Path Planning and Control in an Autonomous Formula Student Vehicle

Adam Slomoi

Supervised by Professor Tom Drummond

Department of Electrical and Computer Systems Engineering
2018

Significant Contributions

- Designed an algorithm to enable a racecar to discover and drive a track laid out by cones
- Designed an algorithm which receives a continual update of new cones positions and builds a map of the race track
- Implementation of Model Predictive Control to follow a reference path at a constant speed
- Computation of vehicle dynamics of a racecar based off of discrete positions and velocities along a racetrack
- Design of a custom Automatic Differentiation class as a foundation to optimising the racing line around a racetrack
- Custom built implementation of Levenberg-Marquardt algorithm to optimise a racing line around a racetrack
- Design of above algorithm with adaptable parameters to ensure the racing line is suited to the specific car

MONASH University Engineering

Department of Electrical and Computer Systems Engineering

ECE4095 Final Year Project 2018

Adam Slomoi

Path Planning in an Autonomous Formula Student Vehicle

Supervisor: Professor Tom Drummond

Project Background

Monash Motorsport is building an autonomous race car which will compete against other driverless cars around a racetrack.

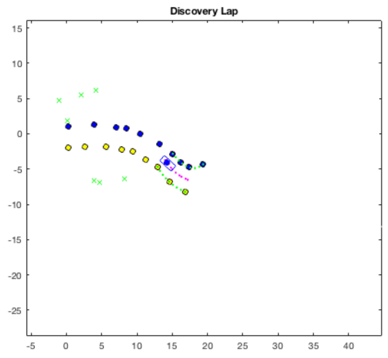
The vehicle must be capable of autonomously **planning a path** around a track.

This project showcases the development of algorithms which enable a racecar to plan a path without prior knowledge of the borders of a racetrack. Using a model of the car, the fastest route around the track is found. Finally, the car is controlled to stick to the route, allowing it to minimise its lap time.

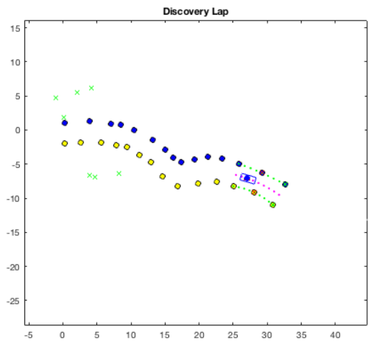
M17-E at Formula Student Germany 2018

Discovering the Track

A track is marked by cones which are initially unknown to the vehicle. It must **discover the track** by finding cones within its field of vision and continually updating its path in order to find all cones.



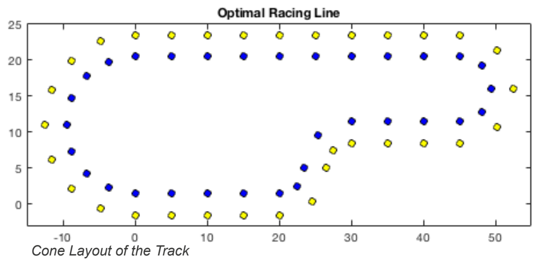
Near the Beginning of the Cone Discovery Lap



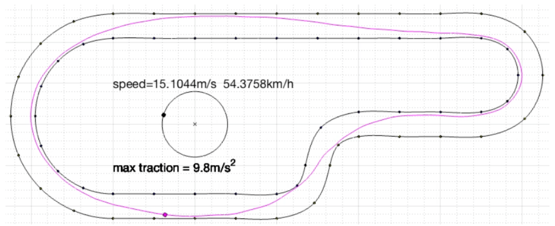
Further on in the Cone Discovery Lap

Finding the Optimal Racing Line

Once a track is known, the car will compute the **fastest route** around the racetrack. This computation forms the **ideal lap** for the car to follow.



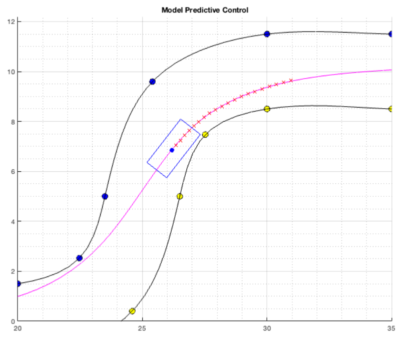
Cone Layout of the Track



Planned Racing Line

Following the Path

The optimal path is used as a reference for the car to follow. The car then uses **Model Predictive Control** to simulate how the vehicle can stick to the path by predicting what it will do in the future. It then updates the steering angle and acceleration/brake commands accordingly.



Vehicle Predicting its Future States (denoted by red x's)

Executive Summary

This report details the design of a path planning and control algorithm for a formula student vehicle. The algorithm first discovers the outline of a track. It does so by linking cones to form two track sides and implementing Model Predictive Control to drive at a slow speed along the centreline of the track. This is run continuously until the track has been discovered in its entirety, at which point the racing line is optimised on-line. The racing line is computed by optimising the positions and speeds of the vehicle at points along the track using the Levenberg-Marquardt Algorithm. This is enabled by the design of a custom-built Automatic Differentiation class in C++. The algorithm is successful in its performance however is limited primarily by the computational speed, which increases exponentially with an increase in the number of points considered along the track.

Acknowledgements

I'd like to thank all those who have assisted me in completing my FYP.

Firstly, my appreciation goes out to my supervisor, Professor Tom Drummond, for his support and willingness to provide valuable advice – even from the other side of the world.

Thanks to Sean Stannard who has been my partner in the Path Planning subsection in the Autonomous Systems section of Monash Motorsport and has worked on this with me for the year.

Autonomous Systems Coordinators Natalie Seeto and Aryaman Pandav have provided constant support and help with time-lining.

The wider Autonomous Systems team has been a great network to bounce ideas off and has provided much assistance along the way.

Thanks to the whole Monash Motorsport team for the support and helping out with vehicle dynamics-related queries.

Nomenclature

AD	Automatic Differentiation
GPS	Global Positioning System
INS	Inertial Navigation System
LMA	Levenberg-Marquardt Algorithm
LTV	Linear Time Varying
MMS	Monash Motorsport
MPC	Model Predictive Control
PRM	Probabilistic Roadmap
ROI	Region of Interest
ROS	Robot Operating System
RRT	Rapidly-exploring Random Tree
SAE	Society of Automotive Engineers
SLAM	Simultaneous Localisation and Mapping

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

This project aims to design a Path Planning system for an autonomous Formula SAE vehicle. Monash Motorsport (MMS) is building an autonomous racecar, due to be launched on Monash Open Day 2019, which will feature this project's design. The system must be safe, reliable and compliant with Formula SAE rules and regulations.

1.2 Monash Motorsport

Monash Motorsport is Monash University's student-run team which designs, builds and races Formula student cars. The team has built internal combustion vehicles since 2000 and built the first electric vehicle in 2017. The team currently operates on a two-year design cycle, with odd years consisting of a complete redesign of the cars. During even-numbered years the previous vehicles are modified and improvements are made. Monash Motorsport competes yearly in the Australian competition, and in UK, Austria and Germany on even-numbered years. In 2017 it was decided that a third vehicle would be built, being an autonomous vehicle. This vehicle will have an autonomous system overlaid on the existing 2018 electric car.

Figure 1: Monash Motorsport M16

Monash Motorsport is structured according to the following engineering sections:

- Aerodynamics
- Autonomous Systems
- Chassis
- Powertrain
- Suspension

Each section is further split into relevant subsections. The Autonomous Systems section is split:

- Cameras
- Computing
- GPS and INS
- LiDAR
- Low Voltage Systems
- Path Planning
- Vehicle Actuation

1.3 Formula Student

Formula Student is a group of student design competitions based around the design, manufacturing and racing of single-seater open-wheeled cars. The competitions are international events and comprise two types of disciplines – Static and Dynamic. Static events focus around strong business and design skills. By contrast, dynamic events test the racing capabilities of the cars on a range of racetracks.

Monash Motorsport will attend the following events in 2018:

- IMechE Formula Student (FSUK)
- Formula Student Austria (FSA)
- Formula Student Germany (FSG)
- Formula SAE-Australia (FSAE-A)

1.4 Formula Student Driverless

The Formula Student Driverless competition is scored separately to the internal-combustion and electric vehicle competitions.

The scoring is:

Table 1: Formula Student Driverless Scoring[1]

Static Events	
Business Plan Presentation	75 Points
Cost and Manufacturing	100 Points
Engineering Design	325 Points
Dynamic Events	
Skid Pad	75 Points
Acceleration	75 Points
Efficiency	100 Points
Trackdrive	250 Points
Overall	**1000 Points**

The driverless event is a relatively recent addition to Formula Student competitions, with the first event being held at FSG 2017. 15 teams entered the driverless competition and only one team, AMZ Racing from ETH Zurich, was able to successfully complete the Trackdrive discipline. In 2018, 17 teams entered the competition however only three teams were able to complete the Trackdrive event [2]. FSAE-A does not yet have a driverless competition, however several other Australian teams have also begun the development of their own autonomous cars.

1.5 Project Scope

Path planning has an integral role in an autonomous vehicle. This involves the development of a software algorithm which builds a dynamic trajectory for the car to follow. This is a crucial step between the car perceiving its environment (via sensors) and the physical actuation of the car's controls.

This project aims to develop such an algorithm which will allow for Monash Motorsport to successfully complete all dynamic disciplines at FSG 2020. As a significant number of team members join and leave the team each year, an emphasis is placed on well-documented material and justifications as to design choices. This will allow for knowledge transfer to take place from year-to-year and for the team to continually improve upon the software design.

This report showcases the theory behind the design of the path planning algorithm for all dynamic events. Given that the Trackdrive event constitutes the majority of dynamic points, it is the focus of the development. Additionally, given that the track is unknown, whereas the Acceleration and Skid Pad tracks are known, it is presumed to be an encompassing solution; if path planning is developed to complete the Trackdrive event successfully, then, with minor tweaking, it should also be able to satisfactorily plan a trajectory for the other two dynamic events.

In order to achieve this, the notion of 'path planning' is broken down into three factors:

1. Discovery Lap
2. Optimal Racing Line
3. Control

This project details the development of the first two factors listed above. Rudimentary Control was introduced to facilitate the development of the Discovery Lap and Optimal Racing Line, however its development is beyond the scope of this project.

1.6 2019 Rule Update

As of writing this report, the updated FSG2019 rules were released. Of note is the introduction of an autocross event, which consists of two single-lap runs of the same track of the trackdrive event. The rules stipulate that no prior track data can be used in the autocross event however do not forbid the use of such data in the trackdrive event [3]. Effectively, this splits the discovery lap of the original trackdrive event into its own event. As such, the path planning algorithm as everything that was designed is still required. The remainder of the report follows the previous rules.

2 Literature Review

2.1 Introduction

Historically vehicles have been human-driven and much of the engineering developments have gone into improving the capabilities and reliability of the vehicle, which are utilised by the human driver to maximise performance. Unmanned vehicles have numerous advantages over manned vehicles, including a reduction in crash casualties, environmental footprint and road congestion [4]. Accordingly, militaries have invested heavily in the development of unmanned systems technology by primarily focussing on remotely controlled systems. Such systems have been catapulted due to technological advances in communications technology since Nikola Tesla's remotely operated unmanned boat in 1898 [5]. With the advent of autonomous systems, propelled by developments in computer performance, inherent limitations in tele-operated systems are bypassed. Indeed, decision-making is reduced to the order of computer ticks and human error is replaced by precise calculations.

The search for creating a way of replacing human drivers is heavily researched in both academia and industry due to having many practical applications. An autonomous system's capacity to plan its route correctly is central to its widespread usage. As such, this literature review aims to compare different approaches to the path planning issue. The review will first quantify what a successful path plan is, then will exhibit how path planning is hierarchically structured and how each component is approached. A specific focus on how optimal racing lines are calculated will be included to find any overlap.

2.2 Defining the Optimal Route

There are many research materials on how best to develop autonomous planning due to the 'best' route being dependant on the situation. In an industrial environment, the use of autonomous robots is to maximise economic output. Translated into robotic metrics, this means an emphasis on efficiency and obstacle avoidance [6]. In military applications, the optimal path plan for an autonomous agent is one which balances maintaining stealth (i.e. avoiding enemy detection) and minimising the path length [7]. Self-driving cars have the added constraint of passenger comfort and general safety, which leads to the possibility of inter-vehicular communications for network-wide path planning [8]. Other applications for autonomous vehicles have different definitions of the optimal path. Nevertheless, an underlying definition is present among all applications: the fastest route subject to extra constraints. A path is optimal if the sum of its edge costs is minimal across all possible paths from the initial position to the final position.

2.3 Vehicle Model

In order to calculate the path of a vehicle, it is necessary to model the vehicle. A vehicle has many forces acting on it and is incredibly complex to model in its entirety. The computation of an optimal path taken by a vehicle is subject to non-holonomic constraints which adds to the complexity [9]. If the time taken to model a vehicle is too long, it can result in the planning algorithm to take longer to compute than the time it has to make a decision in real time. Camacho and Gordons demonstrate that a state space vehicular representation is attractive to both academia and industry due to its robustness and suitability for multivariable processes [10].

The most basic state space model is the kinematic bicycle model, which simplifies a vehicle to a two-wheeled model with several kinematic forces acting upon it. A more comprehensive model is the dynamic model which, though similar to the kinematic bicycle model, incorporates forces acting upon the vehicle and considers tire forces. This model is more common and is used heavily in path planning formulations [11]. Given that reducing the four wheels of a car into a bicycle model simplifies load transfer such as car roll, some path planning algorithms employ a four-wheel dynamic model. This approach is more common in applications where all wheels can be independently steered [12].

The discretised kinematic and dynamic bicycle models are compared in [10], which displays how increasing the discretisation on the kinematic model for use in a controller leads to a reduction in error. Additionally, the authors conclude that the lower computational cost of the kinematic bicycle model performed similarly to the dynamic bicycle model. By contrast, [11] employs a dynamic bicycle model for a Model Predictive Controller, placing a heavy emphasis on the importance of modelling the forces on a tire to accurately model the lateral and longitudinal forces on a car. Noting the computational burden of a nonlinear model in an online computation, the paper concludes a linearized Pacejka tire model with a dynamic bicycle model used in LTV MPC is optimal for path planning on slippery roads. Paden et al. elucidate the kinematic model's suitability in low speed path planning and that its no-slip assumption renders the dynamic model superior in more unpredictable circumstances [9].

2.4 Decision-Making Architecture

The planning of an autonomous car is the processing step between the inflow of streams of then-meaningless information from sensors, and executing the driver control tasks. What lies in between is a series of hierarchical decisions made by the autonomous system that builds context and determines the ideal vehicular commands.

For an urban environment, most hierarchies are structured according to Route Planning, Behavioural Layer, Motion Planning and Local Feedback Control[9]. The paper suggests that first a shortest path is found from a given road network, after which the autonomous system overlays road rules such as stopping and interacting with other cards. These requirements are translated into a trajectory that is then executed by actuators. Kiss and Papp merge these layers into a two-tier system which consists of global and local path planning – an initial path without any constraints and then overlaying human-like paths with "meaningful" manoeuvres [10].

In essence, most available literature propose developing a generalised reference path which can be found offline[11] and then generating a localised trajectory which deals with the dynamic environment in a region of interest[12].

2.5 Global Planning

In order to decide how to best plan a path, it is important to first determine what configuration space, or representation of the environment, can be used to do so. This is expressed by creating a roadmap which is a network of possible waypoints from which the optimal path will be produced. There are different algorithms which deal with finding the best path from point A to point B, nevertheless all algorithms have the common goal of finding the shortest obstacle-free path. Whilst in an ideal world an algorithm would have polynomial time complexity, Lazard et al. demonstrate that curvature-constrained shortest-path planning in an arbitrary environment is an NP-hard problem. Consequently, it is necessary to discretise the problem and use numerical estimation to solve for the path [13].

A popular category of algorithms is graph-search algorithms, which discretise the configuration space of the car into a graph, creating a finite number of vehicle configurations; such examples include the new PRM* [14]and Djikstra's algorithm. While these algorithms provide a global solution, they are constrained by the finite set of paths.

In order to overcome this limitation, incremental-search methods such as RRT and its improved RRT* [14] have been used. These algorithms iteratively explore the configuration space and can build an increasingly finer discretisation to find a solution. Whilst they are designed to handle nonholonomic constraints [15], these tree-like structures can have poor computational convergence.

Other methods exist, such as geometric and variational methods [9], however all are predicated on the assumption that perfect information about the configuration exists. Koenig and Likhachev propose the D* Lite algorithm, a replanning algorithm which combines the incremental and heuristic static algorithms described above, to efficiently plan in unknown terrain [16]. Further improvements include the AD* algorithm, which incorporates the replanning algorithm with an anytime algorithm – a method which produces a quick initial solution and improves it when time permits [17]. This time-sensitive adaptation is crucial in quick-reaction scenarios.

In a racing line-specific paper, Xiong contends that a nonlinear solver method is wide ranging, as opposed to methods such curve-based algorithms (Euler spiral, Bezier), which are best for corners and disregard the vehicle model, or an artificial intelligence approach which has a poor execution time [18]. Rather than approaching a track in its entirety, [19] proposes optimising curved fragments of a track independently and joining them about straight segments. This method reduces the particle swarm algorithm falling into local minima and can reduce computational time.

2.6 Local Planning

It is imperative that any planned path has the latest information about its environment so as to have an accurate and reliable path. Moving objects and incomplete information can render a global path ineffective. Therefore, more particular real-time information is incorporated in a local plan which is predicated on the global path.

Any of the aforementioned methods can be combined to produce a generalised path devoid of any regard of obstacles, and then a more detailed local plan in a region of interest, which more aptly models the real environment. In the 2007 DARPA Urban Challenge, the Stanford team used a revised A* to build a global map, and then overlaid a further variation with obstacle avoidance to get a more accurate local path [18]. Once this is computed, control is required to ensure the vehicle sticks to the path.

By comparing the performance of several low level algorithms, Snider demonstrates that control performance is heavily reliant on path smoothness and curvature continuity [19]. The low-fidelity models of path planning algorithms can also lead to violations of both control and kinodynamic constraints.

2.7 A Combined Approach

In order to mitigate the potential disconnect between the path planning and control implementation, a low-level motion planner able to deterministically handle the discontinuities of the planned path can be used. Significant research has gone into Model Predictive Control, which uses predictions of the future evolution of the vehicle and derives an appropriate next control step [20]. Li et al. demonstrate that MPC is able to track a rough reference path robustly.

2.8 Conclusion

This literature reviews provides a brief introduction into the path planning for autonomous vehicles. Most available sources propose first finding a high-level global path. General path finding algorithms such as A* are popular in autonomous path planning. Alternatively, for an environment such as a racing line, nonlinear methods can be used. Once the reference path is found, low-level planning is required. Incorporating a vehicle model is integral in finding an optimal solution for a specific autonomous agent's dynamic capabilities. Model Predictive Control is a popular low-level controller that is able to overcome approximations in a global reference path.

3 Overview

Path Planning resides on the Autonomous System architecture in between *Sensor Perception* and *Vehicle Actuation*. That is, the vehicle perceives cones, which are inputted into the path planning algorithm, which outputs steering and acceleration commands.

For the initial lap in the trackdrive event, the cones are continually updated. The algorithm must be able to continual update the path to incorporate the cones, lest it veer off course. This requires the algorithm to be fast. In order to ensure all cones are detected, the path does not need to be optimal, rather it must be *reliable*.

The racing line optimisation is extremely important in allowing the car to drive to the limit. Because of unknown track conditions (weather, poor surface, etc.) and for safety, the algorithm must have enough parameterisation to account for any imposed limitations. It needs to be fast enough to compute the racing line while it is on the track, ideally within the order of tens of seconds.

The control must be fast enough to apply controls at a similar rate to that of the physical control of the vehicle actuators (approximately 20Hz). It should follow the optimal racing line and make necessary adaptations where required. It too should be parameterised to impose any limitations sought.

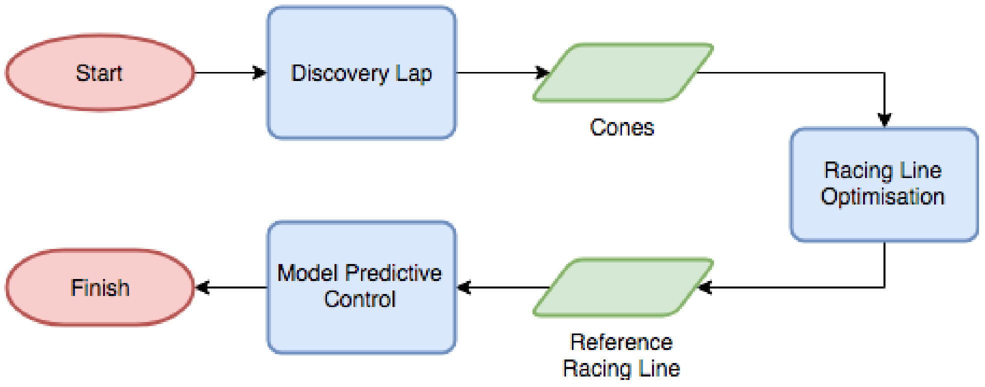


Figure 2: Path Planning Overview

4 Track Layout

4.1 Competition Specifications

The track is built up of a variety of cones of the following specifications:

big orange cone two white stripes	small orange cone single white stripe	small yellow cone single black stripe	small blue cone single white stripe
WEMAS 307.610500.00.00	WEMAS 400.000013.00.00	WEMAS 400.000013.01.10	WEMAS 400.000043.00.00
285 mm × 285 mm × 505 mm 1.05 kg		228 mm × 228 mm × 325 mm 0.45 kg	

Figure 3: Cone Specifications[21]

Each event has its own track rules defined according to D4.3, D5.1 and D5.81 for Acceleration, Trackdrive and Skid Pad respectively [1].

The track layouts are depicted in the following figures:

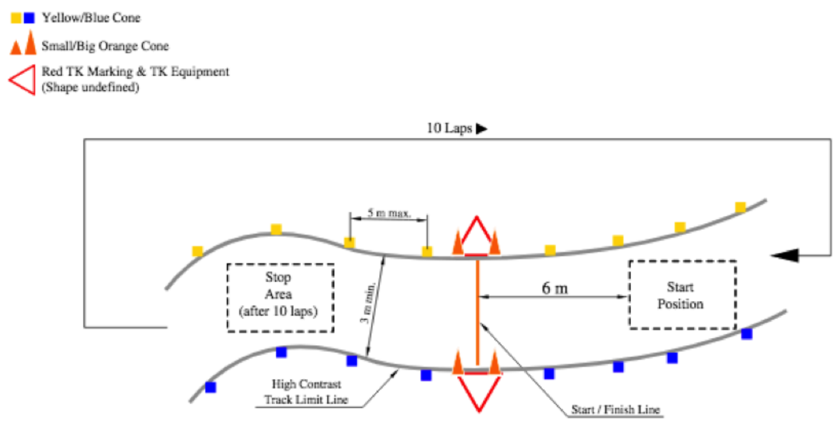


Figure 4: Trackdrive Track Layout[21]

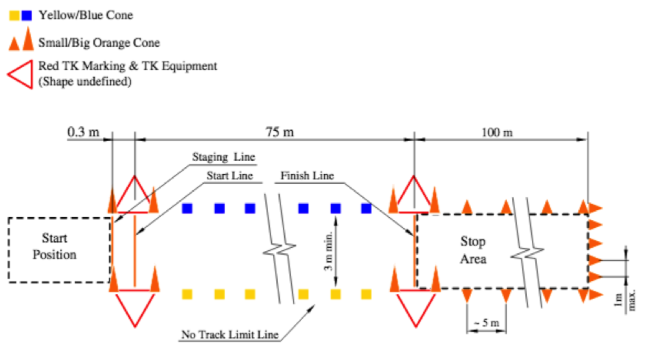


Figure 5: Acceleration Track Layout[21]

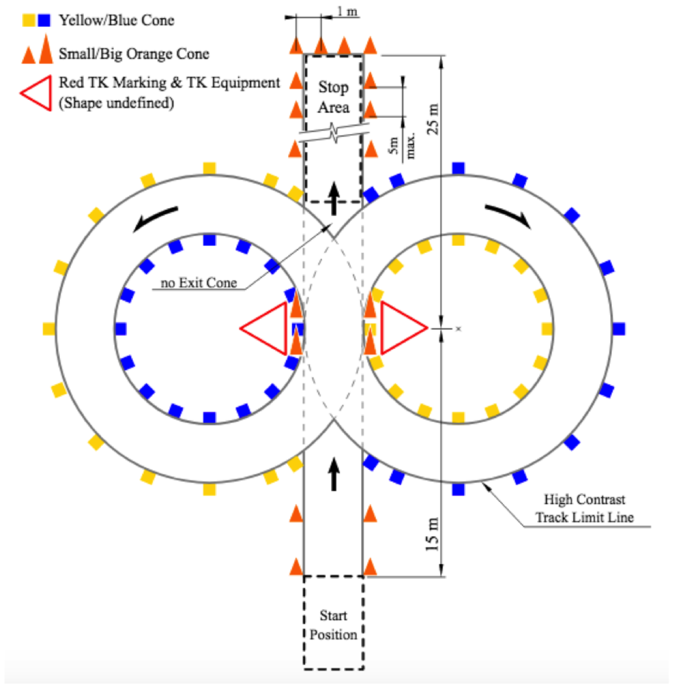


Figure 6: Skidpad Track Layout[21]

4.2 Configuration Space

Using the aforementioned track descriptions, the cone positions can be placed according to (x,y) coordinates about a global map.

The following figure demonstrates the implementation of the Skidpad track lay-out as used in MATLAB.

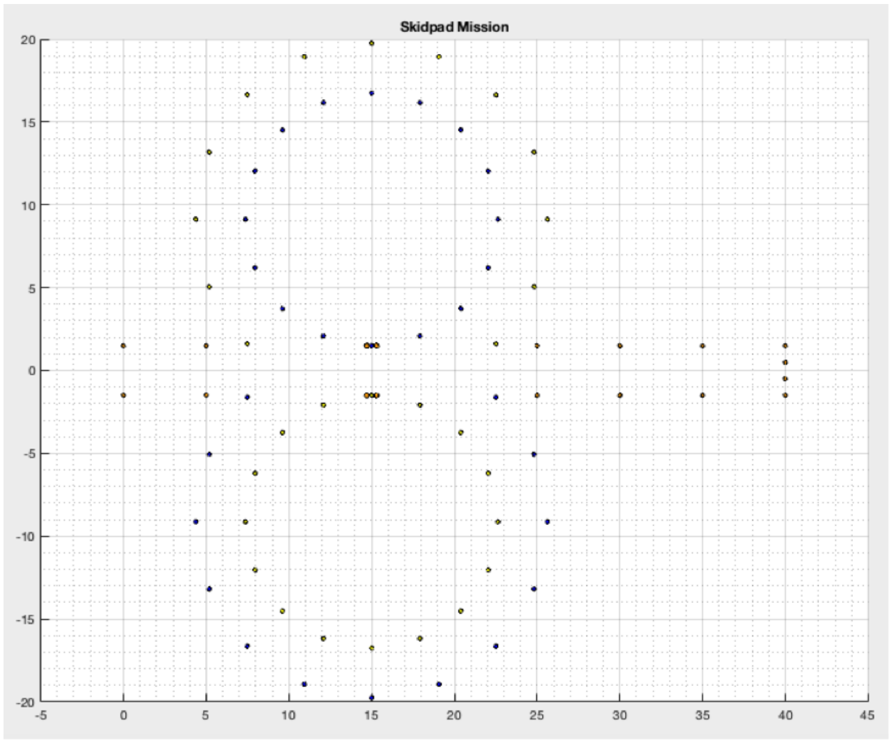


Figure 7: Skidpad Layout in MATLAB

5 Discovery Lap

The exact track and cone positions are known for the Acceleration and Skidpad events. However, the Trackdrive event involves an entirely unknown track which the car must navigate autonomously. This involves the car needing to traverse the entire track to allow its perception components to learn the environment using a SLAM algorithm. In order for this to take place, the path planning and SLAM must work concurrently, where each provides enough information to the other for it to update until the entire track has been driven and SLAM is able to close the loop of the circuit. The discovery lap is the first lap of the Trackdrive event where the car will need to navigate blindly and detect all the cones for a global path to be built.

The autonomous vehicle is using ROS as the foundation of its computing architecture. As robotic middleware, ROS will house all of the software and allow for messages to pass between processes. In the discovery lap, the perception processes will constantly update cone information which is vital for a new updated path to be built.

D8.1 of the FSG rulebook [1] stipulates the following in regards to the closed loop circuit layout for the trackdrive event:

- Straights: No longer than 80 m
- Constant Turns: up to 50 m diameter
- Hairpin Turns: Minimum of 9 m outside diameter (of the turn)
- Miscellaneous: Chicanes, multiple turns, decreasing radius turns, etc.
- The minimum track width is 3 m
- The length of one lap is approximately 200 m to 500 m.

5.1 Discovery Lap Algorithm

At the beginning of a run, the car is placed 6m from the start/finish line while the car is switched to autonomous mode. The car's state machine is switched to 'AS Ready' during which it will have ample time for its sensors to accurately detect cones (while people move behind barriers) before being switched to 'AS Driving' [1].

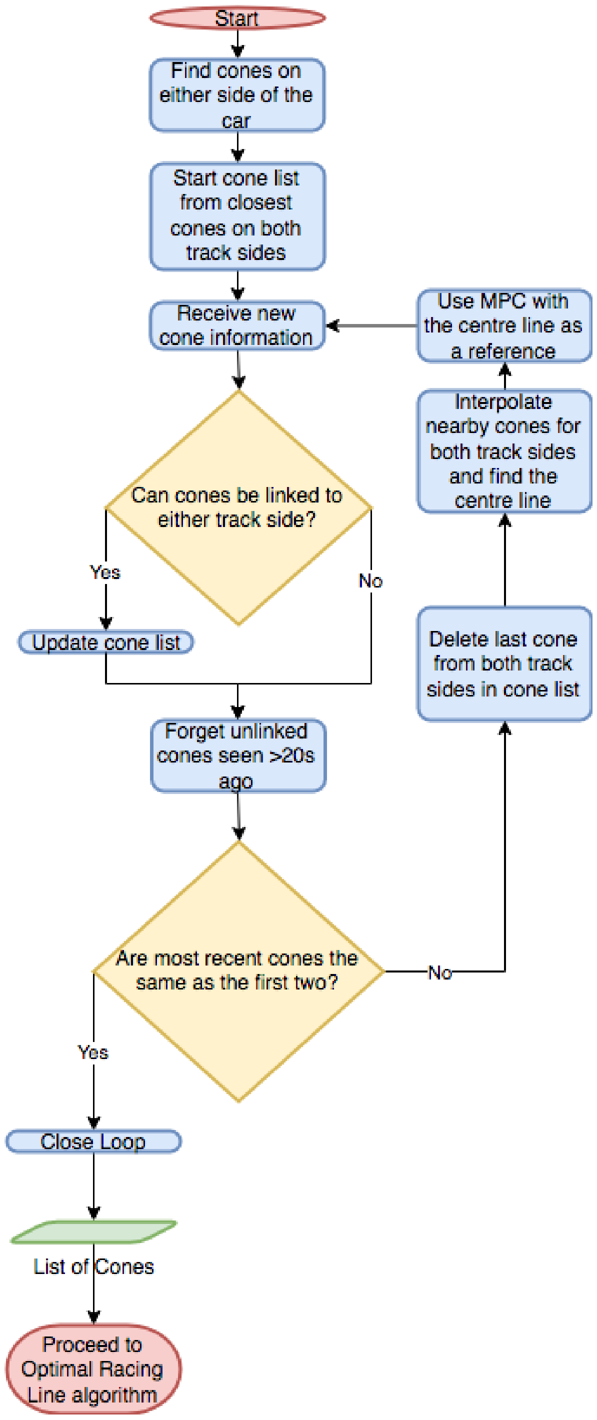

Start
Find cones on either side of the car
Start cone list from closest cones on both track sides
Receive new cone information
Use MPC with the centre line as a reference
Can cones be linked to either track side?
Yes
No
Update cone list
Interpolate nearby cones for both track sides and find the centre line
Delete last cone from both track sides in cone list
Forget unlinked cones seen >20s ago
Are most recent cones the same as the first two?
No
Yes
Close Loop
List of Cones
Proceed to Optimal Racing Line algorithm

Figure 8: Discovery Block Diagram

Initially, the car's position is constant and nearby cones are detected. Given the performance of the cameras and that the car is 6m from the big orange cones, it can be assumed that the cones will be detected, potentially along with blue and yellow cones closer to the car.

The algorithm is predicated on the notion that each of the two sides of the track, consisting of blue and yellow cones, can be linked. As such, the algorithm begins by searching for a cone to begin the left side and a cone to begin the right side. This is carried out by searching for cones within a rectangular search area from the left and right sides of the car. The following image depicts the vehicle in blue and its field of vision denoted by a circle-segment with a red 'x' for detected cones. The left and right search rectangles are shown in orange and yellow respectively. Candidate left and right start cones are shown by coloured arrows of the same colour.

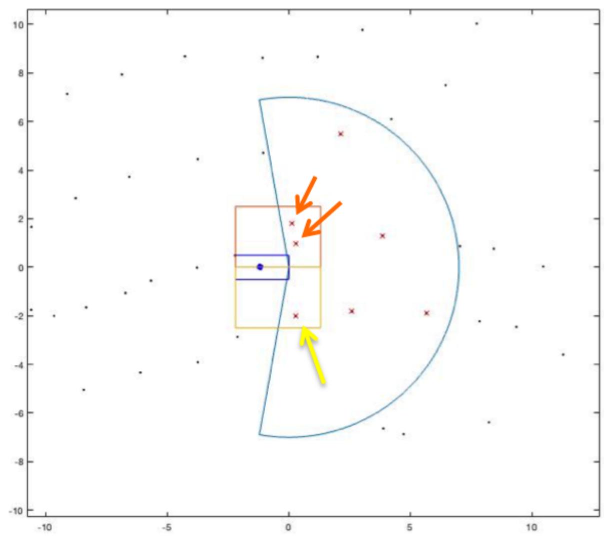


Figure 9: Finding Start Cones

Once candidate cones are found for each side the algorithm calculates the distance of each to the car, and decides upon the closest left and right cones to begin the track sides.

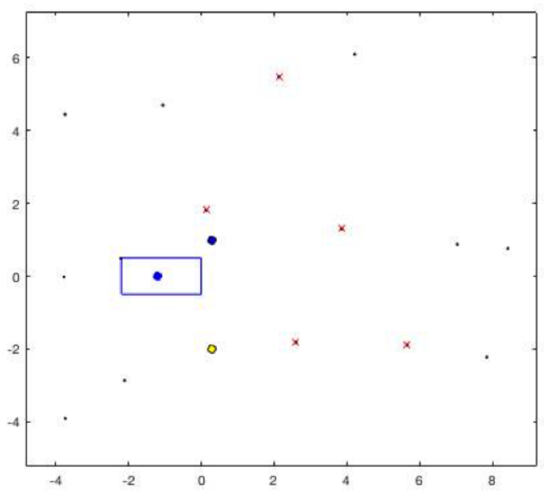


Figure 10: Setting Start Cones

A recursive process now begins in which the vehicle maps out a small path, enabling the vehicle to drive forward slightly and detect more cones, after which a new path is calculated. The path is found by searching for cones which belong to each side.

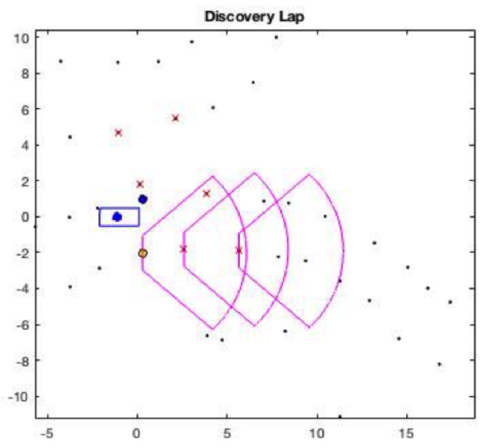


Figure 11: Searching for Additional Cones

A search area is spread from a cone, depicted by the magenta segment. The closest cone detected is linked up to the track side. The process is repeated recursively, for both sides, until no further cones can be linked to a track side.

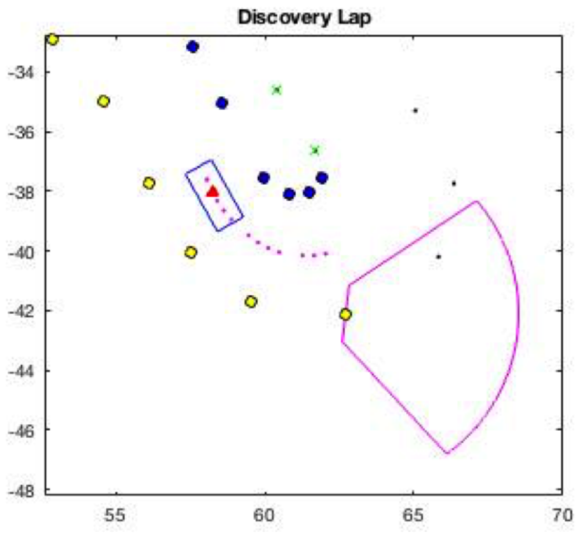


Figure 12: Cone Search Area

In order to account for curves in the track, the search area is rotated according the angle made by the most recent cone and its precursor.

Once more cones have been placed into track sides, the algorithm interpolates between cones, and create a middle line which is used as a path for the car to follow. In instances where not enough cones have been found, the car projects cones ahead and continues in a straight line, enabling more cones to be detected. The algorithm is designed to be as robust as possible. Accordingly, the algorithm does not use the cone colours as a foundation for placing the cones on either side of the track. Therefore, in a scenario where the camera system is down, and only LiDAR is detecting cones, the algorithm is still able to build a path. In the best case scenario, both cameras and LiDAR are working, in which case the cone colours detected in the real world are used to verify that the cones have been placed in the correct side of the track.

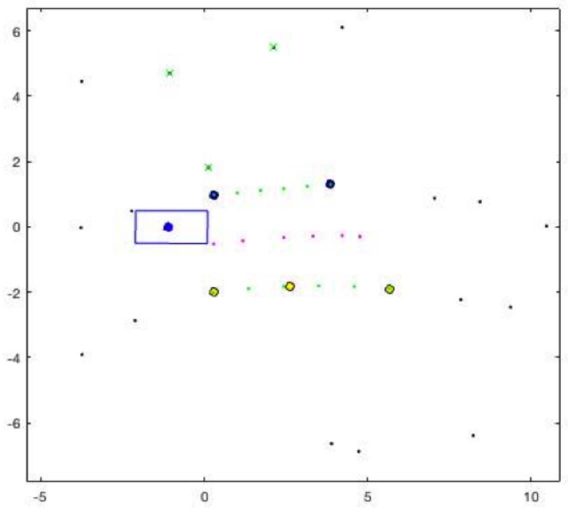


Figure 13: Centreline Reference Generation

There are situations where the algorithm places a cone on the wrong side of the track because it is the best option known to the car at that point in time. It was decided to allow this to happen, because mitigating it directly would mean severely reducing the search area for a next cone. However, in order to prevent incorrect cone placement from corrupting the path, new cones are double checked to be in the correct track side when new cones are detected. This double-check allows for erroneous cone placements to be fixed without actually impacting the planned path. This is due to cones placed on the wrong side of the track actually resulting in a middle reference path to be built which can be corrected upon receiving more track information, as shown below.

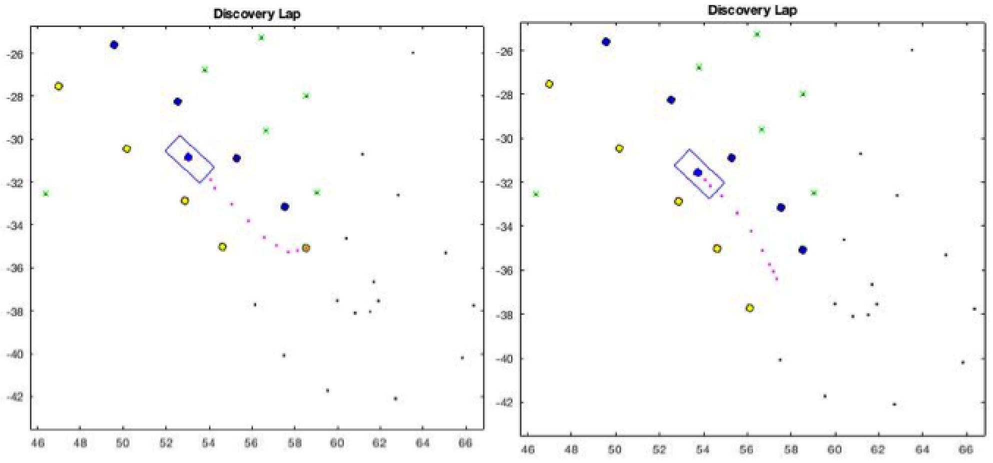


Figure 14: Initial Incorrect Cone Placement (L) Vs. Fixed Cone Placement (R)

The algorithm utilises its ability to sometimes see beyond what is required to build a path in its immediate vicinity. This vital information is stored by the algorithm to be used as future potential cones. However, given that SLAM encounters drift prior to loop closure, unused cones need to be discarded after approximately 20 seconds.

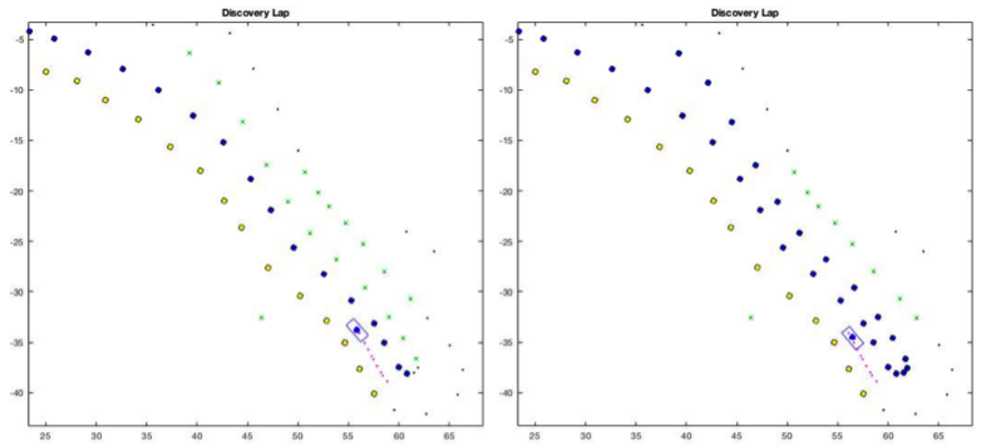


Figure 15: Potential Cones Stored in Memory *[green x's]* (L) Vs. Path Formed using Stored Cones (R)

The SLAM algorithm will use loop closure to adjust cone positions to account for drift. This will allow for the entire track to be accurately denoted by cone positions.

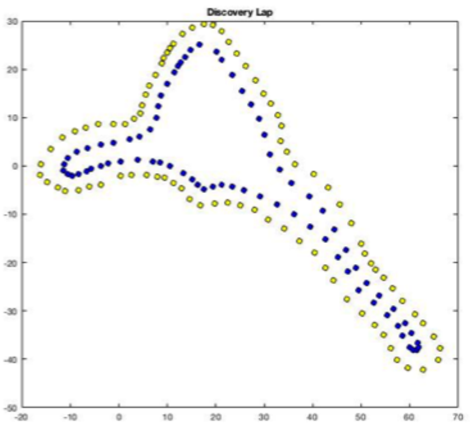


Figure 16: Completed Discovery Lap

The car uses MPC to follow the reference line at each point. The design is covered in further detail in the section about MPC. Importantly, for this part of the path planning, the goal is to drive the track as conservatively as possible. As such, the controller ensures the vehicle travels at a constant slow speed and tries to stick to the reference line, which is at the centre of the track.

6 Optimal Racing Line

With the layout of a track, it is possible to calculate the fastest possible lap time. It is crucial to have a race car driving, within it limits, as fast as possible in order to win a race. This section describes the minimisation of the lap time for the vehicle. As the vehicle will have just received the cone locations at the end of its first lap in the trackdrive event, this computation needs to be quick and computed on-line. For the acceleration and skidpad events, this same computation can be used with the known cone locations. The result of the optimal racing line algorithm provides a theoretical trajectory for the car to follow to obtain its minimum possible lap time.

6.1 Track Discretisation

As discussed in [13], it is important to discretise a track in order for an optimal path to be planned. Using concepts of integration, it is possible to understand a path as a sequence of infinite points along a track. Effectively, the continuous path is a sequence of points along an infinite number of transverse lines along the track. This path is then reduced from an infinite (continuous) number of points, to a finite (discrete) amount.

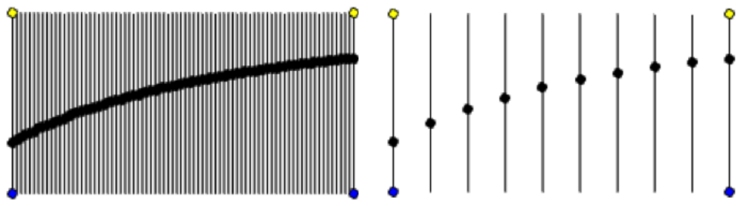


Figure 17: Continuous Path (L) Vs. Discrete Path (R)

In order to achieve this discretization, the transverse lines are needed. There is an embedded trade-off between the accuracy of the track and any associated computational time; the more the track is discretised, any computations related to the track become faster, due to having fewer track lines through which to iterate, however the track begins to represent the reality less and less, and vice versa. As such, it is important to have the number of transverse lines a tuneable parameter, denoted throughout this report by 'N'. Naturally, this value will vary from track to track, given the different tracks are comprised of different shapes. For example, a straight can be discretised heavily, due to having no lost information upon discretisation. By contrast, a hairpin-turn or chicane can have critical information lost if not enough track information is kept.

The track is built by interpolating between the cone positions that have been detected. By interpolating individually for the left and right track sides, many imaginary cones can be placed on the track. They are paired to make transverse lines and as many such imaginary cones are kept as required, becoming the new track configuration space.

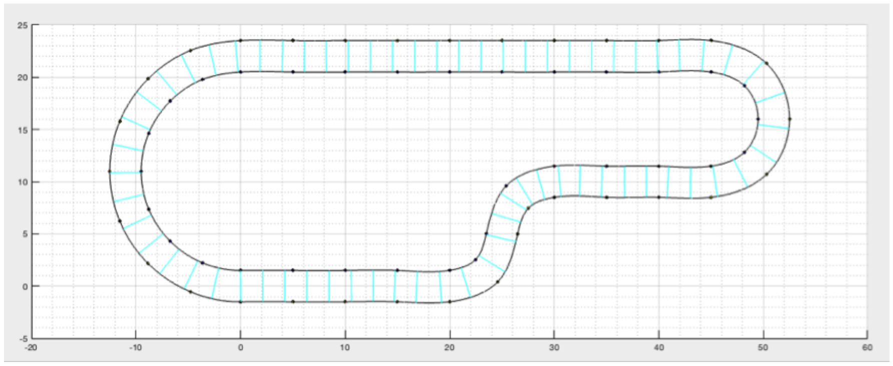


Figure 18: S-bend Track Transverse Lines

Figure 18 depicts the transverse lines of an S-bend track. The original cones can be seen as dots along the track. The interpolation of each trackside is shown by the black lines, from which, in this case 66, transverse line pairs are chosen to represent the track.

6.2 Racing Line Representation

Using the track configuration space derived from the cones, it is possible to created an optimal racing line. Each transverse line is given an associated position and speed. The position is a proportion along the line, where a value of 0 corresponds to the inner side, and 1 corresponds to the outer side. Speed is a value at the corresponding position in metres per second.

These inputs are stacked are expressed as vectors. For N transverse lines, the racing line can be expressed according to Table 2.

Table 2: Racing Line Vector Representation

Point	**Position (P)**	**Speed (S)**
0	P_0	S_0
1	P_1	S_1
2	P_2	S_2
3	P_3	S_3
N-2	P_{N-2}	S_{N-2}
N-1	P_{N-1}	S_{N-1}

6.3 Leapfrog Integration

As the speed and position are the parameters which define the racing line, it is necessary to derive further kinematic parameters which are used to ensure that the racing line is indeed optimal and attainable.

Leapfrog integration is a stable second-order numerical integration method which is common in dynamical systems. The method staggers time-derivatives of position about interleaved points. This theory is used as the basis of the formulation of the kinematic equations which derive such values from the position and speed vectors.

$$\overline{x}_i = p_i \cdot \overline{O}_i + (1 - p_i) \cdot \overline{I}_i$$

$$\overline{d}_i = \overline{x}_{i+1} - \overline{x}_i$$

$$d_i = \sqrt{\overline{d}_i^{\,T} \cdot \overline{d}_i}$$

$$\Delta t_i = d_i \cdot \frac{2}{s_i + s_{i+1}}$$

$$\overline{v}_i = \frac{\overline{d}_i}{\Delta t_i}$$

$$\overline{a_G}_i = (\overline{v}_i - \overline{v}_{i-1}) \frac{2}{\Delta t_i + \Delta t_{i-1}}$$

$$\overline{a}_i = \begin{bmatrix} \left(\frac{\overline{d}_i}{d_i}\right)_1 & -\left(\frac{\overline{d}_i}{d_i}\right)_0 \\ \left(\frac{\overline{d}_i}{d_i}\right)_0 & \left(\frac{\overline{d}_i}{d_i}\right)_1 \end{bmatrix} \times \overline{a_G}_i$$

$$a_i = \sqrt{\overline{a}_i^{\,T} \cdot \overline{a}_i}$$

$$\overline{j}_i = \frac{\overline{a}_{i+1} - \overline{a}_i}{\Delta t_i}$$

$$Pwr_i = m \cdot \overline{a}_{i0} \cdot s_i$$

where
p is a Nx1 position vector
s is a Nx1 speed vector
$\overline{\mathbf{I}}$ is a Nx2 inner-cone matrix where each row is (x,y) coordinates
$\overline{\mathbf{O}}$ is a Nx2 outer-cone matrix where each row is (x,y) coordinates
$\overline{\mathbf{x}}$ is a Nx2 path coordinate matrix where each row is (x,y) coordinates
$\overline{\mathbf{d}}$ is a Nx2 matrix of segment differences where each row is (x,y) coordinates
d is a Nx1 distance vector
$\mathbf{\Delta t}$ is a Nx1 time vector
$\overline{\mathbf{v}}$ is a Nx2 velocity vector in a global (x,y) coordinate frame
$\overline{\mathbf{a_G}}$ is a Nx2 acceleration vector in a global (x,y) coordinate frame
$\overline{\mathbf{a}}$ is a Nx2 acceleration vector in a local car frame where each row is (lateral,longitudinal)
a is a Nx1 acceleration magnitude vector
$\overline{\mathbf{j}}$ is a Nx2 jerk vector in a local car frame where each row is (lateral,longitudinal)
m is the mass of a vehicle
Pwr is a Nx1 power vector

The following graphic depicts the structure of the leapfrog integration equations shown above. It can be seen that successive derivatives of position with respect to time vary from being assigned to a node and to an edge. This ensures the kinematics remain as accurate as possible.

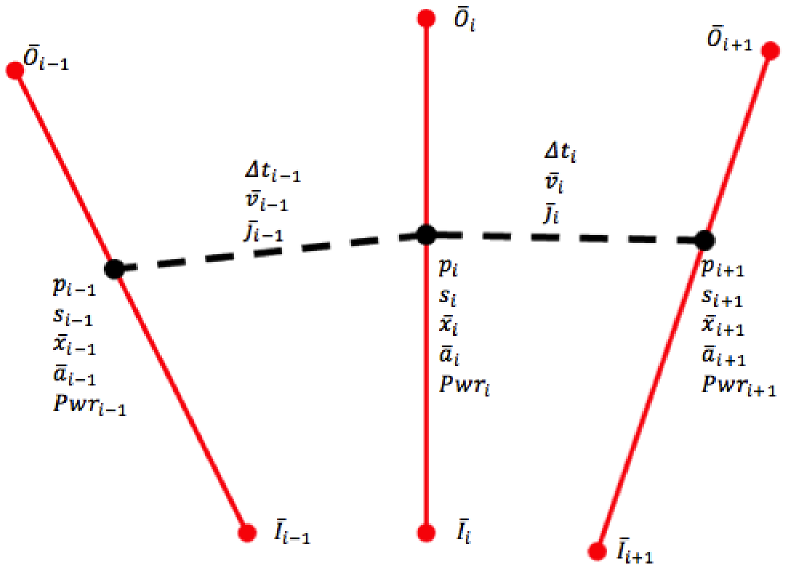


Figure 19: Leapfrog Integration

6.4 Constraints

The path now has values for time, position, velocity, acceleration, jerk and power at each point. Using these values, and known vehicular limits, the formulation of a minimisation function can be built.

The optimal racing line is structured in order to minimise time subject to constraints on: exiting the track, exceeding imposed speed limits, and vehicular limits on acceleration, jerk and power.

Upon consultation with Monash Motorsport team members as well as viewing vehicular data analysed with a MoTeC M150 management system , the following constraints are applied:

Table 3: Vehicular Constraints

Parameter	**Minimum**	**Maximum**
Acceleration ($^m/_{s^2}$)	0	9.8
Lateral Jerk ($^m/_{s^3}$)	-74	74
Longitudinal Jerk ($^m/_{s^3}$)	-90	60
Power (kW)	*none*	$80,000$

6.5 Optimisation

6.5.1 Minimisation Formulation

The optimal racing line occurs when the steady-state lap time around the track is as fast as possible subject to constraints inherent in a vehicle as well as those imposed externally.

Given this, the minimisation function is built to minimise lap time and add a time penalty for any breach of constraints. These penalties correspond to the residuals described below in the Levenberg-Marquardt Algorithm.

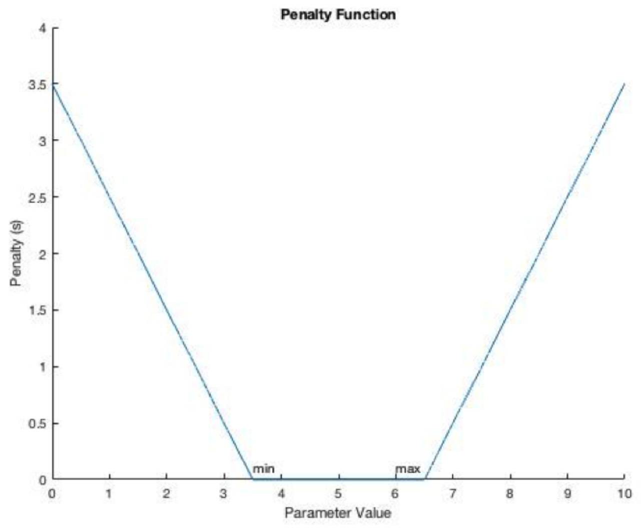


Figure 20: Penalty Function

The above figure displays the penalty function, where any parameter value placed between the minimum and maximum allowable values has no associated penalty, while anything in excess has an increasing penalty. In order to standardise all parameters to time as they are on different scales, the gradients of the excess penalties can be adjusted as required. By implementing soft constraints, the optimisation is allowed to initially break the constraints, at a penalty, to hasten its convergence.

The optimisation function is given by:

$$S = \sum_{i=0}^{N-1} (\Delta t_i + pen(s_i) + pen(p_i) + pen(a_i) + pen(j_{lat\,i}) + pen(j_{lon\,i}) + pen(pwr_i))$$

6.5.2 Levenberg-Marquardt Algorithm

The Levenberg-Marquadt Algorithm is an optimisation method used to solve nonlinear least-squares problems. The algorithm is a combination of the Gauss-Newton algorithm and the gradient descent method. It does this through the use of a damping factor which updates as the minimisation progresses. As the damping factor increases, LMA approximates gradient descent; as it decreases, LMA approximates Gauss-Newton. As the parameters being minimised tend towards their optimal value, the method acts like Gauss-Newton [22]. The algorithm requires an initial guess and can converge to local minima.

The LMA formula aims to minimise the sum of squares:

$$S(\boldsymbol{\beta}) = \sum_{i=0}^{N-1} {r_i}^2(\boldsymbol{\beta})$$

Using the following equation:

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - (\boldsymbol{J}^T\boldsymbol{J} + \lambda \cdot diag(\boldsymbol{J}^T\boldsymbol{J}))^{-1} \cdot \boldsymbol{J}^T\boldsymbol{r}$$

where
$\boldsymbol{\beta}^T = (p_0, p_1, p_2, \ldots, p_{N-1}, s_0, s_1, s_2, \ldots, s_{N-1})$
$\boldsymbol{J}_{i,j} = \frac{\partial r_i(\boldsymbol{\beta}_t)}{\partial \beta_j}$
$\boldsymbol{r(\beta)} = (\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta}))$
λ is the damping factor

λ is a variable parameter which affects how the minimisation occurs. In order to ensure it is at its best value, at each iteration the value is updated. If at iteration 'j' $S(\boldsymbol{\beta})_j < S(\boldsymbol{\beta})_{min}$ then λ is decreased by a 'damp down' factor, otherwise if $S(\boldsymbol{\beta})_j \geq S(\boldsymbol{\beta})_{min}$, λ is increased by a 'damp up' factor.

6.5.3 Automatic Differentiation

In order to overcome the computational expense of dynamically generating Jacobian matrix analytically, an alternate approach is used. Automatic Differentiation is a set of techniques which numerically calculates the derivative of a given function. AD works by implementing the chain rule to find the derivative as all computer computations can be broken into simple arithmetic operations (addition, subtraction, multiplication, division, etc.). The result is as accurate as its numerical precision and is useful when the numerical model changes over time [23].

The foundation equations of AD are the following rules in differentiation: Sum Rule: $\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$
Product Rule: $\frac{d(u \cdot v)}{dx} = v \cdot \frac{du}{dx} + u \cdot \frac{dv}{dx}$

Quotient Rule: $\frac{d(u/v)}{dx} = \frac{v \cdot \frac{du}{dx} - u \cdot \frac{dv}{dx}}{v^2}$
Power Rule: $\frac{du^v}{dx} = v \cdot u^{v-1}$

Automatic differentiation is suitable in this application due to a new Jacobian needing to be calculated for every optimal racing line input.

The foundation of the algorithm is the custom AD class, structured as follows:

Table 4: AD Class

AD Class
• Value • Derivatives • Size
• AD() • getdp() • getds()

Value: A value which corresponds to the value of the class
Derivatives: A vector which is a column stack of $\frac{d}{dp_i}$ and $\frac{d}{ds_i}, i \in \mathbb{Z} \cap [0, N-1]$
Size: A value which corresponds to the length of the derivatives vector, $2N$, where N is the number of transverse lines
AD: A constructor for the AD class
getdp(i): An accessor to Derivatives for the partial derivative of Value with respect to position at tranverse line 'i'
getds(i): An accessor to Derivatives for the partial derivative of Value with respect to speed at tranverse line 'i'

The following is a demonstration of how the AD works. For simplicity, the AD class will comprise only a value and its derivatives.

Assume we have the following three AD classes:

A		**B**		**C**	
Value	1	*Value*	4	*Value*	-2
Derivatives	$\begin{pmatrix} 1.5 \\ -2 \\ 2 \end{pmatrix}$	*Derivatives*	$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$	*Derivatives*	$\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$

We seek to perform the following operation $D = (A + C) \cdot B$.

Utilising the sum rule, A+C is calculated.

A+C	
Value	-1
Derivatives	$\begin{pmatrix} 2.5 \\ -2 \\ 3 \end{pmatrix}$

Now, D can be calculated using the product rule.

D	
Value	-4
Derivatives	$\begin{pmatrix} 9 \\ -10 \\ 9 \end{pmatrix}$

The above demonstration shows the ease with which AD is used. Given that many kinematic values are needed in computation of the optimal racing line, and it needs to be known how each changes with respect to its and neighbouring points' speeds and positions, AD is a suitable choice for deriving the Jacobian matrix.

6.5.4 Constraint Implementation

The optimisation function was initially defined as

$$
\begin{aligned}
S = \sum_{i=0}^{N-1} (\Delta t_i) \\
\text{subject to} \\
0 \leq & p_i \leq 1 \\
0 \leq & s_i \leq maxSpeed \\
0 \leq & a_i \leq 9.8 \\
-74 \leq & j_{lat\,i} \leq 74 \\
-90 \leq & j_{lon\,i} \leq 60 \\
& Pwr_i \leq 80,000
\end{aligned}
$$

This is known as *hard contraints* implementation due to the constraints being immutable. Whilst this may seem logical, it can cause complexities in carrying out the optimisation. This is due to the computation not exceeding the constraints *at all*, meaning that constraints cannot be temporarily exceeded to minimise the overall function and then tightened again. This causes severe issues in an implementation as sophisticated as finding the optimal racing line given that the many parameters being optimised need to be within the six aforementioned constraints at all times. This limitation resulted in convergence to local minima which was visibly not the optimal solution. Additionally, the implementation relied heavily on the MATLAB *fmincon* function, which was extremely slow.

To combat this issue, *soft constraints* were used instead of the hard constraints. Section 6.5.1 details the generalised implementation of these constraints. The points beyond which constraints are penalised are the same values as for the hard constraint implementation. Soft constraints require fine tuning due to the relative differences between exceeding constraints despite all being of the form depicted in 20. For example, exceeding the track by 60cm at one instance would result in a penalty of 0.2s, whereas going $0.2^m/_s$ over the imposed speed limit would result in the same penalty, despite being significantly less important. Accordingly, each constraint has its excess penalty function scaled by a factor to normalise them, ensuring that relative penalties have equitable constraints. As such, speed, acceleration and jerk are scaled by $\sim 10^{-1}$ and power is scaled by $\sim 10^{-5}$. Fine tuning can be made beyond this however would not significantly impact the result.

The solution of the result is not what would be expected. In order to demonstrate this, figure 21 depicts the output of the minimisation truncated before reaching its optimal solution. The lap time at this truncated point is 14.03s. It is clear that the car is turning excessively on straight parts of the track.

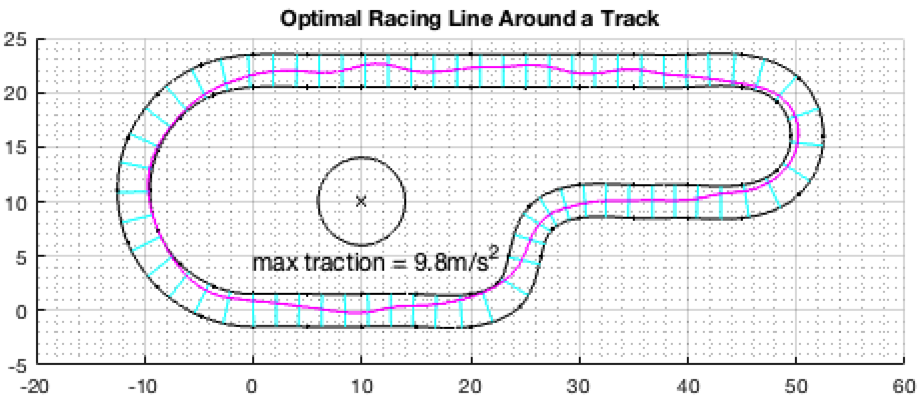


Figure 21: Soft Constraints without Regularisation

In order to mitigate this, regularisation is used on acceleration. Figure 22 displays how smoothing is used to make the car drive according to a preferable racing line. This image is taken using the same constraints and number of iterations as 21, but with the addition of regularisation. Given that it is also truncated before reaching the optimal racing line, it is not the ideal trajectory, however it aptly portrays the efficacy of adding regularisation. This is reinforced in the track time, which is 13.36s as opposed to 14.03 seconds.

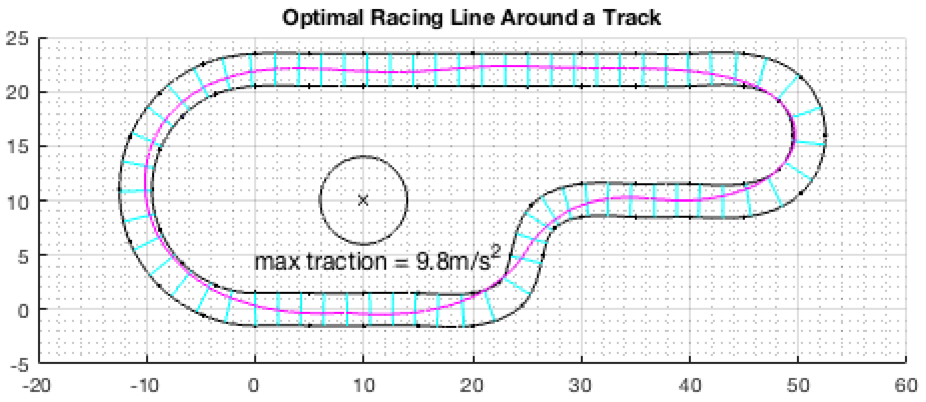


Figure 22: Soft Constraints with Regularisation

Regularisation is a technique used to introduce additional information to a problem which is ill-posed. In this case, the regularisation term is the introduction of a penalty on *any* acceleration. This ensures the algorithm only accelerates when necessary. Prior to regularisation the minimisation function is

$$S = \sum_{i=0}^{N-1} (\Delta t_i + pen(s_i) + pen(p_i) + pen(a_i) + pen(j_{lat\,i}) + pen(j_{lon\,i}) + pen(pwr_i))$$

where

$$pen(x) = \begin{cases} m(x - min) & x < min \\ 0 & min \leq x \leq max \\ m(max - x) & x > max \end{cases}$$

min is the minimum allowable value
max is the maximum allowable value
m is the normalisation gradient

The new minimisation function is :

$$S = \sum_{i=0}^{N-1} (\Delta t_i + pen(s_i) + pen(p_i) + pen(a_i) + pen(j_{lat\,i}) + pen(j_{lon\,i}) + pen(pwr_i) + reg(a_i))$$

where

$$pen(x) = \begin{cases} m(x - min) & x < min \\ 0 & min \leq x \leq max \\ m(max - x) & x > max \end{cases}$$

$$reg(x) = m \cdot x$$

min is the minimum allowable value
max is the maximum allowable value
m is the normalisation gradient

Placing this formulation in MATLAB's optimisation function obtains the optimal racing line with a lap time of 11.01s.

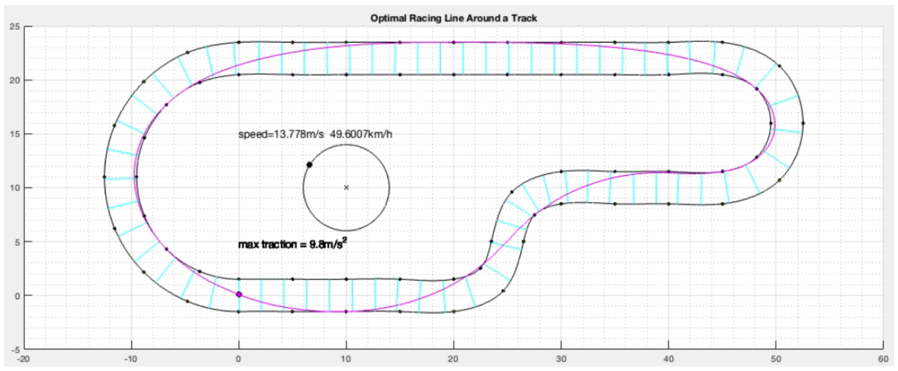


Figure 23: Optimal Racing Line in MATLAB

6.5.5 Choosing Parameters

A test scenario was set up to isolate each constraint to identify a suitable gradient factor. The scenario involves running 100 iterations and cycling through a range of factor values for a specific constraint and determining which results in the lowest lap time. Once found, a new constraint was added and the process repeated. Table 5 displays the parameter values in order of implementation from top-to-bottom.

Table 5: Minimisation Parameter Values

Constraint	**Type***	**Gradient**	**Minimum**	**Maximum**
Position	pen	10	0	1
Speed	pen	20	0	*variable*
Accel.	pen	0.1	0	9.8
Accel.	reg	10^{-3}	-	-
Power	pen	10^{-6}	-10^7	$80 \cdot 10^6$
Lat. Jerk	pen	0.01	-74	74
Lon. Jerk	pen	0.01	-90	60

*Types are according to the equations declared in section 6.5.4

With all constraints set according to Table 5, the values of the damping factors can be tuned. A range of damping factors, for both λ_{up} and λ_{down}, are evaluated according to the cost in table 6. The results show that there is no clear pattern to determine the best damping factors.

Table 6: Damping Factor Values

Damp Up	**Damp Down**	**100 Iter. Cost (s)**	**500 Iter. Cost (s)**
2	2	15.05	11.23
2	3	15.07	11.59
3	2	**13.79**	11.05
5	5	18.41	11.83
5	1.5	15.02	**11.04**
10	10	15.55	11.89
11	9	15.07	11.59
15	15	16.24	13.37

$\lambda_0 = 0.001$ for all tests

Slight variations in the damping factors can cause drastic changes in optimisation performance. While the results indicate small values for the damping factors, it is commonplace in software libraries to default the values to $\lambda_{up} = \lambda_{down} = 10$ [24]. The significance of the damping factor choice is more prominent in an optimisation with a limited number of iterations. Table 6 shows that after a few hundred iterations that the difference in choice of damping parameters only leads to a cost variation in the order of milliseconds.

The starting value of λ_0 impacts the convergence time of the algorithm. The fastest convergence occurs with an initial $\lambda = 10^2$ which is significantly larger than $\lambda = 10^2$ which recommended in [22]. Comparing the costs of Tables 6 and 7 highlight that there is pattern in deciding upon parameter values. Many of the parameters are interdependent, giving rise to the difficulty in analytically choosing values.

Table 7: Initial Damping Factor

$\boldsymbol{\lambda_0}$	**100 Iter. Cost (s)**
10^3	14.18
10^2	**13.75**
10	14.19
1	15.25
10^{-1}	14.37
10^{-2}	14.83
10^{-3}	15.05
10^{-4}	15.10

$\lambda_{up} = \lambda_{down} = 2$ for all tests

6.6 Component Scaling

Marquadt's insight to LMA provides the scaling of λ by $diag(\boldsymbol{J}^T\boldsymbol{J})$ in $\boldsymbol{J}^\dagger$, which scales each component of the gradient by its own curvature – avoiding slow convergence in directions of small gradients. Counter-intuitively, this implementation causes slower convergence than Levenberg's traditional scaling of λ by $\mathbf{I}$. $\mathbf{I}$ is altered slightly so that components on the diagonal related to p are 1, and components related to s are 0.01.

6.7 Performance

The MATLAB function uses the interior point method to find the optimal racing line. It employs additional back-end techniques which produce a faster result. As this is not directly convertible to C, a custom optimisation function was built, LMA, as described in section 6.5.2. In their article, Li et al. demonstrate that the Levenberg-Marquardt method has higher computational precision and a faster convergence speed than the path-following interior point method [25]. Whilst this would vary on a case-by-case basis, this highlights that LMA is an effective optimisation method.

Implementation of LMA in C++ achieves the expected result at a significant perfomance boost over MATLAB. Table 8 demonstrates that LMA in C++ is 6.5x faster than MATLAB's interior point method, and converges in half the number of iterations. Beyond this, LMA is able to reach a far lower convergence value if allowed to continue running, making it a far superior algorithm.

Table 8: MATLAB Vs. C++

	Iterations	**Execution Time (s)**	$\frac{s}{iter}$
MATLAB	404	50.9153	0.1260
C++	217	7.9752	0.0368

Performance to reach a cost of 11.2s

The number of transverse lines severely limit the performance of the algorithm. Figure 24 delineates the exponential relationship between the number of transverse lines and execution time. This is largely due to the pseudoinverse calculation; for 150 transverse lines, the algorithm spends **95.04**% of the execution time calculating $B_{i+1} = B_{best} - Jac^\dagger r$.

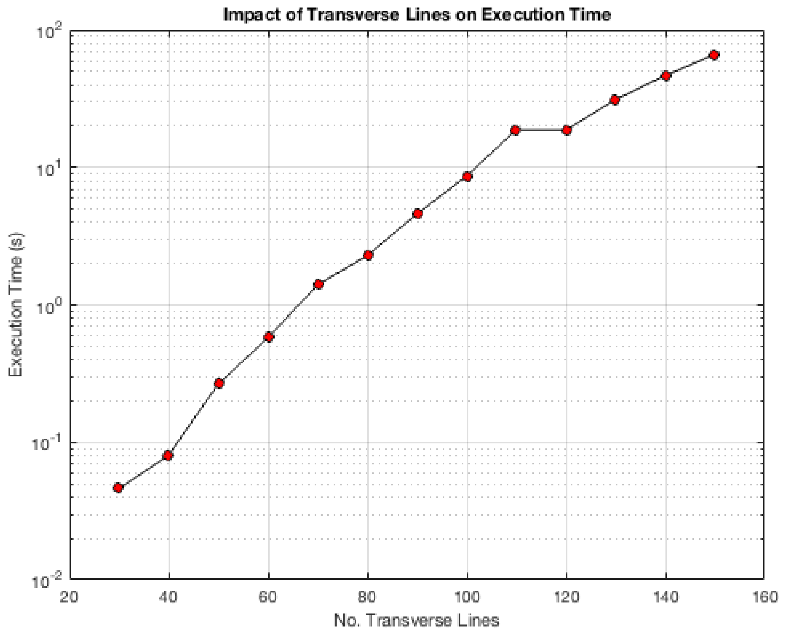


Figure 24: Impact of Transverse Lines of Execution Time

The above figures and tables highlight the sensitive nature of the LMA. Convergence time is largely dependant on the number of transverse lines in the problem rather than the parameter choices. Whilst parameter selection *does* affect the execution time of the algorithm, its impact pales in comparison to the degree with which size of the matrices in the problem affect computation time. Assuming a reasonable selection of parameters, the algorithm performance is beyond human input.

7 Model Predictive Control

Model Predictive Control is used to control the car to the optimal racing line. Given that the car will not start driving according to the racing line, it needs to be controlled onto it. Additionally, because the racing line is a theoretical calculation, real world variations may result in the car going slightly off the desired track. As such, control is necessary to allow for the car to respond to any changes. This is, in essence, the calculation of the local trajectory of the car based off of the global path computed by the faster racing line algorithm. MPC is able to deal with constraints as well as non-linear systems [26], making it ideal for vehicular implementation.

7.1 MPC Overview

The following figure demonstrates how MPC fits into the system. ‘u’ refers to the vehicle controls and ‘Z’ refers to the vehicle state.

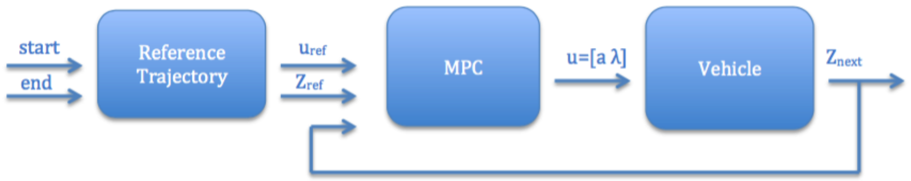


Figure 25: MPC Block Diagram

The following table details each variable in Z which describe the vehicle state.

Table 9: State Parameters

Parameter	Description
X	X-coordinate of the vehicle’s centre of mass
Y	Y-coordinate of the vehicle’s centre of mass
Ψ	The angle the car’s centre of mass is travelling with respect to the vechicle’s longtitudinal direction
v_x	The velocity of the centre of mass in the x-direction
v_y	The velocity of the centre of mass in the y-direction
$\frac{d\Psi}{dt}$	The rate of change of the angle the car’s centre of mass is travelling with respect to the vechicle’s longtitudinal direction

The following table details each variable in u which describe the vehicle controls.

Table 10: Control Parameters

Parameter	**Description**
a	The vehicle's acceleration (determined the the throttle/brakes)
λ	The vehicle's wheel angle

The problem becomes a control optimization problem in which the goal is to minimise the car's error from the optimal racing line in terms of position and speed. Exact formulation of the MPC differs in different implementations but there are four common stages [26]:

1. Start at a time interval and predict a finite number future inputs and outputs for the system
2. Derive a cost function predicated on the future control inputs and state outputs. Optimise the function with respect to the input control signals.
3. Input the first of the future control signals to the system
4. Wait for an updated system state (next time step) and repeat Stage 1

The design takes place in step 2, for which an optimisation function is formulated.

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & \sum_{i=1}^{H} J(Z_i, u_i) \\ \text{subject to} \quad & Z_{i+1} = f(Z_i, u_i) \\ & C(Z_i, u_i) \le M \end{aligned}$$

where
J is the cost function
H is the horizon (number of future predictions to consider)
f is the vehicle prediction model
C is the vehicular constraints

7.2 Initial Implementation

As discussed earlier, the MPC used thus far has been a basic version to enable further development of the discovery lap algorithm. This section discusses the overarching design principles used rather than delving into the specifics of the implementation.

The design uses the simplest of the vehicle representations – the kinematic bicycle model. It was set to have a constant speed of $5 m/s$ in order to reduce the complexity of adding a reference speed. The only reference used is a path, which the algorithm attempts to minimise its sum of squared errors to this path for 20 time horizons ahead.

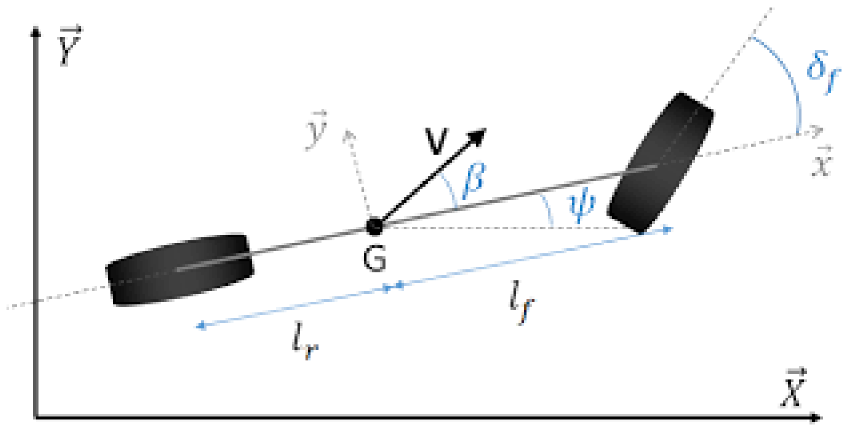


Figure 26: Kinematic Bicycle Model [27]

8 Computing

8.1 Hardware

The autonomous vehicle's computing platform consists of an Intel i7 8700 CPU, PSoC 5LP and an NVIDIA Jetson TX2. The i7 is the main processing unit of the vehicle. It will house the ROS nodes, one of which is mms_path_planner. This node will be the form of communication for the path-planning algorithm to the rest of the processes. The i7 is best for sequential processing, whilst the Jetson is best for parallel processing.

The path-planning algorithm consists of different aspects that differ in processing form. For example, the discovery lap algorithm is very sequential and cannot utilise the 256 cores provided by the Jetson. Contrarily, the racing line computation involves large matrix manipulations that can be optimised to run faster on the Jetson.

The Jetson will be used solely for camera processing due to large amounts of data in the images. Monash Motorsport currently has another Jetson that can be added to the car. Given its size and the associated complexities of adding another unit to the computing system, the team's preference is to leave it off. However, if it can be shown that it significantly increases the path-planning performance then it can be added. In this case the code will need to be written in CUDA.

Notwithstanding, the path planning will be trialled initially on the i7, which has an integrated UHD Graphics 630 unit which can be used to speed up components of the algorithm which can be run in parallel. A library called 'ArrayFire' can be used to utilise the integrated graphics processor.

8.2 Software

8.2.1 Platform

The operating system for the computing system is Ubuntu, on which ROS will be running the robotic processes of the vehicle. ROS will be run in C++, which was decided to be the foundational software language of the vehicle. C++ is useful due to its speed and ubiquity. Additionally, code written in C++ can be adapted to incorporate CUDA or ArrayFire for parallel processing speed boosts.

It was decided to use MATLAB to prototype algorithms due to its simple format and large library of inbuilt functions. A simulation environment was set up to visualise the car's behaviour on a track. For real implementation, the code is translated into C++.

8.2.2 Numerics Library

In order to carry out many of the operations that were afforded by MATLAB, a numerics library is needed in C++. All algorithms in path planning utilise matrices which are not a default class in C++. Additionally, the Levenberg-Marquardt algorithm has large matrix manipulations.

The track is a maximum of 500m, mapped out with cones at a maximum distance of 5m apart. Given that the 5m spacing will presumably only be applicable on straights, and that turns will require a higher density of cones, it is reasonable to estimate that there may be 150 cones on either side. This corresponds to $N = 150$ transverse lines. Therefore, the Jacobian matrix is 1050x300; the operations involved in computing $Jac^{\dagger}$ are significant.

TooN is a library which provides easy access to matrix decompositions. Because the Jacobian matrix is a Hermitian, positive-definite matrix, the library's Cholesky decomposition is used to compute the pseudoinverse of the Jacobian. The Cholesky decomposition is the most efficient matrix decomposition method when it can be applied (which it can here) [28].

9 Conclusion

The goals of this project were achieved satisfactorily. The discovery lap algorithm is able to continuously plan a path, and follow it, enabling the vehicle to drive and learn a racetrack without *any* prior knowledge. The resultant cones are used to compute an optimal racing line around the track on-line. The racing line optimisation is successful and is able to be easily tuned to factor in different considerations, such as limiting speed. The computation time relies heavily on the size of the track being considered, and further work needs to go into optimising the code to hasten the execution time. Overall, the path planning is at a promising stage of enabling Monash Motorsport in achieving its goal of completing all dynamic events at a formula student driverless competition.

9.1 Future Work

- Integration with other systems of the autonomous vehicle.
- Creation of an automatic track generator would greatly assist in the verification and tuning of the algorithms.
- Further testing to refine the shapes of the search areas in the discovery lap algorithm. This relates to the above point and is merely a matter of extensive testing.
- The matrix manipulations are a clear bottleneck in the computation of the racing line. It would be best to compare different numerics libraries, such as Eigen, and decide upon a library based on experimental factors.
- As above, a parallel processing approach is likely to assist with reducing the execution time of the algorithm. Future code developments should include the Jetson TX2 and the i7-integrated graphics card.
- Apart from making the processing time faster, work should be carried out to make the computation size smaller. This would include an adaptive interpolation function which selectively places fewer transverse lines on straights and a higher density of transverse lines on corners and areas with higher curvature in general. This would reduce redundant transverse lines taking up computation time.
- Investigate scaling of λ in $\boldsymbol{J}^{\dagger}$ in LMA to improve convergence time
- MPC needs to be improved. A basic implementation was used to allow the development of the discovery lap algorithm. This has a constant speed and is not acceptable for laps which have the optimal racing line.

9.2 Links

Code repository: https://bitbucket.org/mms-driverless/path-planning

FYP video: https://www.youtube.com/watch?v=uPFKnfqb3ZE

References

[1] (2017, October). [Online]. Available: https://www.formulastudent.de/fileadmin/user_upload/all/2018/rules/FS-Rules_2018_V1.1.pdf

[2] (2018, September). [Online]. Available: https://www.formulastudent.de/fsg/results/

[3] (2018). [Online]. Available: https://www.formulastudent.de/fileadmin/user_upload/all/2019/rules/FS-Rules_2019_V1.1.pdf

[4] S. Bagloee, M. Tavana, and M. e. a. Asadi, "J. mod. transport. (2016) 24: 284."

[5] M. Boulet, "The autonomous systems tidal wave," *Lincoln Labaratory Journal*, vol. 22, no. 2, 2017.

[6] Y. Ting, W. Lei, and H. Cha, "A path planning algorithm for industrial robots," *Computers and Industrial Engineering*, vol. 42, no. 2, 2002.

[7] S. Bortoff, "Path planning for uavs," in *American Control Conference*, Chicago, IL, USA, USA, 2000.

[8] C. Katrazas, M. Quddus, W.-H. Chen, and L. Deka, "Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions," *Transportation Research Part C: Emerging Technologies*, vol. 60, pp. 416–442, 2015.

[9] B. Paden, M. Cap, S. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," Massachusetts Institute of Technology, Laboratory for Information and Decision Systems, Cambridge MA, USA, Tech. Rep., 2016.

[10] C. E.F and C. Cordons, *Model Predictive Control*, second edition ed. London, UK: Springer-Verlag, 2007.

[11] S. Omidshafiei, "Optimal racing line control," Massachusetts Institute of Technology, Cambridge MA, USA, Tech. Rep., 2014.

[12] R. Oftadeh, R. Ghabcheloo, and J. Mattila, "Time optimal path following with bounded velocities and accelerations for mobile robots with independently steerable wheels," in *IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, China, 2014.

[13] S. Lazard, J. Reif, and H. Wang, "The complexity of the two dimensional curvature-constrained shortest-path problem," Duke University, Department of Computer Science, Durham, NC, USA, Tech. Rep., 2002.

[14] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[15] S. LaValle, “Rapidly-exploring random trees: A new tool for path planning,” Iowa State University, Department of Computer Science, Ames, IA, USA, Tech. Rep., 1998.

[16] S. Koenig and M. Likhachev, “D* lite,” *American Association for Artificial Intelligence*, 2002.

[17] D. Ferguson, M. Likhachev, and A. Stentz, “A guide to heuristic-based path planning,” Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, USA, Tech. Rep., 2005.

[18] Y. Xiong, “Racing line optimisation,” Massachusetts Institute of Technology, School of Engineering, Cambridge, MA, USA, Masters Thesis, 2010.

[19] M. Bevilacqua, A. Tsourdos, and A. Starr, “Particle swarm for path planning in a racing circuit simulation,” Cranfield University, Cranfield, Bedfordshire, UK, Tech. Rep., 2017.

[20] P. Falcone, F. Borrelli, J. Asgari, T. H.E, and D. Hrovat, “Predictive active steering control for autonomous vehicle systems,” *IEEE Transactions on Control Systems Technology*, vol. 15, no. 3, April 2007.

[21] (2018, January). [Online]. Available: https://www.formulastudent.de/fileadmin/user_upload/all/2018/rules/FSG2018_Competition_Handbook_V1.0.pdf

[22] H. Gavin, “The levenberg-marquardt method for nonlinear least squares curve-fitting problems,” Duke University, Department of Civil and Environmental Engineering, Durham, NC, USA, Tech. Rep., 2017.

[23] N. A. Group, “Exact first- and second-order greeks by algorithmic differentiation,” 2011.

[24] S. Roweis, “Levenberg-marquardt optimization,” New York University, Computer Science Department, Tech. Rep., 2009.

[25] X. Li, D. Du, and J. Cao, “Comparison of levenberg-marquardt method and path following interior point method for the solution of optimal power flow problem,” *International Journal of Emerging Electric Power Systems*, vol. 13, no. 3, July 2013.

[26] M. Isaksson Palmqvist, “Model predictive control for autonomous driving of a truck,” KTH Royal Institute of Technology, School of Electrical Engineering, Stockholm, Sweden, Degree Project, 2016.

[27] P. Polack, F. Altché, B. D’Andréa-Novel, and A. De La Fortelle, “The kinematic bicycle model: a consistent model for planning feasible trajectories for autonomous vehicles?” in *IEEE Intelligent Vehicles Symposium (IV)*, Los Angeles, CA, USA, June 2017.

[28] D. Lee, “Numerically efficient methods for solving least squares problems,” University of Chicago, Tech. Rep., 2012.