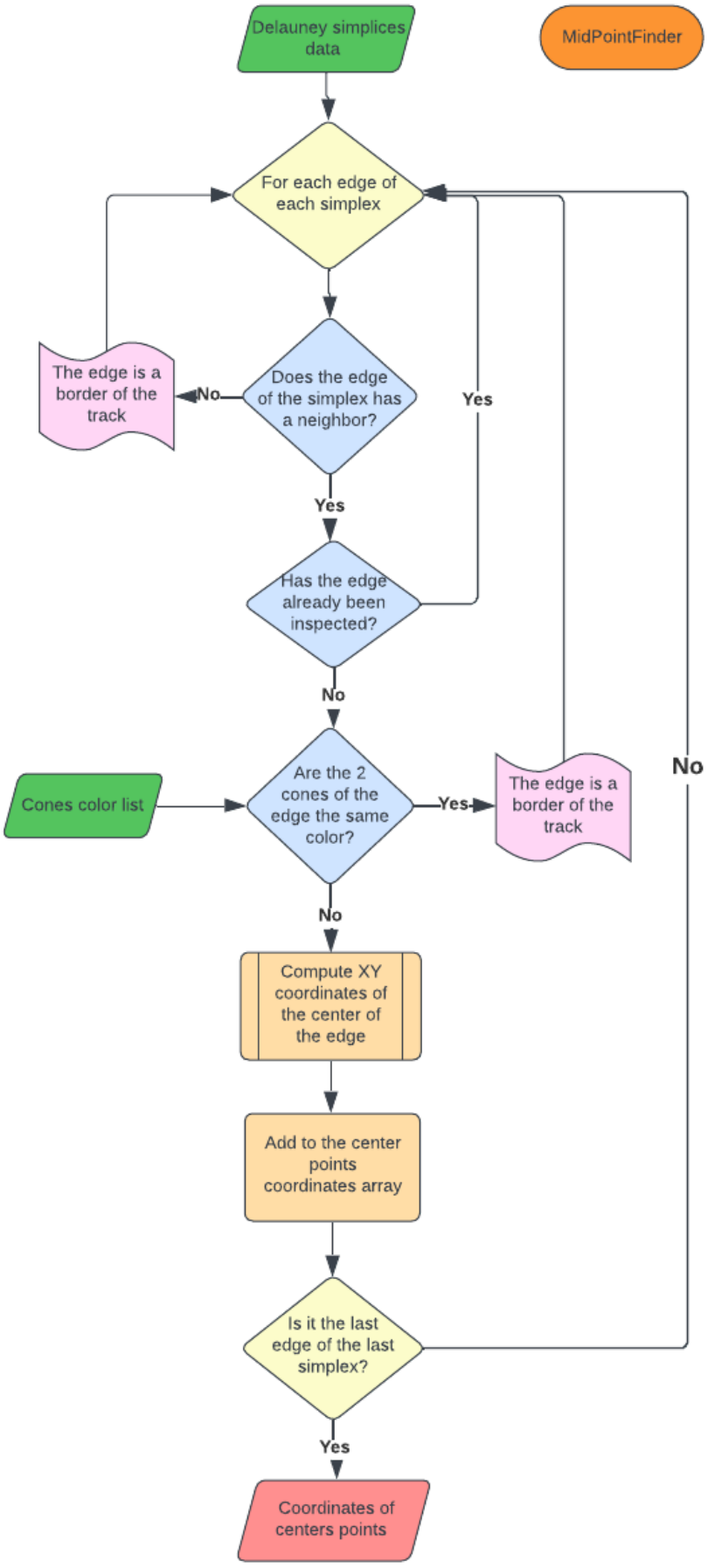


Figure 30: MidPointFinder() function flowchart

4.1.3 – Path Planning Algorithm

The speed of the "Contrained A*" algorithm is inversely proportional to the number of points in the trajectory, the number of potential points considered, and the number of criteria assessed. Thus, multiple variations of the algorithm have been developed and tested to reduce these factors.

The first version consisted in considering every point from the SLAM input compared to the last point in the trajectory and using three criteria to filter out the points – less than 10m from away, being in front, and not crossing a border. To reduce the computational effort, the order in which criteria were assessed depended on the difficulty to assess them. Not crossing a border being the hardest criteria to assess, it was implemented last so that fewer points had to be tested through it. Among the points that passed all criteria, the closest point to the last trajectory point was selected. The process was then repeated until no point passed all criteria. The algorithm resulted in a smooth trajectory within the track but a slow computational time. The output of this version can be seen in figure 25.

One inconvenience of the previous version is that it creates a trajectory made of multiple points in straight lines when only two points are needed to represent it. This considerably increases the number of steps to complete the path. Thus, the second version was designed to select the furthest point among those that pass all criteria. It resulted in a considerably faster time but the angle changes in the trajectory were too aggressive to be followed by the car.

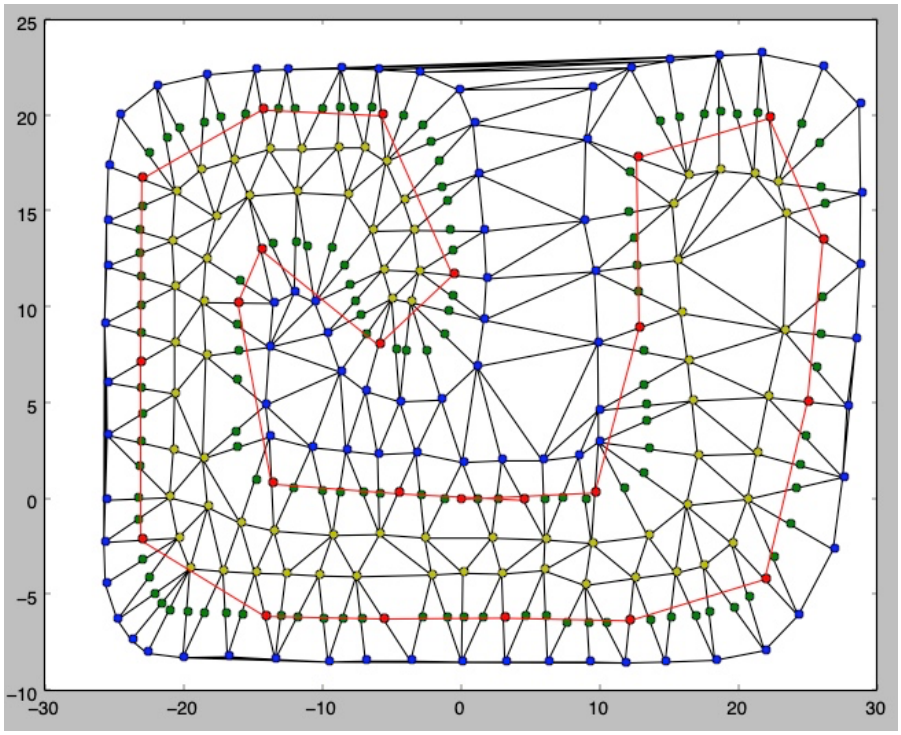


Figure 31: Version 2 of the local planning algorithm. Adding the furthest valid point

A second attempt to reduce the number of points in a straight line consisted in still selecting the furthest point but finding the angle change and selecting the second next furthest point if the angle change was superior to 20° to maintain a smooth trajectory. As it can be seen in figure 32 the algorithm was unable to output a valid path in a chicane. After selecting the furthest point with an angle change inferior to 20°, the car is in apposition where no points are within 20°. Thus, this version did not pass the first stage of testing. However, the trajectory would be rerun before the car reaches the end of the straight line given dynamic conditions, thus, the car should not end up in the position describe in figure 32. Given the time-constrained of the project and the necessity to develop a valid system, the version was abandoned. It is a potential route for future improvement.

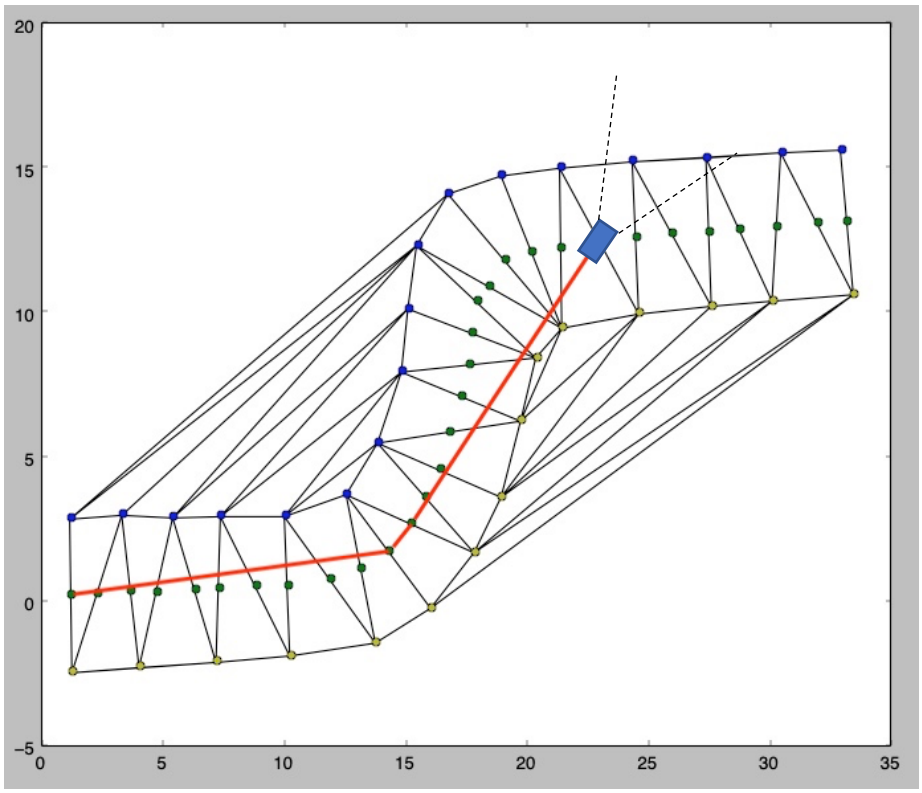


Figure 32: Version 3 of the local planning algorithm. Adding the furthest valid point if the angle change is inferior to 20°. In the above position the car can't see any valid point and is stuck.

The final version is reducing the number of points considered and criteria assessed by applying some of the criteria to the *TextToArray()* function directly. As explained in Section XX, *DistanceFinder()* and *AngleChange()* allows to extract only the cones within 10m and in front of the car. It is then only necessary to assess if adding a point to the trajectory result in crossing a border. This is performed by the *IsCrossBorder()* function. This version produced a smoother pass than the second version and faster result than the first one. It is described by the flowchart in figure 33.

The *IsCrossBorder()* function creates a first segment between the last trajectory point and the point being considered. Then, it creates a segment between every cone of the same colour spaced by less than 5m and checks if it is intersecting the first segment. If it does, reaching this point results in crossing a border. To check if two segments intersect it uses the function *IsIntersect()*. *IsIntersect()* equates the line equations of the two segments - $y = ax + b$ - to find the coordinates of the intersection point of the line that contains the segments. If the point is contained in the X-axis and Y-axis range of the two segments, then the segment intersects. Special cases such as the segments being parallel to the other, being parallel to the axis or being aligned are anticipated.

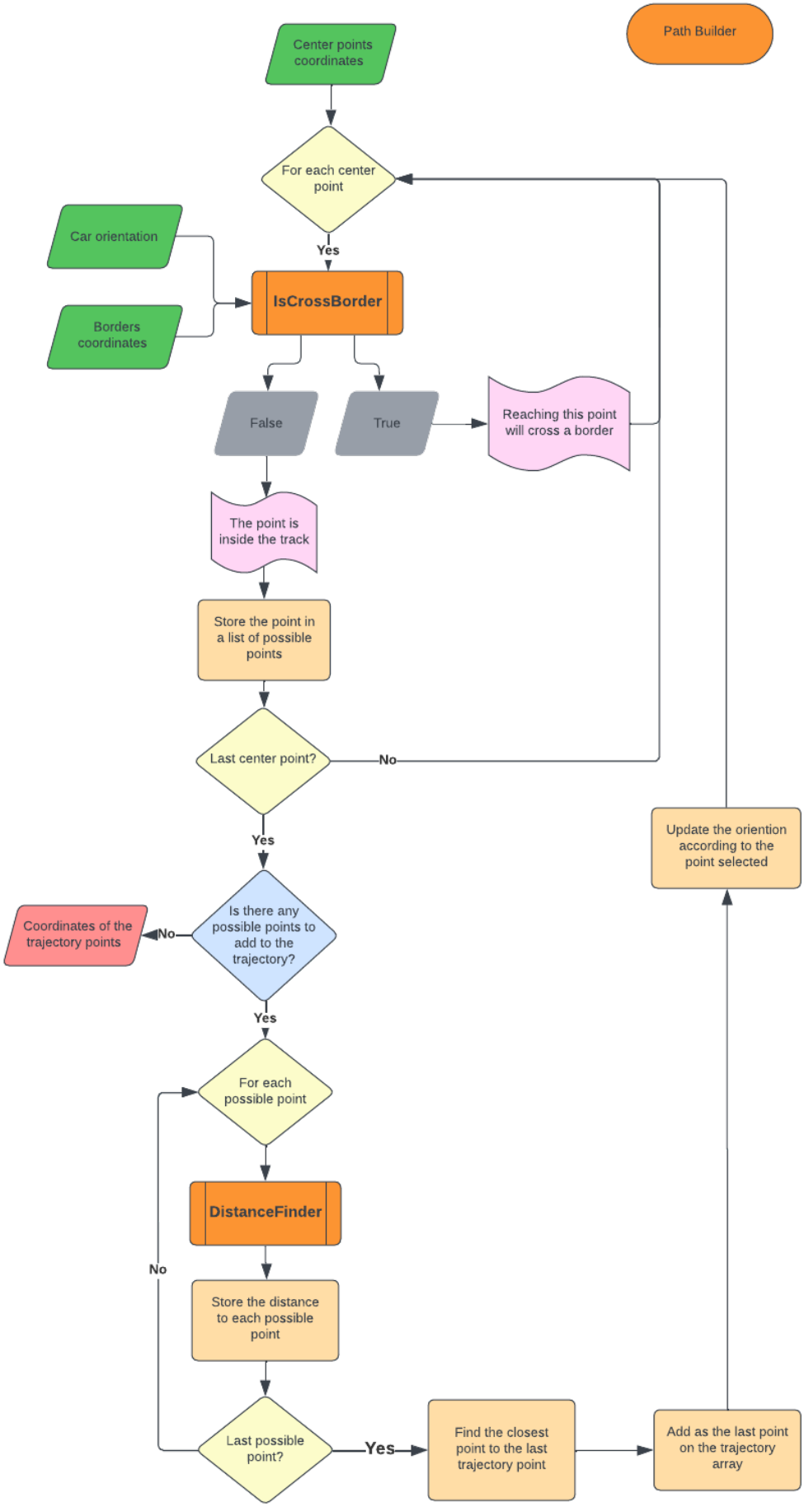


Figure 33: Flowchart of the local pathfinding algorithm

4.1.4 – Write-Output File

The output file is created using the function *ArrayToFile()*. It is a .csv file that contains the trajectory points, the position and orientation of the car and the borders. The first line of the file describes the structure of the following lines – tag, x, y, angle. The first element - tag - describe which type of line it is describing, “traj” for a trajectory point, “blue”, “yellow” or “orange” for a cone, and “car” for the car. The second and third elements – x, y - are respectively the x and y coordinates. The last element – angle – only applies to the car and give the last orientation of the car.
The order of the trajectory points in the file corresponds to their position in the trajectory, the last “traj” line being the last point of the trajectory. The car line is always written first, then it is the trajectory, the yellow cones and the blue cones.

Control_Output.csv

```
tag,x,y
point,-23.0,1.0
point,-23.774509804,1.64215686274
point,-24.9509803922,2.25490196078
point,-25.637254902,2.74509803922
point,-26.2009803922,3.18627450981
point,-26.9117647058,3.84803921569
point,-27.5490196078,5.0
point,-28.3578431373,6.0294117647
point,-28.7745098039,7.03431372549
yellow,-27.5,9.46078431373
yellow,-26.6666666667,7.45098039216
yellow,-25.0490196078,5.39215686275
yellow,-23.9215686275,4.50980392157
blue,-30.0490196078,4.60784313725
blue,-28.7745098039,2.30392156863
blue,-27.3529411765,0.980392156863
blue,-25.9803921569,0.0
blue,-23.6274509804,-1.22549019608
```

Figure 34: Output file in .csv format for the control system

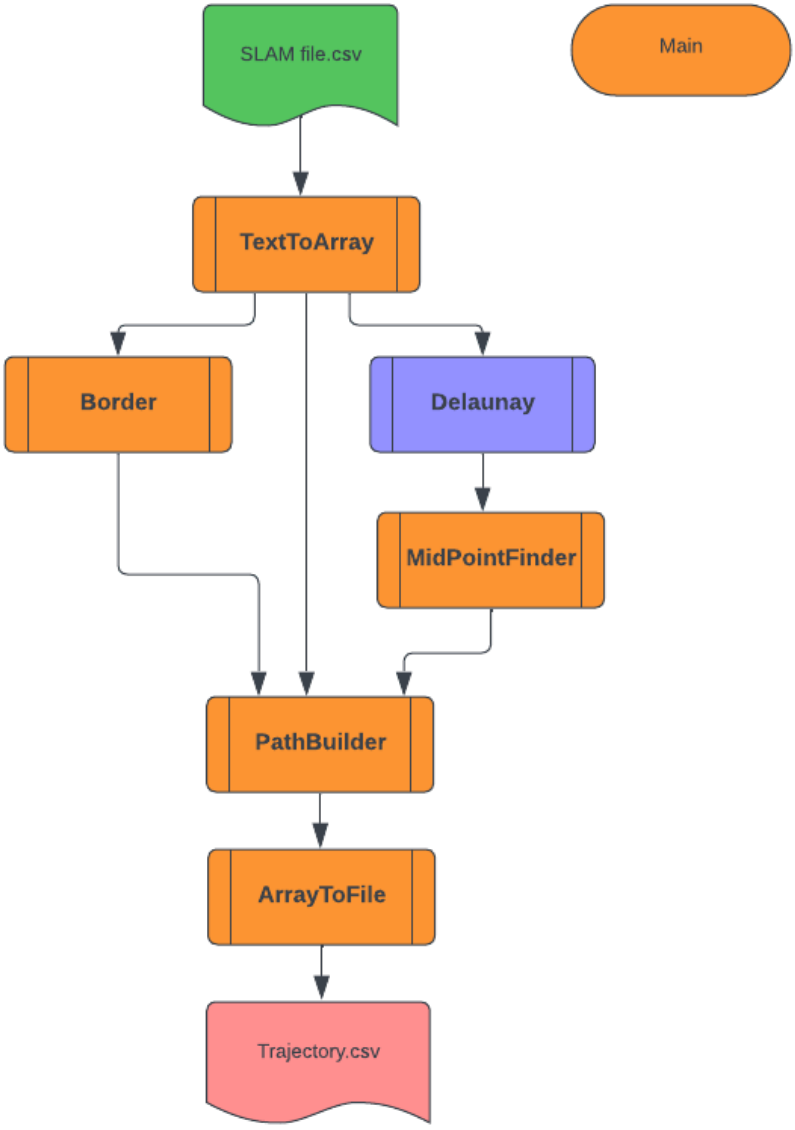


Figure 35: Complete Local Planning system flowchart

4.2 – Results

4.2.1 – Static testing

As can be seen in figure 36, 39 and 41 , the algorithm was successful in every type of curve when the cones are correctly identified. When one or multiple cones were missing, the algorithm was also successful. Failure to produce the complete path occurred when a cone was assigned with the wrong colour.

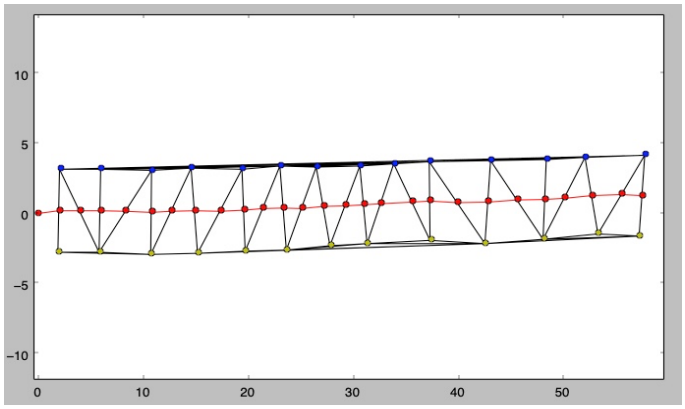


Figure 36: Valid path in straight line correctly identified

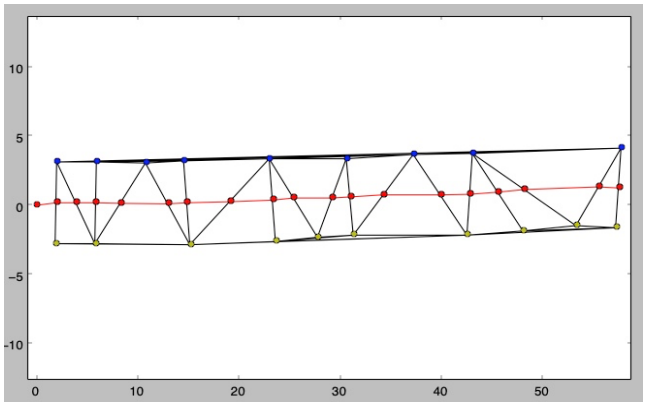


Figure 37: Valid path in straight line with missing cones

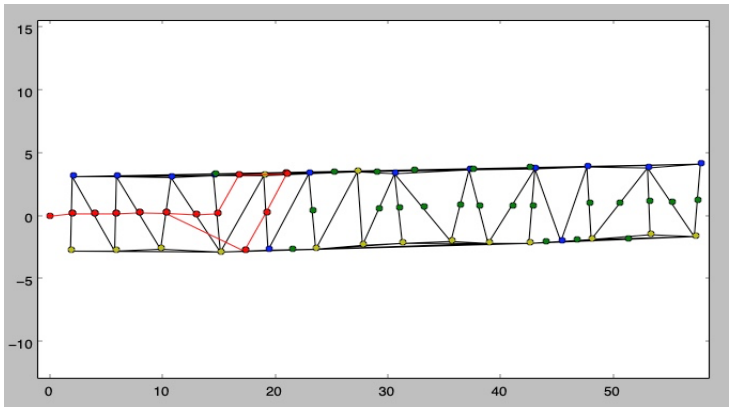


Figure 38: Invalid path in straight line with cones incorrectly identified

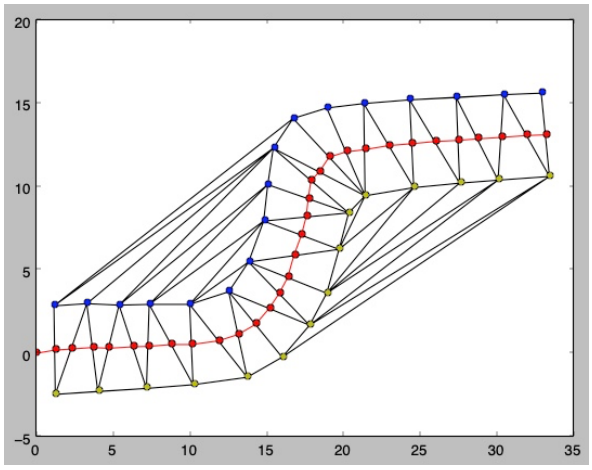


Figure 39: Valid path in chicane with cones correctly identified

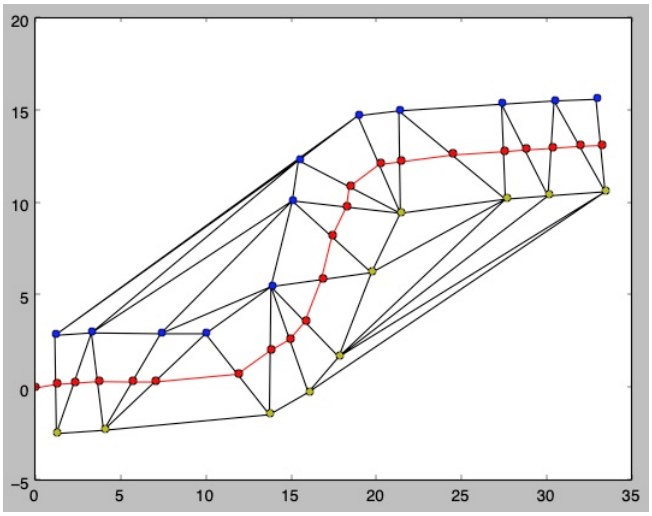


Figure 40: Valid path in chicane with cones correctly missing

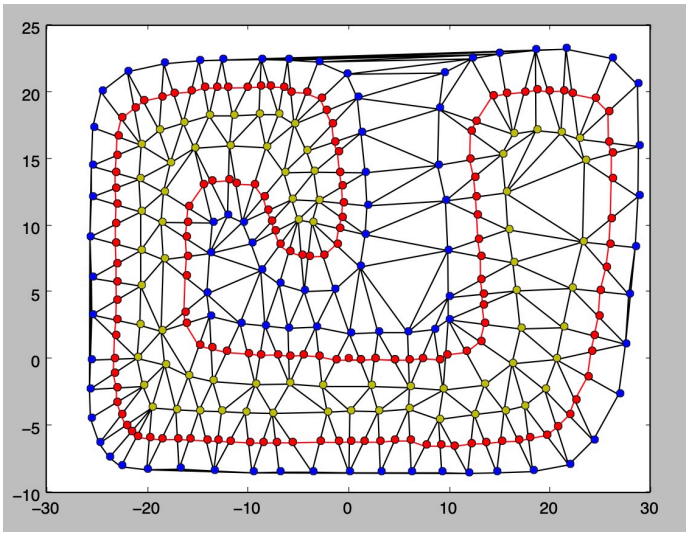


Figure 41: Valid path in a of different curve with cones correctly identified

4.2.3 – Dynamic testing

The program was put through *Simulator* to test its response in a dynamic environment and compatibility with the whole system. The car was able to complete one lap on every test track. And it resulted in an average path length of 9.72m, which satisfies the control system needs.
Figures 42 show what local information is available to the car along the track and the path that is found.

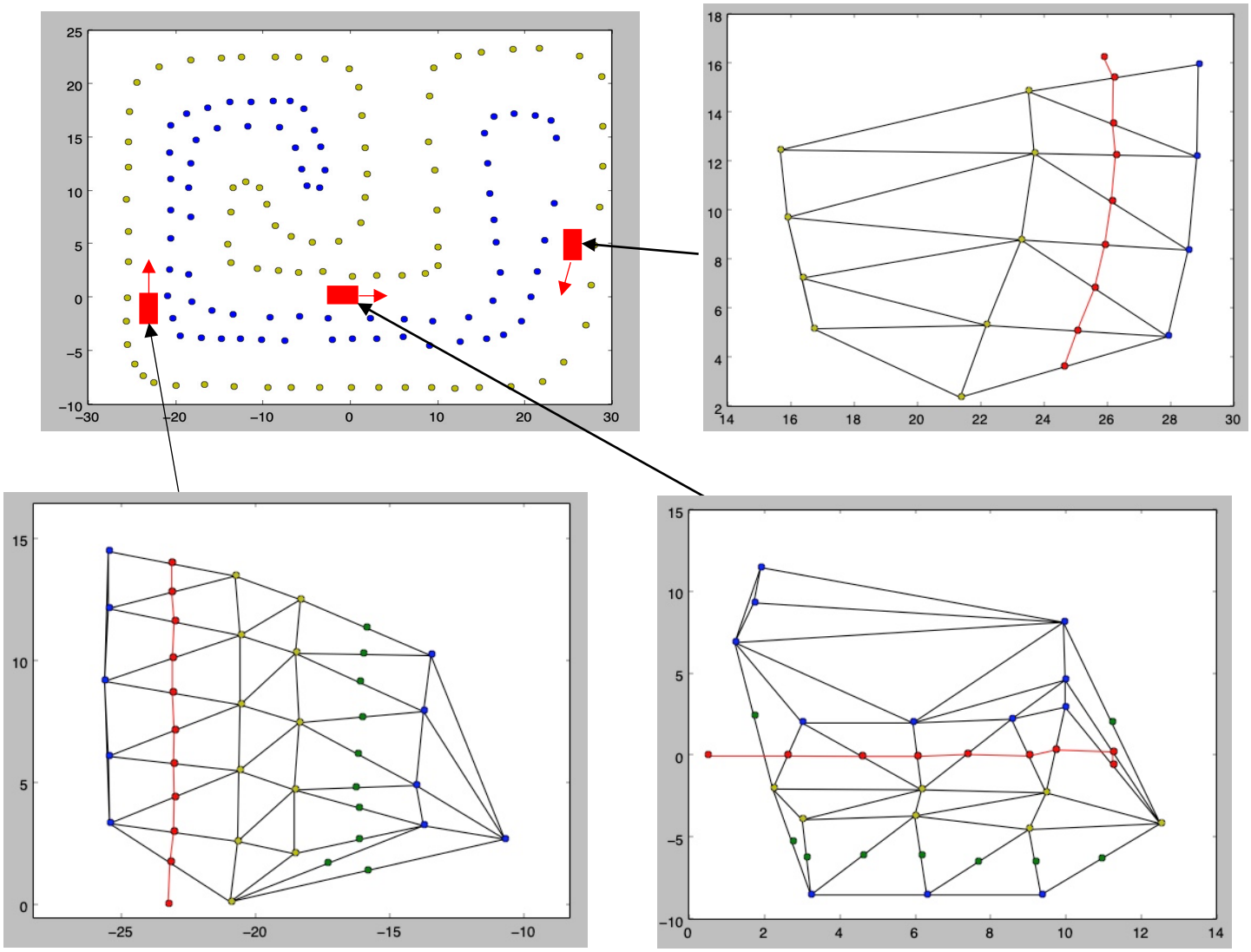


Figure 42: Shows the path produced given different local map in dynamic simulation. Yellow and Blue cones are inversed on the local graph due to an error

4.2.3 – Performance testing

The algorithm operated at an average speed of 0.2 sec in the figure 41 track. The maximum achievable speed was 4.2m/s. The trajectory was long enough to allow the control system to produce the right model. And it was smooth enough to be followed by the car.

4.3 – Limitations and potential errors

The “Delauney/Constrained A*” method was selected with the aim of, being able to develop at minimum a system that can operate on the car. Thus, in regard to this requirement and according to the test results obtained, the design selection and implementation can be seen as successful. However, in regard to the other requirement, the method was clearly limited in two aspects - reliability and speed.

As a single path is being built by the car, the algorithm has a tendency to produce no path if no points are considered as valid potential trajectory points. This would result in the car being stuck in position. It was found that such circumstances could happen when the car is travelling too fast, when the color of some cones is incorrectly identified or when the car deviates too much from its supposed position.

The performance of the algorithm allowed a maximum speed of X m.s. Such speed would result in poor competition results. The limiting factors for speed are path length and computational speed. It was shown that using every cone that has been seen to increase the path length resulted in considerably decreasing the computational speed. Thus, the path length is limited by the performance of the perception pipeline and so improving the computational speed is the remaining option.

The reliability and speed could be increased by implementing one of the “weighted selection” methods. That way, a path would always be produced, thus, the car will be less likely to get stuck in position and a faster restart frequency could be used.

Moreover, one alternative that was not investigated due to the whole pipeline segmentation, is to fusion the control system and pathfinding pipeline. That way the car model would be available for the path planning system and so a trajectory based on the car capacity could be produced directly. This would allow to produce a more reliable trajectory and increase the overall computational speed of the whole system.

6 - Reference

[1] -
Formula Student, “FORMULA STUDENT AI 2022 RULES,” 2022. [Online]. Available: https://www.imeche.org/docs/default-source/1-oscar/formula-student/2022/rules/fs-ai-2022-rules-v1-0.pdf?sfvrsn=2

[2] -
iMech, “FS-AI Dynamic Events Setup and Cones Specification,” 2021. [Online]. Available: https://www.imeche.org/docs/default-source/1-oscar/formula-student/2021/forms/ai/fs-ai-dynamic-events-setup-and-cones-specification.pdf?sfvrsn=2

[3] -
M. Bevilacqua, A. Tsourdos, and A. Starr, “International Instrumentation and Measurement Technology Conference,” 2017.

Appendix

A - LocalPlanning3_0

```
import numpy as np
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt
import random
import math

# Read the text file where information about cones that have been seen are stored
# Cones data is stored in the following format [X coordinate, Y coordinate, color]
# conesPoses stores the X and Y coordinates, conesColors stores the colors
# Both array have the same length and order, so that color and coordinates of a
cone can be found with the same index
# Cones that too far from the car and behind teh acr are not added
def TextToArray(name1, carXY, carAngle):
    conesPoses = []
    conesColors = []
    file = open(name1 +'.csv', 'r')
    for line in file:
        if "blue" in line or "orange" in line or "yellow" in line:
            if line != "" and line != " " and line != '\n':
                coma1 = line.find(',')
                Color = line[0: coma1]
                coma2 = line[coma1+1:].find(',')
                coneX = float(line[coma1 + 1: coma1 + coma2 + 1])
                coma3 = line.find('\n')
                coneY = float(line[coma1 + coma2 + 2: coma3])
                conePose = []
                conePose.append(coneX)
                conePose.append(coneY)
                dist = DistFinder(conePose, carXY)
                if dist <= 15:
                    #if AngleChange(carXY, conePose, carAngle)< 90 and
AngleChange(carXY, conePose, carAngle) >-90:
                    conesColors.append(Color)
                    conesPoses.append(conePose)
    file.close()
    conesPoses = np.array(conesPoses)
    return [conesPoses, conesColors]


# Find the coordinate of the center of the edge of the Delauney simplices
# Only the coordinates of centers between a blue and a yellow cones are stored,
meaning that the point is in the track
def MidPointFinder(conesPoses, conesColors, tri):
```

```
    MidPoints = []
    for i in range(len(tri.simplices)):
        for n in range(3):
            # if tri.neighbors = -1, the edge has no neighbor, therefore the
segment is an external border of the track
            if tri.neighbors[i,n] == -1 or tri.neighbors[i,n] > i:
                #triangle contain the coordinates of the 3 vertices of a delauney
triangle
                triangle = conesPoses[tri.simplices[i]]
                MidPointXY = []
                ColorA= ""
                ColorB= ""
                if n == 0:
                    for j in range(len(conesPoses)):
                        if conesPoses[j,0] == triangle[1,0] and conesPoses[j,1] ==
triangle[1,1]:
                            ColorA = conesColors[j]
                            break
                    for j in range(len(conesPoses)):
                        if conesPoses[j,0] == triangle[2,0] and conesPoses[j,1] ==
triangle[2,1]:
                            ColorB = conesColors[j]
                            break
                    # Check if the midpoint is on the borders or in the tracks
                    if (ColorA == 'blue' and ColorB =='yellow') or (ColorA ==
'yellow' and ColorB =='blue'):
                            MidPointX = (triangle[1,0] + triangle[2,0])/2.
                            MidPointY = (triangle[1,1] + triangle[2,1])/2.
                if n == 1:
                    for j in range(len(conesPoses)):
                        if conesPoses[j,0] == triangle[0,0] and conesPoses[j,1] ==
triangle[0,1]:
                            ColorA = conesColors[j]
                            break
                    for j in range(len(conesPoses)):
                        if conesPoses[j,0] == triangle[2,0] and conesPoses[j,1] ==
triangle[2,1]:
                            ColorB = conesColors[j]
                            break
                    if (ColorA == 'blue' and ColorB =='yellow') or (ColorA ==
'yellow' and ColorB =='blue'):
                        MidPointX = (triangle[0,0] + triangle[2,0])/2.
                        MidPointY = (triangle[0,1] + triangle[2,1])/2.
                if n == 2:
                    for j in range(len(conesPoses)):
                        if conesPoses[j,0] == triangle[0,0] and conesPoses[j,1] ==
triangle[0,1]:
                            ColorA = conesColors[j]
                            break
                    for j in range(len(conesPoses)):
```

```
                        if conesPoses[j,0] == triangle[1,0] and conesPoses[j,1] ==
triangle[1,1]:
                            ColorB = conesColors[j]
                            break
                    if (ColorA == 'blue' and ColorB =='yellow') or (ColorA ==
'yellow' and ColorB =='blue'):
                        MidPointX = (triangle[0,0] + triangle[1,0])/2.
                        MidPointY = (triangle[0,1] + triangle[1,1])/2.

                if ColorA != ColorB:
                    MidPointXY.append(MidPointX)
                    MidPointXY.append(MidPointY)
                    MidPoints.append(MidPointXY)
    MidPoints = np.array(MidPoints)
    return MidPoints

# Find the absolute distance between two points
def DistFinder(A, B):
    if A[0] >= B[0]:
        DistX = A[0] - B[0]
    else:
        DistX = B[0] - A[0]

    if A[1] >= B[1]:
        DistY = A[1] - B[1]
    else:
        DistY = B[1] - A[1]

    Dist= (DistX**2 + DistY**2)**0.5
    return Dist


# Find if two segments intersect
# if S1 is a segment between A and B, S1 = [[xA, yA],[xB, yB]]
def IsIntersect(S1,S2):
    S1 = np.array(S1)
    S2 = np.array(S2)

    # Define the range of each segment in the X-axis and Y-axis
    # So that RangeXD1[0] contains the smallest X coordinate of S1 and RangeXD1[1]
the biggest
    if S1[0,0] <= S1[1,0]:
        RangeXD1 = [S1[0,0],S1[1,0]]
    else:
        RangeXD1 = [S1[1,0],S1[0,0]]

    if S1[0,1] <= S1[1,1]:
        RangeYD1 = [S1[0,1],S1[1,1]]
    else:
        RangeYD1 = [S1[1,1],S1[0,1]]
```

```
    if S2[0,0] <= S2[1,0]:
        RangeXD2 = [S2[0,0],S2[1,0]]
    else:
        RangeXD2 = [S2[1,0],S2[0,0]]

    if S2[0,1] <= S2[1,1]:
        RangeYD2 = [S2[0,1],S2[1,1]]
    else:
        RangeYD2 = [S2[1,1],S2[0,1]]

    # special case, if the two segment are perpendiculaire to the X-axis
    if (S1[0,0] == S1[1,0]) and (S2[0,0] == S2[1,0]):

        # special case, if the two segment are perpendiculaire to the X-axis and
aligned
        if S1[0,0] == S2[0,0]:
            if ((RangeYD1[0] <= RangeYD2[1]) and (RangeYD1[0] >= RangeYD2[0]))  or
((RangeYD1[1] >= RangeYD2[0]) and (RangeYD1[1] <= RangeYD2[1])) :
                intersect = True
            else:
                intersect = False
        else:
            intersect = False

    # special case, if the two segment are parrallel to the X-axis
    elif (S1[0,1] == S1[1,1]) and (S2[0,1] == S2[1,1]):

        # special case, if the two segment are parrallel to the X-axis and aligned
        if S1[0,1] == S2[0,1]:
            if ((RangeXD1[0] <= RangeXD2[1]) and (RangeXD1[0] >= RangeXD2[0]))  or
((RangeXD1[1] >= RangeXD2[0]) and (RangeXD1[1] <= RangeXD2[1])) :
                intersect = True
            else:
                intersect = False
        else:
            intersect = False

    # Find CrossX and CrossY where the segment would cross if they were infinite
    else:
        # special case, if S1 is perpendicular to the X-axis
        if S1[0,0] == S1[1,0]:
            CrossX= S1[0,0]

            # if S1 is perpendicular to the X-axis and S2 is parallel to the X -
axis
            if S2[0,1] == S2[1,1]:
                CrossY= S2[0,1]
```

```
            # if S1 is perpendicular to the X-axis and S2 is defined by Y = c * X +
d
            else:
                c = (S2[0,1] - S2[1,1])/1.0/(S2[0,0] - S2[1,0])
                d = S2[0,1]- (S2[0,0]*c)
                CrossY = c * CrossX + d

        # special case, if S2 is perpendicular to the X-axis
        elif S2[0,0] == S2[1,0]:
            CrossX= S2[0,0]

            # if S2 is perpendicular to the X-axis and S1 is parallel to the X -
axis
            if S1[0,1] == S1[1,1]:
                CrossY= S1[0,1]

            # if S2 is perpendicular to the X-axis and S1 is defined by Y = a * X +
b
            else:
                a = (S1[0,1] - S1[1,1])/1.0/(S1[0,0] - S1[1,0])
                b = S1[0,1]- (S1[0,0]*a)
                CrossY = a * CrossX + b

        # special case, if S1 is parallel to the X-axis
        elif S1[0,1] == S1[1,1]:
            CrossY= S1[0,1]

            # if S1 is parrallel to the X-axis and S2 is perpendicular to the X -
axis
            if S2[0,0] == S2[1,0]:
                CrossX= S2[0,0]

            # if S1 is parrallel to the X-axis and S2 is defined by Y = c * X + d
            else:
                c = (S2[0,1] - S2[1,1])/1.0/(S2[0,0] - S2[1,0])
                d = S2[0,1]- (S2[0,0]*c)
                CrossX = (CrossY - d)/c

        # special case, if S2 is parallel to the X-axis
        elif S2[0,1] == S2[1,1]:
            CrossY= S2[0,1]

            # if S2 is parrallel to the X-axis and S1 is perpendicular to the X -
axis
            if S1[0,0] == S1[1,0]:
                CrossX= S2[0,0]

            # if S2 is parrallel to the X-axis and S1 is defined by Y = a * X + b
            else:
                a = (S1[0,1] - S1[1,1])/1.0/(S1[0,0] - S1[1,0])
```

```
                b = S1[0,1]- (S1[0,0]*a)
                CrossX = (CrossY - b)/a

        # General case, where S1 is defined by Y = a * X + b and S2 by Y = c * X +
d
        else:
            a = (S1[0,1] - S1[1,1])/1.0/(S1[0,0] - S1[1,0])
            b = S1[0,1]- (S1[0,0]*a)

            c = (S2[0,1] - S2[1,1])/1.0/(S2[0,0] - S2[1,0])
            d = S2[0,1]- (S2[0,0]*c)

            # Special case where, S1 and S2 are parallel but not parallel or
perpendicular to an axis
            if a == c:
                # if not aligned
                if b != d:
                    intersect = False
                # if aligned
                elif ((RangeYD1[0] <= RangeYD2[1]) and (RangeYD1[0] >=
RangeYD2[0]))  or ((RangeYD1[1] >= RangeYD2[0]) and (RangeYD1[1] <= RangeYD2[1])) :
                    intersect = True
                else:
                    intersect = False
                return intersect

            CrossX = (d - b) /1.0/ (a - c)
            CrossY = (c*b - a*d)/1.0/(c - a)

        # If CrossX and CrossY are included in the x - axis and y - axis range of
S1 and S2, then they intersect
        if (CrossX >= RangeXD1[0] and CrossX <= RangeXD1[1]) and (CrossX >=
RangeXD2[0] and CrossX <= RangeXD2[1]):
            inX = True
        else:
            inX = False

        if (CrossY >= RangeYD1[0] and CrossY <= RangeYD1[1]) and (CrossY >=
RangeYD2[0] and CrossY <= RangeYD2[1]):
            inY = True
        else:
            inY = False

        if inX == True and inY == True:
            intersect = True
        else:
            intersect = False

    return(intersect)
```

```
# Create 2 lists containning the coordinates of all Yellow cones and all Blue cones
def Borders(conesPoses, conesColors):
    BorderY = []
    BorderB = []
    for i in range(len(conesPoses)):
        if conesColors[i] == "yellow":
            BorderY.append(conesPoses[i])
        elif conesColors[i] == "blue":
            BorderB.append(conesPoses[i])
    return [BorderY, BorderB]

# Check if two possible trajectory points can be linked without intersercting one
of the border
def IsCrossBorder(ClosestPoint, NewPoint, conesPose, conesColor):
    S1 = [[0,0],[0,0]]
    S2 = []
    S2.append(ClosestPoint)
    S2.append(NewPoint)
    Yellow = Borders(conesPose, conesColor)[0]
    Blue = Borders(conesPose, conesColor)[1]

    for i in (range(len(Yellow))):
        S1[0] = Yellow[i]
        for n in range(len(Yellow)):
            S1[1] = Yellow[n]
            dist = DistFinder(S1[0], S1[1])
            if dist <= 5:
                if IsIntersect(S1, S2) == True:
                    return True

    for i in (range(len(Blue))):
        S1[0] = Blue[i]
        for n in range(len(Blue)):
            S1[1] = Blue[n]
            dist = DistFinder(S1[0], S1[1])
            if dist <= 5:
                if IsIntersect(S1, S2) == True:
                    return True

    return False

# Find what is the angle between the car and the next point based on the car
orientation
# A point on the positive Y -axis is at 0 degree, a point on the negative Y -axis
is at 180 or -180 degree
# A point on the positive X -axis is at 90 degree, a point on the negative Y -axis
is at -90 degree
def AngleChange(CarXY, Point, CarAngle):

    PointShifted = [Point[0]-CarXY[0], Point[1]-CarXY[1]]
```

```
    CarToPoint = DistFinder([0, 0], PointShifted)
    cosPoint = PointShifted[0] / 1.0/ CarToPoint
    PointAngle = np.arccos(cosPoint)
    PointAngle = PointAngle * 180/ math.pi
    if PointShifted[1] < 0:
        PointAngle = PointAngle * -1

    AngleToCar = 90 - PointAngle - CarAngle
    if AngleToCar > 180:
        AngleToCar = AngleToCar - 360
    elif AngleToCar < -180:
        AngleToCar = AngleToCar + 360

    return AngleToCar

# Build a path through the centers point, starting from the car position
# The next point as to be at less than 10 meter, in a range of 180 degree based on
the car orientation, not intersect a border.
# Among the possible points, the closest is selected
def PathBuilder(carXY, centers, conesPoses, conesColors, carAngle, Start):
    trajPoints = [carXY]
    i = 0
    NbPossiblePts = 1
    while NbPossiblePts != 0:
        PossiblePts = []
        for n in range(len(centers)):
            dist = DistFinder(trajPoints[i], centers[n])
            #Check if the point is at less than 10m from the car. Further cones
have more uncetainty on their position. Close cones are considered for safety
            if dist > 0:
                if len(trajPoints) == 1:
                    PrevAngle = carAngle
                    PrevPoint = carXY
                # Find the deviation between the last point of the path and the
next one
                NewAngle = AngleChange(PrevPoint, centers[n], PrevAngle)
                #Check if the next point is in front of the current one based on
the orientation of the car if it follows the path.
                #Allows to avoid going back on points already contained in the path
                if NewAngle < 90 and NewAngle > -90:
                    # Check if addding the next point to the trajectory would cross
a border
                    if IsCrossBorder(PrevPoint, centers[n], conesPoses,
conesColors) == False:
                        PossiblePts.append(centers[n])
        NbPossiblePts = len(PossiblePts)
        #Stop finding new points if no points match the criteria
        if NbPossiblePts == 0:
            break
        # Among the possible points select the closest one
```

```
        else:
            distMin = 1000
            for n in range(NbPossiblePts):
                dist = DistFinder(trajPoints[i],PossiblePts[n])
                if dist < distMin:
                    distMin = dist
                    distMinIndex = n
            trajPoints.append(PossiblePts[distMinIndex])

            # update variables
            i = i+1
            PrevAngle = AngleChange(PrevPoint, PossiblePts[distMinIndex],
PrevAngle) + PrevAngle
            if PrevAngle >= 360:
                PrevAngle = PrevAngle - 360
            elif PrevAngle <= -360:
                PrevAngle = PrevAngle + 360
            PrevPoint = PossiblePts[distMinIndex]

        # Stop the loop if the start point of the track is added to the path
        # This means that the path found has reached the start line again. at that
point the Global Planning will continue
        S1 = [trajPoints[len(trajPoints) - 2], PossiblePts[distMinIndex]]
        S2 = [[0,-2],[0, 2]]
        if  IsIntersect(S1, S2) == True and i !=1:
            print("Traj reach lap")
            break
        # Protect the programm if blocked in an infinite loop
        elif i>200:
            print('exit2')
            break

    return trajPoints


def ClearFile(name2):
    file = open(name2 +'.csv', 'w')
    file.truncate()
    file.close()

def WriteOutput(Traj, borderY, borderB, name2):
    file = open(name2 +'.csv', 'a')
    file.write("tag,x,y\n")

    for i in range(len(Traj)):
        file.write("point," + str(Traj[i][0]) + "," + str(Traj[i][1]) + "\n")
    for i in range(len(borderY)):
        file.write("yellow," + str(borderY[i][0]) + "," + str(borderY[i][1]) +
"\n")
    for i in range(len(borderB)):
```

```
        file.write("blue," + str(borderB[i][0]) + "," + str(borderB[i][1]) + "\n")

    file.close()

def Main(carPose, CarAngle, firstPoint):

    InputFile = "Partial_SLAM_Input"
    OutputFile = "Control_Output"

    # Get cones coordinates and color from a text file from SLAM
    cones = TextToArray(InputFile, carPose, CarAngle)
    conesXY = cones[0]
    Color = cones[1]

    #Discretise the environment with Delauney triangulation method
    Tri = Delaunay(conesXY)

    # Find the centers of Delauney triangles' edges and keep the one inside the
track
    Centers = MidPointFinder(conesXY,Color, Tri)

    # Create array containning coordinates of borders
    BorderY, BorderB = Borders(conesXY, Color)
    BorderY = np.array(BorderY)
    BorderB = np.array(BorderB)


    # Find the trajectory to stay inside the track starting from the car position
    Trajectory = PathBuilder(carPose, Centers, conesXY, Color, CarAngle,
firstPoint)
    Trajectory = np.array(Trajectory)

    ClearFile(OutputFile)
    WriteOutput(Trajectory, BorderY, BorderB, OutputFile)


    """
    # Plot the Delauney triangulation
    plt.triplot(conesXY[:,0], conesXY[:,1], Tri.simplices)
    #plt.plot(conesXY[:,0], conesXY[:,1], 'o')
    # Plot the track left border in yellow
    plt.plot(BorderY[:,0], BorderY[:,1], 'yo')
    # Plot the track right border in blue
    plt.plot(BorderB[:,0], BorderB[:,1], 'bo')
    # Plot the centers point in green
    plt.plot(Centers[:,0], Centers[:,1], 'go')
    # Plot and link the trajectory point in red
    plt.plot(Trajectory[:,0], Trajectory[:,1], marker = 'o', color = 'red')

    plt.show()
```

```
    """

    return Trajectory
```

B – Simulator3

```
from re import T
import numpy as np
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt
import random
import math
import time
import LocalPlanning3_0

def ReadFile(name):
    FileData = []
    file = open(name +'.csv', 'r')
    for line in file:
        if "blue" in line or "yellow" in line or "orange" in line:
            if line != "" and line != " " and line != '\n':
                FileData.append(line)
    file.close()
    return FileData

def ClearFile(name2):
    file = open(name2 +'.csv', 'w')
    file.truncate()
    file.close()


def Separator(FileData):
    conesPoses = []
    conesColors = []
    for line in FileData:
        coma1 = line.find(',')
        Color = line[0: coma1]
        coma2 = line[coma1+1:].find(',')
        coneX = float(line[coma1 + 1: coma1 + coma2 + 1])
        coma3 = line.find('\n')
        coneY = float(line[coma1 + coma2 + 2: coma3])
        conePose = []
```

```
        conePose.append(coneX)
        conePose.append(coneY)
        conesColors.append(Color)
        conesPoses.append(conePose)
    conesPoses = np.array(conesPoses)
    return [conesPoses, conesColors]

def DistFinder(A, B):
    if A[0] >= B[0]:
        DistX = A[0] - B[0]
    else:
        DistX = B[0] - A[0]

    if A[1] >= B[1]:
        DistY = A[1] - B[1]
    else:
        DistY = B[1] - A[1]

    Dist= (DistX**2 + DistY**2)**0.5
    return Dist

def AngleChange(CarXY, Point, CarAngle):

    PointShifted = [Point[0]-CarXY[0], Point[1]-CarXY[1]]
    CarToPoint = DistFinder([0, 0], PointShifted)
    cosPoint = PointShifted[0] / 1.0/ CarToPoint
    PointAngle = np.arccos(cosPoint)
    PointAngle = PointAngle * 180/ math.pi
    if PointShifted[1] < 0:
        PointAngle = PointAngle * -1

    AngleToCar = 90 - PointAngle - CarAngle
    if AngleToCar > 180:
        AngleToCar = AngleToCar - 360
    elif AngleToCar < -180:
        AngleToCar = AngleToCar + 360

    return AngleToCar

def CarPosition(speed, t, path):
    travelled = t * speed
    totDist = 0
    i = 0
    while totDist < travelled and i < len(path) - 1:
        TotDist0 = totDist
        totDist = TotDist0 + DistFinder(path[i], path[i+1])
        i = i+1

    prevPoint = path[i - 1]
    nextPoint = path[i]
```

```
    if prevPoint[0] >= nextPoint[0]:
        DistX = prevPoint[0] - nextPoint[0]
    else:
        DistX = nextPoint[0] - prevPoint[0]

    if prevPoint[1] >= nextPoint[1]:
        DistY = prevPoint[1] - nextPoint[1]
    else:
        DistY = nextPoint[1] - prevPoint[1]

    if DistX == 0:
        tan= 0
    else:
        tan = DistY/1.0/DistX
    angle = np.arctan(tan)
    dist = travelled - TotDist0
    x = np.cos(angle) * dist
    y = np.sin(angle) * dist

    if prevPoint[0] <= nextPoint[0]:
        carPoseX = prevPoint[0] + x
    else:
        carPoseX = prevPoint[0] - x

    if prevPoint[1] <= nextPoint[1]:
        carPoseY = prevPoint[1] + y
    else:
        carPoseY = prevPoint[1] - y

    return [[carPoseX, carPoseY], prevPoint, nextPoint]

def AddVisibleCones(name2, xy, carPose, carAngle, conesData):
    Index = []
    distance = []
    for cone in range(len(xy)):
        dist = DistFinder(carPose, xy[cone])
        if dist < 10:
            NewAngle = AngleChange(CarPose, xy[cone], carAngle)
            if NewAngle < 60 and NewAngle > -60:
                Index.append(cone)
                distance.append(dist)

    sorted = np.argsort(distance)
    Index = np.array(Index)
    Index = Index[sorted]

    for cone in Index:
        check = False
        file = open(name2 + '.csv', 'r')
```

```
        for line in file:
            if line == conesData[cone] :
                check = True
        if check == False:
            file.close()
            file = open(name2 +'.csv', 'a')
            file.write(conesData[cone])
            file.close()
        else:
            file.close()

def CarOrientation(nextPoint, prevPoint):

    PointShifted = [nextPoint[0] - prevPoint[0], nextPoint[1] - prevPoint[1]]
    PointToCar = DistFinder([0, 0], PointShifted)
    cosPoint = PointShifted[0] / 1.0/ PointToCar
    PointAngle = np.arccos(cosPoint)
    PointAngle = PointAngle * 180/ math.pi
    if PointShifted[1] < 0:
        PointAngle = PointAngle * -1

    AngleToCar = 90 - PointAngle
    if AngleToCar > 180:
        AngleToCar = AngleToCar - 360
    elif AngleToCar < -180:
        AngleToCar = AngleToCar + 360

    return AngleToCar

# Find if two segments intersect
# if D1 is a segment between A and B, D1 = [[xA, yA],[xB, yB]]
def IsIntersect(D1,D2):
    D1 = np.array(D1)
    D2 = np.array(D2)

    # Define the range of each segment in the X-axis and Y-axis
    # So that RangeXD1[0] contains the smallest X coordinate of D1 and RangeXD1[1]
the biggest
    if D1[0,0] <= D1[1,0]:
        RangeXD1 = [D1[0,0],D1[1,0]]
    else:
        RangeXD1 = [D1[1,0],D1[0,0]]

    if D1[0,1] <= D1[1,1]:
        RangeYD1 = [D1[0,1],D1[1,1]]
    else:
        RangeYD1 = [D1[1,1],D1[0,1]]

    if D2[0,0] <= D2[1,0]:
        RangeXD2 = [D2[0,0],D2[1,0]]
```

```
    else:
        RangeXD2 = [D2[1,0],D2[0,0]]

    if D2[0,1] <= D2[1,1]:
        RangeYD2 = [D2[0,1],D2[1,1]]
    else:
        RangeYD2 = [D2[1,1],D2[0,1]]

    # special case, if the two segment are perpendiculaire to the X-axis
    if (D1[0,0] == D1[1,0]) and (D2[0,0] == D2[1,0]):

        # special case, if the two segment are perpendiculaire to the X-axis and
aligned
        if D1[0,0] == D2[0,0]:
            if ((RangeYD1[0] <= RangeYD2[1]) and (RangeYD1[0] >= RangeYD2[0]))  or
((RangeYD1[1] >= RangeYD2[0]) and (RangeYD1[1] <= RangeYD2[1])) :
                intersect = True
            else:
                intersect = False
        else:
            intersect = False

    # special case, if the two segment are parrallel to the X-axis
    elif (D1[0,1] == D1[1,1]) and (D2[0,1] == D2[1,1]):

        # special case, if the two segment are parrallel to the X-axis and aligned
        if D1[0,1] == D2[0,1]:
            if ((RangeXD1[0] <= RangeXD2[1]) and (RangeXD1[0] >= RangeXD2[0]))  or
((RangeXD1[1] >= RangeXD2[0]) and (RangeXD1[1] <= RangeXD2[1])) :
                intersect = True
            else:
                intersect = False
        else:
            intersect = False

    # Find CrossX and CrossY where the segment would cross if they were infinite
    else:
        # special case, if D1 is perpendicular to the X-axis
        if D1[0,0] == D1[1,0]:
            CrossX= D1[0,0]

            # if D1 is perpendicular to the X-axis and D2 is parallel to the X -
axis
            if D2[0,1] == D2[1,1]:
                CrossY= D2[0,1]

            # if D1 is perpendicular to the X-axis and D2 is defined by Y = c * X +
d
            else:
                c = (D2[0,1] - D2[1,1])/1.0/(D2[0,0] - D2[1,0])
```

```
                d = D2[0,1]- (D2[0,0]*c)
                CrossY = c * CrossX + d

        # special case, if D2 is perpendicular to the X-axis
        elif D2[0,0] == D2[1,0]:
            CrossX= D2[0,0]

            # if D2 is perpendicular to the X-axis and D1 is parallel to the X -
axis
            if D1[0,1] == D1[1,1]:
                CrossY= D1[0,1]

            # if D2 is perpendicular to the X-axis and D1 is defined by Y = a * X +
b
            else:
                a = (D1[0,1] - D1[1,1])/1.0/(D1[0,0] - D1[1,0])
                b = D1[0,1]- (D1[0,0]*a)
                CrossY = a * CrossX + b

        # special case, if D1 is parallel to the X-axis
        elif D1[0,1] == D1[1,1]:
            CrossY= D1[0,1]

            # if D1 is parrallel to the X-axis and D2 is perpendicular to the X -
axis
            if D2[0,0] == D2[1,0]:
                CrossX= D2[0,0]

            # if D1 is parrallel to the X-axis and D2 is defined by Y = c * X + d
            else:
                c = (D2[0,1] - D2[1,1])/1.0/(D2[0,0] - D2[1,0])
                d = D2[0,1]- (D2[0,0]*c)
                CrossX = (CrossY - d)/c

        # special case, if D2 is parallel to the X-axis
        elif D2[0,1] == D2[1,1]:
            CrossY= D2[0,1]

            # if D2 is parrallel to the X-axis and D1 is perpendicular to the X -
axis
            if D1[0,0] == D1[1,0]:
                CrossX= D2[0,0]

            # if D2 is parrallel to the X-axis and D1 is defined by Y = a * X + b
            else:
                a = (D1[0,1] - D1[1,1])/1.0/(D1[0,0] - D1[1,0])
                b = D1[0,1]- (D1[0,0]*a)
                CrossX = (CrossY - b)/a
```

```
        # General case, where D1 is defined by Y = a * X + b and D2 by Y = c * X +
d
        else:
            a = (D1[0,1] - D1[1,1])/1.0/(D1[0,0] - D1[1,0])
            b = D1[0,1]- (D1[0,0]*a)

            c = (D2[0,1] - D2[1,1])/1.0/(D2[0,0] - D2[1,0])
            d = D2[0,1]- (D2[0,0]*c)

            # Special case where, D1 and D2 are parallel but not parallel or
perpendicular to an axis
            if a == c:
                # if not aligned
                if b != d:
                    intersect = False
                # if aligned
                elif ((RangeYD1[0] <= RangeYD2[1]) and (RangeYD1[0] >=
RangeYD2[0]))  or ((RangeYD1[1] >= RangeYD2[0]) and (RangeYD1[1] <= RangeYD2[1])) :
                    intersect = True
                else:
                    intersect = False
                return intersect

            CrossX = (d - b) /1.0/ (a - c)
            CrossY = (c*b - a*d)/1.0/(c - a)

        # If CrossX and CrossY are included in the x - axis and y - axis range of
D1 and D2, then they intersect
        if (CrossX >= RangeXD1[0] and CrossX <= RangeXD1[1]) and (CrossX >=
RangeXD2[0] and CrossX <= RangeXD2[1]):
            inX = True
        else:
            inX = False

        if (CrossY >= RangeYD1[0] and CrossY <= RangeYD1[1]) and (CrossY >=
RangeYD2[0] and CrossY <= RangeYD2[1]):
            inY = True
        else:
            inY = False

        if inX == True and inY == True:
            intersect = True
        else:
            intersect = False

    return(intersect)

Name = 'Full_SLAM_Input'
Name2 = 'Partial_SLAM_Input'
ClearFile(Name2)
```

```
ConesData = ReadFile(Name)
ConesXY, Color = Separator(ConesData)

StartPoint = [0,0]
CarPose = StartPoint
CarAngle = 90 #CarStartAngle
Speed = 1
OneLap = True
dist = 0
DistToNext = 0
FirstPoint = [0,0]
TimeRecord = []

AddVisibleCones(Name2, ConesXY, CarPose, CarAngle, ConesData)
StartT= time.time()
Trajectory= LocalPlanning3_0.Main(CarPose, CarAngle, FirstPoint)
ComputeTime = time.time() - StartT
FirstPoint = Trajectory[1]
t0 = time.time()
DistToNext = DistFinder(Trajectory[0], Trajectory[1])

i = 0
while OneLap == True:
    if i == 30:
        print(CarPose)
        print(CarAngle)
        exit()
    t = time.time() - t0
    CarPose, PrevPoint, NextPoint = CarPosition(Speed, t, Trajectory)
    AddVisibleCones(Name2, ConesXY, CarPose, CarAngle, ConesData)
    t = time.time() - t0
    Travelled = t * Speed
    if Travelled > DistToNext:
        print("i  ", i)
        t = time.time() - t0
        CarPose, PrevPoint, NextPoint = CarPosition(Speed, t, Trajectory)
        t0 = time.time()
        CarAngle = CarOrientation(NextPoint, PrevPoint)

       # print("CarPose ", CarPose)
       # print("CarAngle ", CarAngle)

        print(i)
        StartT= time.time()
        Trajectory = LocalPlanning3_0.Main(CarPose, CarAngle, FirstPoint)
        ComputeTime = time.time() - StartT

        #print("Len Traj  ", len(Trajectory))

        TimeRecord.append(ComputeTime)
```

```
        DistToEnd = 0
        for n in range(len(Trajectory) - 1):
            DistToEnd = DistToEnd + DistFinder(Trajectory[n], Trajectory[n +1])

        if ComputeTime * Speed > DistToEnd:
            print("ToSlow")
            print("DistToEnd")
            print(DistToEnd)
            print("Travelled")
            print(ComputeTime * Speed)
            print[TimeRecord]

            print(Trajectory)
            print(CarPose)
            print(CarAngle)
            exit()

        S1 = [Trajectory[0], Trajectory[1]]
        S2 = [[0,-2],[0, 2]]
        if  IsIntersect(S1, S2) == True:
            OneLap == False
            print("One Lap")
            print(Trajectory)
            print(TimeRecord)

            Tot = 0
            for j in range(len(TimeRecord)):
                Tot = Tot + TimeRecord[i]
            Average = Tot/1.0/ len(TimeRecord)
            print(Average)
            print(max(TimeRecord))

            exit()
        elif i > 200:
            print("exit")
            print(TimeRecord)
            exit()

        DistToNext = DistFinder(Trajectory[0], Trajectory[1])
        i = i+1
```

C – TrackReader2

```
# Theodore Bedos 28/03/21
# Script to plot image of measured temperature, and trace it using the mouse.
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt


def DistFinder(A, B):

    if A[0] >= B[0]:
        DistX = A[0] - B[0]
    else:
        DistX = B[0] - A[0]

    if A[1] >= B[1]:
        DistY = A[1] - B[1]
    else:
        DistY = B[1] - A[1]

    Dist= (DistX**2 + DistY**2)**0.5
    return Dist




#Create a variable for the image
name = "TrackAndCones2"
img= Image.open(name + '.png')
imgWidth, imgHeight  = img.size
pix = list(img.getdata())
pix = np.array(pix).reshape(imgHeight, imgWidth, 4)
type = img.mode

print(imgWidth)
print(imgHeight)


yaxisY = []
xaxisY = []
yaxisB = []
xaxisB = []
yaxisG = []
xaxisG = []
yaxisW = []
xaxisW = []
```

```
#Filter all the black pixels on the graph
for i in range(imgHeight):
    for n in range(imgWidth):
        if pix[i][n][0]>180 and pix[i][n][1]>180 and pix[i][n][2]<80:
            yaxisY.append(i)
            xaxisY.append(n)
        elif pix[i][n][0]<60 and pix[i][n][1]<60 and pix[i][n][2]>180:
            yaxisB.append(i)
            xaxisB.append(n)
        elif pix[i][n][0]<50 and pix[i][n][1]>200 and pix[i][n][2]<50:
            yaxisG.append(i)
            xaxisG.append(n)
        elif pix[i][n][0]>200 and pix[i][n][1]>200 and pix[i][n][2]>200:
            yaxisW.append(i)
            xaxisW.append(n)


# [xaxis, index] = unique(xaxis, 'last');
# yaxis = yaxis(index);

# Define the length of the green line
ymax = max(yaxisG)
y0 = min(yaxisG)
Ylen = ymax- y0
print("Ylen")
print(Ylen)

#Define origine
xaxisW = np.array(xaxisW)
yaxisW = np.array(yaxisW)
OrigineX = (xaxisW[len(xaxisW)/2] * 5) / 1.0/ Ylen
OrigineY = ((-1 * yaxisW[len(yaxisW)/2] + imgHeight) * 5) / 1.0/Ylen

# Transform tules into arrays
xaxisY = np.array(xaxisY)
yaxisY = np.array(yaxisY)
xaxisB = np.array(xaxisB)
yaxisB = np.array(yaxisB)


yaxisY= ((-1 *yaxisY + imgHeight )*5/1.0/Ylen) - OrigineY
xaxisY=(xaxisY*5/1.0/Ylen) - OrigineX
yaxisB= ((-1 *yaxisB + imgHeight )*5/1.0/Ylen) - OrigineY
xaxisB=(xaxisB*5/1.0/Ylen) - OrigineX

i = 0
while i < len(yaxisY):
    n = 0
    while n < len(yaxisY):
        if i !=n :
```

```
            dist = DistFinder([xaxisY[i],yaxisY[i]], [xaxisY[n],yaxisY[n]])
            if dist < 1:
                xaxisY = np.delete(xaxisY, n)
                yaxisY = np.delete(yaxisY, n)
                n =n-1
        n = n+1
        if n>500:
            exit()
    i = i +1
    if i>500:
        exit()

i = 0
while i < len(yaxisB):
    n = 0
    while n < len(yaxisB):
        if i !=n :
            dist = DistFinder([xaxisB[i],yaxisB[i]], [xaxisB[n],yaxisB[n]])
            if dist < 1:
                xaxisB = np.delete(xaxisB, n)
                yaxisB = np.delete(yaxisB, n)
                n = n-1
        n = n+1
        if n>2000:
            print('exit')
            exit()
    i = i +1
    if i>2000:
        print('exit')
        exit()


file = open('Full_SLAM_Input.csv', 'w')
file.truncate()
file.write("tag,x,y\n")
for i in range(len(yaxisY)):
    line = 'yellow,' + str(xaxisY[i]) + ',' + str(yaxisY[i])
    file.write(line)
    file.write('\n')
for i in range(len(yaxisB)):
    line = 'blue,' + str(xaxisB[i]) + ',' + str(yaxisB[i])
    file.write(line)
    file.write('\n')
file.close()




plt.plot(xaxisY, yaxisY, 'yo')
plt.plot(xaxisB, yaxisB, 'bo')
```

```
plt.show()
```