Path planning and control for an autonomous race car

Ignat Georgiev

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2019

Abstract

Recent technological advancements have enabled the rise of self-driving cars. In response to that, a new student competition was introduced, challenging students to develop an autonomous racing car. This requires aggressive path planning and control in real time, capable of pushing the vehicle to its limits. This project attempts to solve that problem by surveying the field and then deciding to utilise a novel algorithm under the name Model Predictive Path Integral (MPPI). It combines path planning and control into a single optimisation and can to handle non-linear system dynamics and complex cost criteria. A simulation, capable of accurately handling vehicle dynamics, is designed for development and evaluation. Thereafter a vehicle model of the algorithm is developed using a neural network to learn the dynamics of the car. To achieve this an automated learning framework is developed. The cost function of the algorithm is modified for the needs of the project and optimised for speed. Finally, a toolset for evaluation is created and the algorithm is evaluated in simulation, showing that it is capable of achieving better performance than human experts.

Acknowledgements

I would like to thank my supervisor Dr Michael Mistry for his continuous support, useful discussions and keeping me focused. I would also like to thank Grady Williams for his support and conceiving the algorithm this report is based on. Finally, I would like to thank Edinburgh University Formula Student for sparking my interest in robotics and for making me the man I am today.

Table of Contents

Glossary

ADS-DV Autonomous Driving System - Dedicated Vehicle, the car supplied by the IMechE for the FS-AI competition. 6–8, 23–25, 29–31, 35, 76

CAD Computer Aided Design software. 31

EUFS Edinburgh University Formula Student. 1–3, 5, 8, 9, 21–23, 25, 31, 32, 61, 62, 64

FS Formula Student International student competition. 28

FS-AI Formula Student - Artificial Intelligence. Branch of the Formula Student competition format which challenges students to develop a self-driving vehicle. 5, 6, 71

FSUK Formula Student United Kingdom, International student competition. 7

IMechE Institute of Mechanical Engineers, organisers of the UK Formula Student competition. 6, 76

MPC Model Predictive Controller, a family of common control algorithms. 11, 12

MPPI Model Predictive Path Integral controller. A stochastic forward-sampling algorithm that combines path planning and control into one. This is the topic of the report. 2, 3, 12, 13, 18, 22, 23, 31–33, 35, 40, 44, 50–53, 56–59, 61–64, 77

NN Artificial Neural Network. A machine learning model which has gained great popularity in recent years. 44–51

PID Proportional Integral Derivative, a classic controller for various applications. 7, 9–11

RNN Recurrent Neural Network. A machine learning model which extends standard neural networks by adding a memory which can be used to propagate data through time series. 63, 64

ROS The Robotics Operating System - the defacto standard for software system building in robotics. 8, 21, 22, 29, 32, 38, 40, 42, 58, 62, IX

SLAM Simultaneous Localisation and Mapping; a family of algorithms used in the field of robotics. 8, 23, 62

Mathematical Notation

Note that several symbols have different meanings in the steering section of the report, this is to be in keeping with the reference that the analysis is based on.

$\mathbf{x}_t$	Vehicle state at time t. This vector consists of both kinematic and dynamic state variables
$\mathbf{x}_k$	Vehicle kinematic state
$\mathbf{x}_d$	Vehicle dynamic state
$\mathbf{u}_t$	Vehicle control signals at time t. Consists of a target speed and steering angle
$\mathbf{v}_t$	Noisy vehicle control signals at time t
$\mathbf{F}(\cdot)$	State transition function
$\phi(\cdot)$	Terminal cost function
$\mathcal{L}$	Quadratic cost term
α	Slip angle
λ	Slip ratio
Ω	Rotational speed
$[*]_w$	the w denotes that the variable is in the world coordinate frame
$[*]_v$	the v denotes that the variable is in the vehicle coordinate frame
$\dot{x}$	velocity along the x axis of the coordinate frame
$\dot{y}$	velocity along the y axis of the coordinate frame
ϕ	roll. Rotation around the x axis
θ	pitch. Rotation around the y axis
ψ	yaw. Rotation around the z axis. Also known as heading
$\dot{\phi}$	roll velocity. Rotational speed around the x axis
$\dot{\theta}$	pitch velocity. Rotational speed around the y axis
$\dot{\psi}$	yaw velocity. Rotational speed around the z axis. Also known as heading rate
Δt	timestep

Specialised Terms

Ackermann-type vehicle - everyday cars seen on the road. They are defined as Ackermann if the geometric arrangement of linkages in the steering is designed to solve the problem of wheels on the inside and outside of a turn needing to trace out circles of different radii.

Wheelbase - the distance between the front and rear axles of a car.

Track - the distance between the centres of the left and right wheel of a car.

Polygon mesh - a collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics and solid modelling.

Centre of Mass (CoM) - In physics, the centre of mass of a distribution of mass in space is the unique point where the weighted relative position of the distributed mass sums to zero.

Moment of Inertia Matrix - a quantity that determines the torque needed for a desired angular acceleration about a rotational axis.

odometry - term in the robotics field used to define the position and velocity estimates of a mobile robot relative to its starting position.

rosbag - a convenient data recording tool used within the ROS ecosystem. Full documentation. Documentation is available at `http://wiki.ros.org/rosbag`.

Euler angles - the three standard angles (roll, pitch and yaw) which describe the orientation of a rigid body with respect to a fixed coordinate system.

Quaternion angles - alternative method of representing angles with complex numbers. This representation is usually preferred in practical applications in continuous space as it solves the issue of the Gimbal lock.

Gazebo - an open-source robotics simulator used for this project.

PyTorch - one of the most popular open-source machine learning libraries for Python.

CUDA - a parallel computing platform and programming model for general computing on Nvidia graphic cards.

Activation function - a function that is used to ”trigger” a neuron within a neural network.

Learning rule - an optimiser which alters the parameters of neural networks during learning.

Chapter 1

Introduction

Robots are one of the most fascinating machines that humans have ever developed. They are complex machines which involve the integration of many different bodies of knowledge - dynamics, kinematics, artificial intelligence, computer vision, control theory, optical engineering and many more. This makes robotics as interdisciplinary a field as there can be. Having mastered stationary robotic arms in the past decades, interests have shifted towards a more interesting, albeit more difficult, topic - mobile autonomous robots. These captivating machines have raised hopes of exploring extraterrestrial planets (Figure 1.1), replacing humans workers in dangerous environments and offering assistance to humans with laborious tasks [Siegwart et al., 2011].

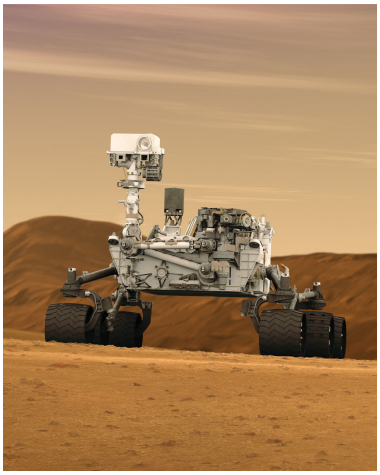


Figure 1.1: Curiosity, the NASA Mars Rover. Source: NASA/JPL-Caltech

Recent technological developments have given rise to self-driving cars which are now promising to alleviate traffic congestion, reduce road incidents and fatalities, and lower pollution [Thrun, 2010] [Fagnant and Kockelman, 2015]. These robocars operate similarly to any other autonomous mobile robot. They have to perceive the environment, create a map and localise within it, plan their next movements based on what they have seen and then execute a control sequence to achieve a given goal. As such the software behind an autonomous car can be classified into four different categories - perception, localisation, path planning and control. The focus of this report is on the latter two stages.

The goal of this project is to choose, integrate and evaluate a suitable algorithm for path planning and control for an autonomous racing car in an aggressive driving scenario. The motivation for this is to compete in the world's largest educational competition Formula Student which challenges students to develop a driverless racing car (Figure 1.2). As such the outcomes of this project are meant to further the efforts of the team from the University of Edinburgh - Edinburgh University Formula Student (EUFS).

However, due to a lack of a hardware platform, this project is based entirely in simulation. All of the work presented in this project is entirely the authors unless the opposite is explicitly stated.

This report initially surveys the fields of path planning and optimal control and designs a simulation for the development of such algorithms. Then it is chosen to utilise the recent Model Predictive Path Integral (MPPI) Controller which solves the problems of path planning and control simultaneously, while relying on a stochastic optimal control framework and path integrals approximated with an efficient importance sampling scheme [Williams et al., 2016]. The original implementation of this algorithm by researchers at the Georgia Institute of Technology is then adopted for the purposes of this project and integrated into the EUFS software stack.

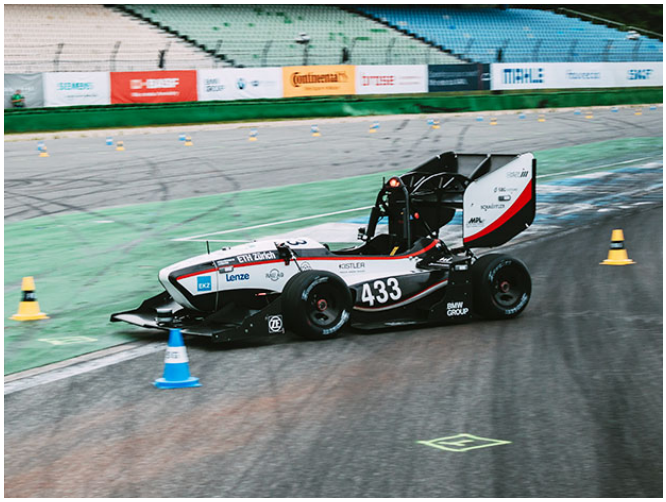


Figure 1.2: Fluela, a driverless racecar made by students from ETH Zurich. Source: Formula Student Germany

1.1 Objectives and Contributions

Achieving the primary goal of the report - integrating, improving and evaluating the MPPI algorithm into the EUFS autonomous racing software stack can be broken down into supplementary goals:

1. Review the literature of path planning, optimal control and system identification.
2. Design a Gazebo simulation for the development of path planning and control algorithms.
3. Create an efficient and accurate vehicle model to be used by MPPI.
4. Develop an optimisable cost function for MPPI suitable for the purposes of Formula Student.
5. Present an unbiased evaluation of the performance of the algorithm and potential areas for improvement.

Additionally, there was other work that was carried out in order to achieve the goals mentioned above. These are showcased below along with other contributions to the research community and the EUFS team.

Contributions

1. The simulation developed for this project was also extended to be used for other, wider-spreading purposes and has been open-sourced.
2. A plug-in for tyre dynamics simulation in Gazebo has been developed.
3. To aid in learning the dynamics of the vehicle, a framework has been developed for data collection, processing, filtering and training machine learning models.
4. The original MPPI algorithm is extended by allowing it to further learn and improve its dynamics model online.
5. Tools for evaluation, tuning and visualisation of the MPPI algorithm have been developed and fully integrated.

1.2 Report outline

The following chapters document the course of the project. Chapter 2 deals with refining the problem, presenting the challenge of Formula Student and the targeted hardware. Additionally, the chapter touched on system requirements, surveys existing literature on the topics of path planning and control, and finally presents the MPPI algorithm. Chapter 3 handles system integration while making the necessary abstractions and assumptions; the chapter also deals with defining the tools available for this project, describes the design of the simulation and presents the visualisation utilities used for debugging. Chapter 4 focuses on the state representation and the model of the vehicle. This chapter tackles the issues associated with data collection and processing, identifying an appropriate state representation and learning the dynamics via neural networks. Chapter 5 deals with the definition of the cost function for MPPI, details a method of integrating the surrounding world into the cost of the algorithm and evaluates its performance with varying parameters. Finally, Chapter 6 presents a summary of the work undertaken, the results obtained and possible further steps for improving the approach based on the results is provided.

Chapter 2

Background and Related Work

2.1 Motivation

The primary motivation for this project is for its results to be used by Edinburgh University Formula Student (EUFS) team for the world's largest educational competition - Formula Student. Traditionally, it challenges students to design, manufacture and build a racing car. Then students take these cars to competitions where they race them against other university teams, while also being judged on their design and business prowess. Over the past decade, the competition has significantly expanded. Today there are 13 different competitions on five different continents held every year and attract more than 40,000 students.

Recently the UK competition was extended with the new FS-AI format which challenges students to modify an existing car and make it completely autonomous, that is without any human intervention. The path planning and control algorithm designed in the course of this project will be used by the EUFS team for the competition and will be part of the software stack developed by the student team.

An additional motivation is that although existing control methodologies have proven to be adequate for many standard vehicle tasks such as lane keeping, turning and parking, there is an important frontier of control at the limits of vehicle performance that has not been adequately addressed. Autonomous racing and the mitigation of risk during collision avoidance are examples of aggressive driving domains in which success requires vehicles to operate near their performance limits [Funke et al., 2012]. However, this is a difficult problem to solve due to highly nonlinear operating regimes and dynamically changing situations which require real-time execution.

2.2 Problem Definition

2.2.1 Environment

The FS-AI competition consists of three events to challenge the autonomous cars. The most difficult one and of interest to this project is the *trackdrive event*. It represents a closed loop circuit of length between 200m and 500m, where track boundaries are marked with yellow and blue traffic cones. Cars are required to complete ten laps overall and are scored according to their average lap time. The autonomous cars have no prior knowledge of the layout of the track and thus must learn over the course of the event. The track consists:

- Straights: No longer than 80m.
- Constant Turns: up to 50m diameter.
- Hairpin Turns: Minimum of 9m outside diameter

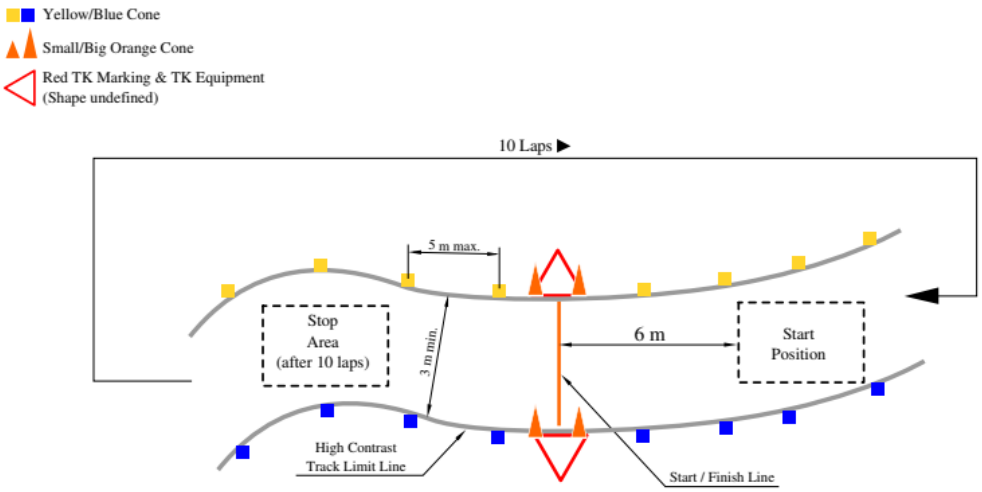


Figure 2.1: Trackdrive event layout. Source: Formula Student Rules 2019

The other events of the competition are described in Appendix A.

2.2.2 Vehicle

To aid the development of the competition in the UK, the organisers, the IMechE, have provided teams with an autonomous racing car named the ADS-DV as seen in Figure 2.2. It is an electric powered Ackermann-type vehicle[1] designed specifically for the event and will be the subject of the algorithm developed in this report.

[1] Ackermann-type vehicles are the ones seen on the road everyday. They are defined as Ackermann if the geometric arrangement of linkages in the steering is designed to solve the problem of wheels on the inside and outside of a turn needing to trace out circles of different radii.

Figure 2.2: The FSUK ADS-DV autonomous car. Source: IMechE

Vehicle Specifications

Key specifications of the vehicle are shown in Table 2.1. These will be later used in simulation development in Section 3.

The vehicle has a 4-wheel drive system with front and rear axles being powered by a dedicated Saietta 119R-68 motor independently.

Wheelbase	1530 mm
Track	1201 mm
Overall width	1430 mm
Overall length	2814.6 mm
Max steering angle	27.2°
Static caster	6°
Static camber	2°
Tyre size	13”
Differentials	Open
Peak motor torque	55.7 Nm
Peak motor power	17 kW
Continuous motor torque	27.2 Nm
Continuous motor power	8.8 kW
Maximum motor speed	4000 RPM
Total weight (est.)	120 kg

Table 2.1: ADS-DV Key Technical Specifications

Vehicle Commands

The vehicle accepts two types of control commands:

- Steering command with a value in the range [-1, 1]. -1 corresponds to a full left lock of -27.2°and a value of 1 corresponding to a full right lock of 27.2°.
- Target speed command in m/s with a value between [-40, 40]. Then a low-level PID controller attempts to reach the target speed as swiftly as possible.

The sensor suite and computing of the ADS-DV is detailed in Appendix B.

2.2.3 Overall System

This project is done in collaboration with EUFS and as such is just one part of the overall system. In this subsection, the overall architecture of the software stack is presented and how the work in this report contributes to the goal of autonomous racing.

The overall system is similar to a standard autonomous mobile robot and follows the *see-think-act cycle* presented by [Siegwart et al., 2011]. On an abstract level, it can be seen in Figure 2.3. If it is viewed as a pipeline system, then the *Path Planning* and *Control* stages are the absolute latest, and such certain assumptions must be made about the prior systems in the pipeline. These will be covered in Chapter 3.

The autonomous driving software has two modes. The first is called *Exploration mode* where the car has no data about the layout of the racecourse and there is some other *Planning and Control* algorithm guiding the car around the racetrack. During this mode, there is a full-fledged perception system detecting the traffic cones and a SLAM[2] algorithm building an accurate map of the environment. Once the car finished a single lap and the SLAM algorithm loop closes[3] and produces a complete map, then the car transitions into *Racing mode* where the algorithm developed in this paper takes over the tasks of *Planning and Control*.

The overall software stack is based on Ubuntu 16.04 and ROS Kinetic. ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms. Kinetic is the version of ROS released in 2016 for Ubuntu 16.04 [O'Kane, 2014].

2.2.4 Path Planning and Control Problems

Traditionally, autonomous control for cars involves a hierarchical approach that splits the problem into two. First, a path is planned through the environment, and then a controller is used to follow it. This methodology was first introduced in the DARPA Grand and Urban Challenges [Paden et al., 2016].

Path Planning
At the highest level, a vehicle's decision-making system must select a route from its current position to the requested destination using some representation of the environment. In the conditions of this report, some might state that path planning is absolute

[2]Simultaneous Mapping and Localisation (SLAM) algorithms are tasked with constructing or updating a map of an unknown environment while simultaneously keeping track of a robot's location within it.

[3]Loop closure refers to one of the crucial stages of a SLAM algorithm where it is able to detect previously seen scenes and correct the map of the environment generated thus far.

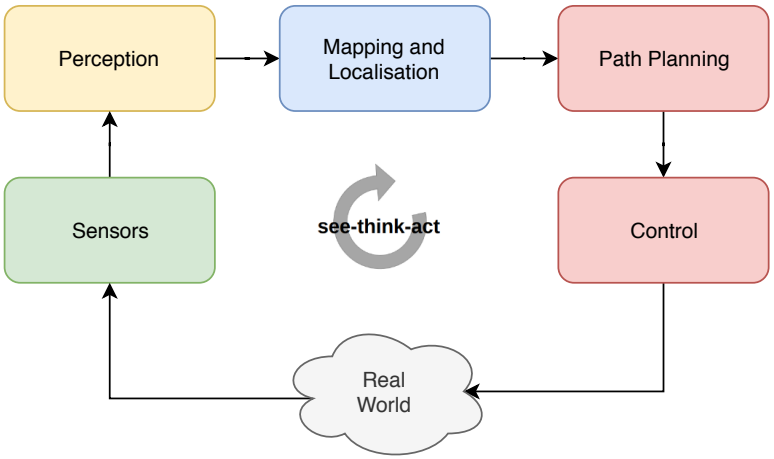


Figure 2.3: System architecture of EUFS.

as it is possible to easily generate a path that follows the midpoints of the racing circuit. Although possible, this will not be the most optimal trajectory and assumes that the car will be able to follow any arbitrary trajectory. A more elaborate approach might consider deriving the shortest or minimum-cost path along the race circuit using one of the classical shortest path algorithms such as the Dijkstra algorithm [Dijkstra, 1959] or A* [Nilsson, 1969]. However, these algorithms have one major limitation - they do not consider the capabilities of the robot using them. To address that issue one must enter the domain of model-based planners which consider the kinematics and dynamics of the vehicle during path planning.

Control

To execute the reference path or trajectory from the path planning system, a feedback controller is used to select appropriate actuators inputs to carry out the planned motion and correct tracking errors. Many would state that given that Ackermann-type vehicles have only 3 degrees of freedom in their pose and only 2 in control, then the control problem can be solved with simply PID controllers [Zhao et al., 2012]. However, cars operate in highly nonlinear regimes and are often subject to events which are not directly controlled by the actuators (e.g. tyre slipping). Furthermore, a control algorithm must generate an output which is feasible for the car both in absolute terms and in time-domain terms, as such similar to the previous paragraph, one must make use of a model of the vehicle. Another concern for this family of algorithms is to not only generate a command which attempts to reach the desired reference as soon as possible but to do so while taking into account how these actions would affect the future state of the car [Liniger et al., 2015].

2.3 Related Work

Given the popularity of the field, there is no lack of both simple and elaborate algorithms for path planning and control in autonomous driving. Most of these take shape and form of the hierarchical approach described in the previous section - first, a trajec-

tory is planned, and then a standard controller is used to follow it. The first successful implementations can be traced back to the DARPA Grand Challenge [Thrun et al., 2006] which are very conservative and plan trajectories and control outputs in the safest regions of the vehicle capabilities.

2.3.1 Path Planning

Most frequently used today in robotics overall are the *Graph Search Algorithms* which discretise the state space of the robot as a graph, where the vertices represent a finite collection of vehicle states and the edges represent transitions between vertices. The desired path is found by performing a search for a minimum-cost in such a graph. These methods do not suffer from local minima; however, they are limited to optimise only over a finite set of paths, namely those that can be constructed from atomic motion primitives of the graph. Some of the most widely-recognised ones are Dijkstra's algorithm [Aho and Hopcroft, 1974] and A* [Hart et al., 1968], which have been implemented successfully in autonomous driving scenarios. The three best cars in the DARPA Grand Challenge all used A* for path planning or some variation of it [Dolgov et al., 2008]. Most of these take into account the kinematics of the vehicle and plan safely possible paths and have even been used in an autonomous driving scenario [Valls et al., 2018]. In recent years there have been attempts to also encompass the dynamics into this path planning stage, however, that makes the computational complexity unfeasible [Paden et al., 2016].

Incremental Search algorithms are also popular in path planning. They attempt to address this issue by promising the same performance as Graph Search Algorithms at a fraction of the computational cost. These algorithms sample the state space and incrementally build a reachability tree that maintains a discrete set of reachable states and possible transitions. Once the graph is large enough so that at least one node is in the goal region, the desired path is obtained by tracing the edges that lead to that node from the starting position. Indeed there have been successfully deployed versions of Rapidly-exploring Random Trees (RRT) in autonomous driving scenarios which take into account both the kinematics and dynamics [Leonard et al., 2008].

2.3.2 Control

In terms of control, the goal then is to stabilise to the reference trajectory in the presence of modelling error and other forms of uncertainty. Many of these algorithms rely on feedback as a function of the nearest point to the reference path. The simplest approach is a standard Proportional Integral Derivative (PID) controller. In the case of steering, the car is given a reference trajectory it must follow and its position. Based on that it can calculate the error e between the two. That error is then fed into:

$$u(t) = K_p e(t) + K_i \int_0^t e(t')dt' + K_d \frac{de(t)}{dt}$$

where K_p, K_i and K_d are all non-negative, denote the coefficients for the proportional, integral, and derivative terms respectively (sometimes denoted P, I, and D).

PID controllers operate as seen in Figure 2.4, however, they suffer from short sightedness as at timestep t, they only considers the next timestep $t+1$. The issue of this is highlighted between e_2 and e_3. In the figure at timestep $t = 2$ the PID controller measures an error e_2 and then using the equation above calculates a new control command $u(t = 2)$ which is then continuously applied until timestep $t = 3$. However, at some point, the trajectory of the car crosses the reference trajectory and overshoots resulting in an increased error e_3. Given enough time, the car will reach the reference trajectory, but this solution is not optimal. A similar issue persists if PID is applied to control the speed or throttle, yet this is still widely used in practice [Alonso et al., 2013].

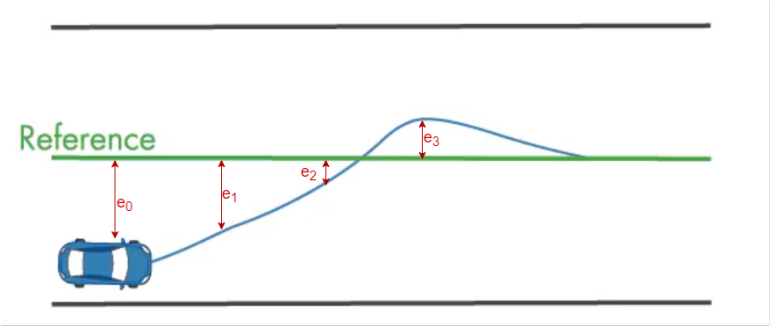


Figure 2.4: PID Controller for steering of an autonomous car.

An alternative to PID are the *Model Predictive Control (MPC)* family of algorithms. Although there is a considerable variety in this category most, in general, follow the architecture in Figure 2.5. The main advantage of MPCs is the fact that they allow the current timestep to be optimised while accounting for the future time horizon. This is achieved by forward-sampling (or simulating) into the future using a *model* of the vehicle. The possible control sequence and trajectory are then optimised over a finite time-horizon but only implementing the current timestep and then optimised again. This is repeated until convergence or a satisfactory solution is obtained. The key here is that the optimisation is subject to a variety of *costs* and *constraints* such as the rotation of the steering wheel.

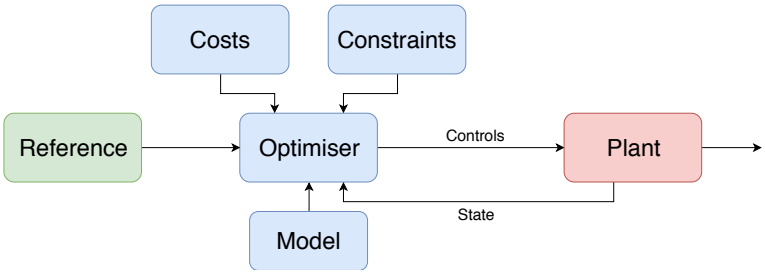


Figure 2.5: Standard MPC architecture

Figure 2.6 showcases how MPC compares to the previous PID example in Figure 2.4. For the sake of comparison, the first sampled trajectory (the one with the biggest error) is similar to the one generated by the PID controller. Then the optimiser measures the

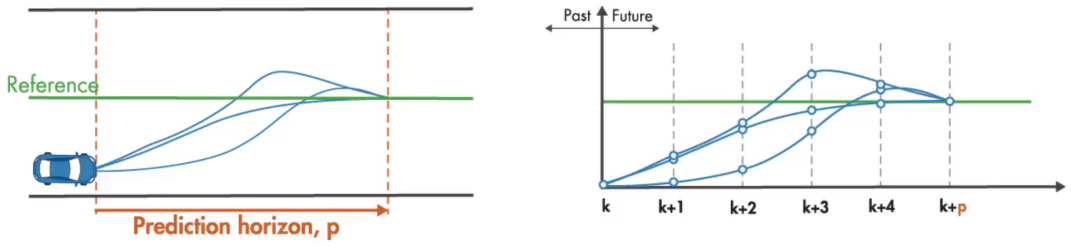


Figure 2.6: MPC for steering of an autonomous car.

predicted error at each timestep and re-optimises the control sequence. The process is repeated to gain the 3rd and most optimal trajectory.

The MPC coupled with one of the standard path planning methods discussed at the beginning of the subsection is the standard in the industry today and has many successful applications to autonomous vehicles [Thrun et al., 2006] [Paden et al., 2016] [Urmson et al., 2008] and in aggressive driving [Valls et al., 2018] [Funke et al., 2012]. Although the approaches above make control problems tractable, the decomposition into planning and control phases introduces inherent limitations. In particular, the path planner typically has coarse knowledge of the underlying system dynamics, usually only utilising kinematic constraints [Dolgov et al., 2008] [Pepy and Lambert, 2006]. This means that performing aggressive manoeuvres is problematic since the planned path may not be feasible. Attempts have been made to address the issue [Pepy et al., 2006] but then we end in a scenario where the planning and control algorithms are solving the same problem individually. Due to their close relation, a better option will be to combine the two processes into one. However, an issue then arises of computational capability and the real-time requirements.

There have been approaches trying to solve the planning and control problems simultaneously in limited lab environments [Keivan and Sibley, 2013] or by solving the optimal control inputs offline and then applying a controller to stabilise about an open loop trajectory [Velenis et al., 2007]. Another work for aggressive sliding manoeuvres to avoid collision where optimal paths are generated offline for a variety of states and a controller is synthesised using Gaussian Process regression [Tsiotras and Diaz, 2014]. However, in all of these methods, most of the computation is done offline over a finite number of states which would not be suitable for the case of aggressive driving required by this project.

2.4 Model Predictive Path Integral Control

The approach that addresses all concerns from the previos section is the *Model Predictive Path Integral (MPPI)* Controller developed by researches at Georgia Institute of Technology [Williams et al., 2017]. Unlike the approaches in the previous section, this algorithm focuses on the Reinforcement Learning (RL) task of computing the most optimal trajectory and control sequence simultaneously in an open-loop. This is done using a stochastic optimal control approach and exploiting a fundamental relationship

between the information theoretic notations of free energy and relative entropy resulting in an optimal solution which takes the form of iterative update law which can then be stochastically forward-sampled in parallel on a GPU. This approach is based on path integral control theory which has been put to practice in [Williams et al., 2016]. The appeal of the path integral framework is that it requires no derivatives of dynamics or costs. This in term allows for the faster optimisation needed in an autonomous racing scenario.

One of the major breakthrough of the MPPI algorithm is that it does not make the assumptions of control-affine dynamics. This allows for purely data-driving model learning. For example, one can leverage the recent advancements in machine learning to train a neural network to represent the model.

For the above reasons and the fact that the algorithm has been open-sourced, it was chosen to make the MPPI the main focus of this report. Therefore the rest of the chapter is devoted to the theory behind MPPI as derived by [Williams et al., 2017].

2.4.1 Problem Formulation

Consider the state and controls of the robot at time t to be denoted as $\mathbf{x}_t \in \mathbb{R}^n$ and $\mathbf{u}_t \in \mathbb{R}^m$. Then we can define the state transition as:

$$\mathbf{x_{t+1}} = \mathbf{F}(\mathbf{x_t}, \mathbf{v_t}) \tag{2.1}$$

where we assume that the actual control inputs are sampled from $\mathbf{v}_t \sim \mathcal{N}(\mathbf{u_t}, \Sigma)$ due to the noise induced in many robotic systems where the commanded input has to pass through other low-level controllers. Now we also define $\mathbf{u}(\cdot) : [t_0, T] \to \mathbb{R}^m$ as the function which maps time to control inputs. Additionally, we define the function $\tau : [t_0, T] \to \mathbb{R}^n$ of the system. Now it is possible to define the optimal control commands:

$$\mathbf{u}^*(\cdot) = \underset{U}{\operatorname{argmin}}\, \mathbb{E}_{\mathbb{Q}} \left[\phi(\mathbf{x}, T) + \int_{t_0}^{T} \mathcal{L}(\mathbf{x}_t, \mathbf{v}_t, t) dt \right] \tag{2.2}$$

where $\phi(\cdot)$ is the terminal cost and $\mathcal{L}$ is the quadratic cost term.

We now define $V = (\mathbf{v}_0, \mathbf{v}_1, ...\mathbf{v}_{T-1})$ as the sequence of inputs over some timesteps T which is itself a random variable defined as mapping $V : \Omega \to \Omega_V$. Now there are three distributions of interest:

- $\mathbb{P}$ is the probability distribution of input sequences in an uncontrolled system (i.e. $U \equiv 0$)
- $\mathbb{Q}$ is the probability distribution when the control input is an open-loop control sequence
- $\mathbb{Q}^*$ is the abstract optimal distribution in RL terms.

The probability density functions for these can then be denoted as $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{q}^*$, where the first two have analytical forms:

$$\mathbf{p}(V) = \prod_{t=0}^{T-1} Z^{-1} \exp\left(-\frac{1}{2}\mathbf{v}_t^T \sigma^{-1} \mathbf{v}_t \right) \tag{2.3}$$

$$\mathbf{q}(V) = \prod_{t=0}^{T-1} Z^{-1} \exp\left(-\frac{1}{2}(\mathbf{v}_t - \mathbf{u}_t)^T \sigma^{-1} (\mathbf{v}_t - \mathbf{u}_t) \right) \tag{2.4}$$

where $Z = \sqrt{(2\pi)^m |\sigma|}$.

Then given initial conditions x_0 and an input command sequence V, we can determine the system trajectory by recursively applying $\mathbf{F}$. For the most optimal trajectory, we need to define a cost to which we need to optimise. Below we define the state-dependent cost function for trajectories:

$$C(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_T) = \phi(\mathbf{x}_T) + \sum_{t=1}^{T-1} q(\mathbf{x}_t) \tag{2.5}$$

where $q(\cdot)$ is the instantaneous state cost. This is then mapped over input sequences.

The final state is to define the free-energy of the control system $(\mathbf{F}, S, \lambda)$ with coefficient λ as:

$$\mathcal{F}(V) = -\lambda \log\left(\mathbb{E}_{\mathbb{Q}}\left[\frac{\mathbf{p}(V)}{\mathbf{q}(V)} \exp -\frac{1}{\lambda} S(V) \right] \right) \tag{2.6}$$

Through the definitions of $\mathbf{p}(V)$ and $\mathbf{q}(V)$, we can then rewrite and minimise it to:

$$= \mathbb{E}_{\mathbb{Q}}\left[S(V) + \frac{\lambda}{2} \sum_{t=0}^{T-1} \mathbf{u}_t^T \Sigma^{-1} \mathbf{u}_t \right] \tag{2.7}$$

For the full derivation and minimisation of the above term, please refer to [Williams et al., 2017]. This leaves us with the cost of an optimal control problem bounded from below by the free energy of the system. Now we define the abstract optimal solution $\mathbb{Q}^*$ through its density function:

$$\mathbf{q}^*(V) = \frac{1}{\eta} \exp\left(-\frac{1}{\lambda} S(V) \mathbf{p}(V) \right) \tag{2.8}$$

where η is the normalisation constant. Now we have the optimal distribution and can compute the controls to push the controlled distribution as close as possible to the

optimal one. Formally this is defined with the KL divergence and is solved in the next subsection:

$$U^* = \underset{U}{\operatorname{argmin}} \mathbb{D}_{KL}(\mathbb{Q}^* \| \mathbb{Q}) \tag{2.9}$$

2.4.2 Minimisation and Sampling

In this subsection, we will refer to the minimisation of Equation 2.8 and its sampling as presented by [Williams et al., 2017].

Using the definition of KL divergence [Kullback and Leibler, 1951]:

$$\begin{aligned} D_{KL}(\mathbb{Q}^* \| \mathbb{Q}) &= \int_{\Omega_V} \mathbf{q}^*(V) log\left(\frac{\mathbf{q}^*(V)}{\mathbf{q}(V)}\right) dV \\ &= \int_{\Omega_V} \mathbf{q}^*(V) log\left(\frac{\mathbf{q}^*(V)}{\mathbf{p}(V)} \frac{\mathbf{p}(V)}{\mathbf{q}(V)}\right) dV \\ &= \int_{\Omega_V} \mathbf{q}^*(V) log\left(\frac{\mathbf{q}^*(V)}{\mathbf{q}(V)}\right) - \mathbf{q}^*(V) log\left(\frac{\mathbf{q}(V)}{\mathbf{p}(V)}\right) dV \end{aligned} \tag{2.10}$$

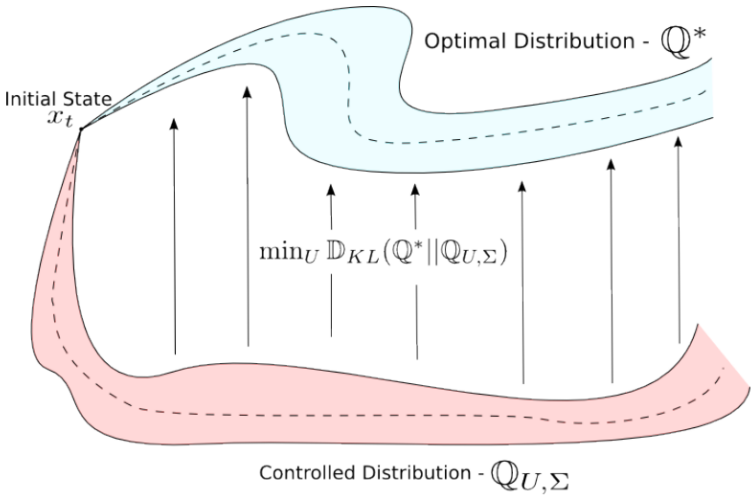


Figure 2.7: Visualisation of the information theoretic control objective of "pushing" the controlled distribution close to the optimal one. [Williams et al., 2017].

Neither the optimal distribution $\mathbf{q}^*(V)$ nor the uncontrolled dynamics distribution $\mathbf{p}(V)$ depend on the control inputs U we apply. Therefore we can now express U^*, flip the sign and change the minimisation to maximisation:

$$\begin{aligned} U^* &= \underset{U}{\operatorname{argmin}} \int_{\Omega_V} -\mathbf{q}^*(V) log\left(\frac{\mathbf{q}(V)}{\mathbf{p}(V)}\right) dV \\ &= \underset{U}{\operatorname{argmax}} \int_{\Omega_V} \mathbf{q}^*(V) log\left(\frac{\mathbf{q}(V)}{\mathbf{p}(V)}\right) dV \end{aligned} \tag{2.11}$$

From equations 2.3 and 2.4 we can show that:

$$\frac{\mathbf{q}(V)}{\mathbf{p}(V)} = \exp\left(\sum_{t=0}^{T-1} -\frac{1}{2}\mathbf{u}_t^T\Sigma^{-1}\mathbf{u}_t + \mathbf{u}_t^T\Sigma^{-1}\mathbf{v}_t\right) \tag{2.12}$$

Inserting it into Equation 2.11 gives us:

$$U^* = \underset{U}{\operatorname{argmax}} \int_{\Omega_V} \mathbf{q}^*(V)\left(\sum_{t=0}^{T-1} -\frac{1}{2}\mathbf{u}_t^T\Sigma^{-1}\mathbf{u}_t + \mathbf{u}_t^T\Sigma^{-1}\mathbf{v}_t\right)dV \tag{2.13}$$

After integrating our the probability in the first term, this expands to:

$$U^* = \underset{U}{\operatorname{argmax}} \left(\sum_{t=0}^{T-1} -\frac{1}{2}\mathbf{u}_t^T\Sigma^{-1}\mathbf{u}_t + \mathbf{u}_t^T\int_{\Omega_V} \mathbf{q}^*(V)\Sigma^{-1}\mathbf{v}_t dV\right) \tag{2.14}$$

This is concave with respect to each $\mathbf{u}_t$, therefore we can find the maximum with respect to each $\mathbf{u}_t$ by taking the gradient and setting it to zero, yielding:

$$\mathbf{u}_t^* = \int \mathbf{q}^*(V)\mathbf{v}_t dV \tag{2.15}$$

This effectively gives us our optimal control input $\mathbf{u}_t^*$. This can be obtained only be sampling from the optimal distribution $\mathbb{Q}^*$. However, as we introduced in 2.4.1 $\mathbb{Q}^*$ is an arbitrary distribution, and we cannot directly sample from it. Instead, we can do it via importance sampling as introduced in [Luo et al., 2016].

Importance sampling allows us to sample from an unknown distribution by drawing samples from a *proposal distribution* and re-weight the integral using the *importance weights* in such a way that the correct distribution is targeted. For example, if we have a well-defined integral:

$$J = \int_X h(x)dx$$

it can be expressed as an expectation for a wide range of probability distributions:

$$J = \mathbb{E}_f[H(X)] = \int_X H(x)f(x)dx$$

where f denotes the density of a probability distribution and H is:

$$H(x) = \frac{h(x)}{f(x)}$$

now J becomes

$$J = \int_X \frac{h(x)}{f(x)} f(x)dx$$

if we know the probability density function $f(x)$ and under the restriction $f(x) > 0$ when $h(x) \not\equiv 0$ we can sample approximate the unknown distribution $h(x)$ [Luo et al., 2016].

Using this technique, we can now rewrite Equation 2.15 as:

$$\mathbf{u}_t^* = \int \mathbf{q} \frac{\mathbf{q}^*(V)}{\mathbf{p}(V)} \frac{\mathbf{p}(V)}{\mathbf{q}(V)} \mathbf{v}_t dV \tag{2.16}$$

Now we can rewrite it as an expectation with respect to $\mathbb{Q}$:

$$\mathbf{u}_t^* = \mathbb{E}_{\mathbb{Q}}[w(V)\mathbf{v}_t] \tag{2.17}$$

where the importance sampling weight coupled with Equation 2.8 becomes:

$$\begin{aligned} w(V) &= \frac{\mathbf{q}^*(V)}{\mathbf{p}(V)} \exp\left(\sum_{t=0}^{T-1} -\mathbf{v}_t^T \Sigma^{-1} \mathbf{u}_t + \frac{1}{2} \mathbf{u}_t^T \Sigma^{-1} \mathbf{u}_t \right) \\ &= \frac{1}{\eta} \exp\left(-\frac{1}{\lambda} S(V) + \sum_{t=0}^{T-1} -\mathbf{v}_t^T \Sigma^{-1} \mathbf{u}_t + \frac{1}{2} \mathbf{u}_t^T \Sigma^{-1} \mathbf{u}_t \right) \end{aligned} \tag{2.18}$$

To obtain a full solution we need to swap the variables $\mathbf{u}_t + \varepsilon_t = \mathbf{v}_t$ and denote the noise sequence as $\mathcal{E} = (\varepsilon_0, \varepsilon_1, ...\varepsilon_{T-1}$ we can then define $w(\mathcal{E})$ as:

$$w(\mathcal{E}) = \frac{1}{\eta} \exp\left(-\frac{1}{\lambda} \left(S(U+\mathcal{E}) + \lambda \sum_{t=0}^{T-1} \frac{1}{2} \mathbf{u}_t^T \Sigma^{-1} (\mathbf{u}_t + 2\varepsilon_t) \right) \right) \tag{2.19}$$

where η can be approximated using the Monte-Carlo estimate:

$$\eta = \sum_{n=1}^{N} \exp\left(-\frac{1}{\lambda} \left(S(U+\mathcal{E}) + \lambda \sum_{t=0}^{T-1} \frac{1}{2} \mathbf{u}_t^T \Sigma^{-1} (\mathbf{u}_t + 2\varepsilon_t) \right) \right) \tag{2.20}$$

with each of the N samples drawn from the system with U as the control input. We then have the iterative update law:

$$\mathbf{v}_t^{i+1} = \mathbf{v}_t^i + \sum_{n=1}^{N} w(\mathcal{E}_n) \tag{2.21}$$

Note that this iterative procedure exists only to improve the Monte-Carlo estimate of Equation 2.16 by using a more accurate importance sampler. This scheme would not be needed if we were able to sample $\mathbb{Q}^*$ directly.

2.4.3 MPPI Algorithm

In the setting of this algorithm, optimisation and execution take place simultaneously: a control sequence $U = (\mathbf{u}_t, \mathbf{u}_{t+1}, \mathbf{u}_{t+2}...)$ is computed, then the first element $\mathbf{u}_t$ is executed. This procedure is repeated using the un-executed portion of the previous control sequence as the importance sampling trajectory for the next iteration while utilising the transition model $\mathbf{F}(\cdot)$ from Equation 2.1.

If given access to a perfect model, then doing forward-sample would be enough to find the most optimal trajectory and control sequence. However, the real world is complex, and even if a perfect vehicle model exists, it would be difficult to predict motion in real-time. Instead, here it is assumed that the model will inevitably be error-prone. This coupled with the inherently noisy control inputs as defined in Section 2.4.1 will make a single forward-sampled control sequence noisy and not consistently optimal. To address that, it is possible to Monte-Carlo estimate the optimal control sequence by forward-sampling multiple noisy control sequences and then cost-weighted averaging all of the sampled trajectories. A key to this is to have an underlying costmap which gives the cost of being on a particular location on the race course. The sampling process is visualised in Figure 2.8. It can be seen that if only one trajectory is sampled, it is noisy and inefficient. Even if four trajectories are sampled, they are still diverse and if averaged, the result would still be unstable. Cost-wise averaging ten sampled trajectories might result in a smooth optimal trajectory as seen in the figure. In that setting, trajectories with high costs like the first one will have less weight as the cost of following it is high. It is worth noting that Figure 2.8 shows a simplified example. In practice, it is needed to sample significantly more than 10 trajectories in order to consistently find the optimal one.

Now it is possible to define the full MPPI algorithm as developed in [Williams et al., 2017]:

Algorithm 1: MPPI Algorithm

Given: $\mathbf{F}$: Transition Model
K: Number of samples
T: Number of timesteps
$(\mathbf{u}_0, \mathbf{u}_1 \ldots \mathbf{u}_{T-1})$: Initial control sequence
Σ, ϕ, q, λ: Control hyper-parameters
while *task not completed* **do**
  $\mathbf{x}_0 \leftarrow$ GetStateEstimate()
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $\mathbf{x} \leftarrow \mathbf{x}_0$
    Sample $\mathcal{E}_k = (\varepsilon_0^k, \varepsilon_1^k, \ldots \varepsilon_{T-1}^k)$
    for $t \leftarrow 1$ **to** T **do**
      $\mathbf{x}_t \leftarrow \mathbf{F}(\mathbf{x}_{t-1}, \mathbf{u}_{t-1} + \varepsilon_{t-1}^k)$
      $S(\mathcal{E}_k) \mathrel{+}=$
       $\mathbf{q}(\mathbf{x}_t) + \lambda \mathbf{u}_{t-1}^T \Sigma^{-1} \varepsilon_{t-1}^k$
    end
    $S(\mathcal{E}_k) \mathrel{+}= \phi(\mathbf{x}_T)$
  end
  $\beta \leftarrow \min_k [S(\mathcal{E}^k)]$
  $\eta \leftarrow \sum_{k=0}^{K-1} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $w(\mathcal{E}^k) \leftarrow$
     $\frac{1}{\eta} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$
  end
  for $t \leftarrow 0$ **to** $T-1$ **do**
    $\mathbf{u}_t \mathrel{+}= \sum_{k=1}^{K} w(\mathcal{E}^k) \varepsilon_t^k$
  end
  SendToActuators($\mathbf{u}_0$)
  for $t \leftarrow 1$ **to** $T-1$ **do**
    $\mathbf{u}_{t-1} \leftarrow \mathbf{u}_t$
  end
  $\mathbf{u}_{T-1} \leftarrow$ Initialise($\mathbf{u}_{T-1}$)
end

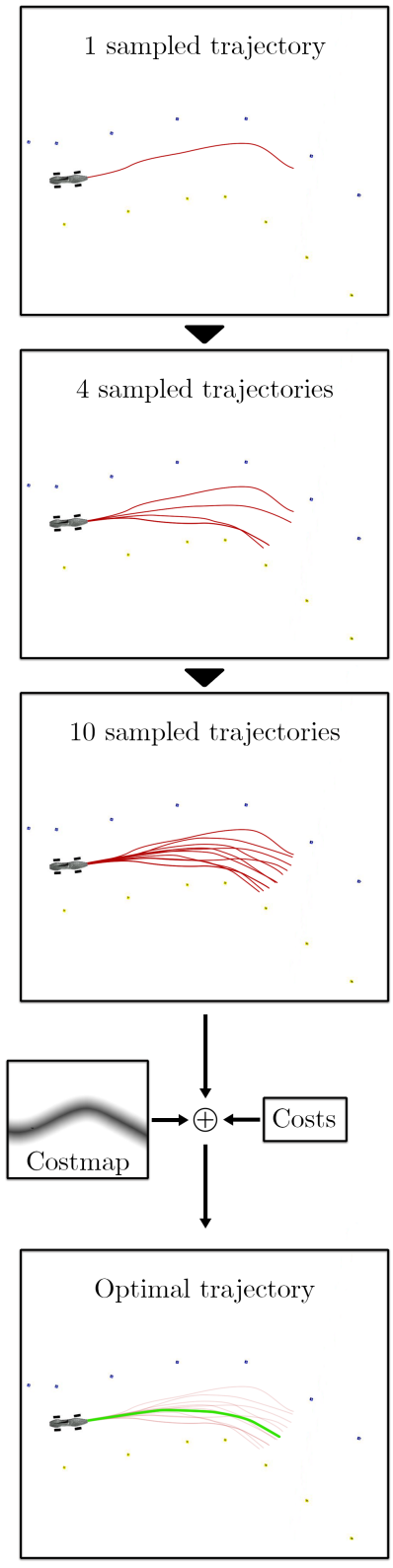


Figure 2.8: MPPI Sampling and cost-wise averaging

Chapter 3

Simulation and Integration

3.1 Setup and tools

As the end-goal is to integrate this project into the EUFS software stack as presented in Section 2.2.3, this requires results of this report to be implemented for ROS. This does impose limitations of available tools and programming languages, however, ROS offers a plethora of other benefits including standard communication methods, easy visualisation and parameter tuning [O'Kane, 2014]. For this reason and the fact that the MPPI source code provided by GeorgiaTech is also integrated with ROS [Williams, 2019], it was chosen to use ROS Kinetic and a combination of C/C++, Python and CUDA as the main development tools for this project.

Simulation choice
The next important step for the development of this project is to choose a simulation environment which meets the following requirements:

- Accurate simulation of the vehicle kinematics and dynamics.
- Efficient real-time simulation.
- Ease of use and integration with the project.
- Reusable by the EUFS team for other development.

With the rise of interest towards autonomous cars, in recent years different simulators have been developed with this particular use in mind. The most popular open-source ones are CARLA [Dosovitskiy et al., 2017] and AirSim [Shah et al., 2018]. However, these and similar simulators focus on providing high-fidelity visuals which require extensive processing power but yield no benefits for the aim of this project.

Instead, the author opted for a more standard robot simulator which usually offer better physics simulation. The most popular simulators are:

- **V-REP** is one of the most complex and versatile robotics simulators available. It offers multiple physics engines, a large model library and it's highlight feature

- mesh manipulation[1]. However, V-REP is also very resource hungry and its unique features are not of interest for this project [Rohmer et al., 2013].

- **ARGoS** sacrifices environment and physics complexity for efficient swarm robotics simulation. It is usually favoured for research in the areas of multi-agent environments, however, it offers very limited customisation [Pinciroli et al., 2011]. Similarly to V-REP, the unique features this simulator offers are of no interest to this project, which values accurate physics simulation above all.

- **Gazebo** occupies a space between V-REP and ARGoS, while also being the simplest and most optimised simulator for single-robot environments. This combined with its versatile robot modelling and out-of-the-box integration with ROS have made it a favourite within the robotics community [Koenig and Howard, 2004].

With the above in mind, Gazebo is the obvious choice for this project and the remainder of this chapter will be focusing on building an accurate simulator with Gazebo.

3.2 MPPI Integration

The MPPI source code is already developed to work with ROS and follows many of the standard conventions. Thus, integrating it into the EUFS software stack is simplified. An overall layout of the system is seen in Figure 3.1, where the red boxes represent the ready-available MPPI algorithm in the form of a ROS node, which will not be modified extensively. Instead the goal of this project is to provide the surrounding systems, with the blue boxes being the inputs and the yellow boxes being the outputs. However, since this project is abstracted from the states prior to Planning and Control (Section 2.2.3), then it is necessary to make certain assumptions which will be highlighted below. The three main inputs necessary to run the algorithm are:

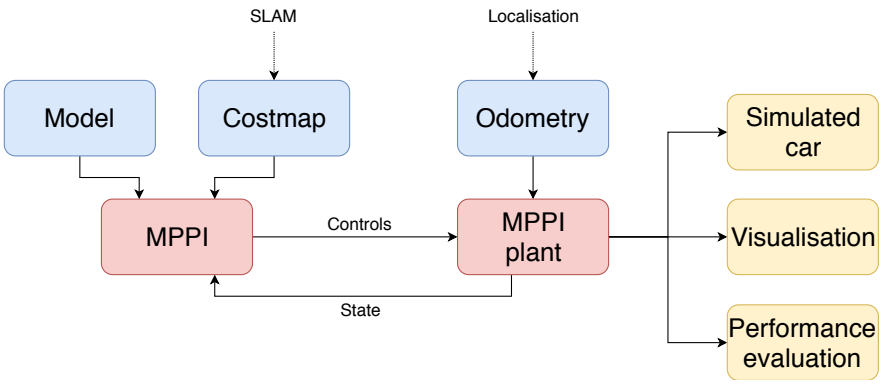


Figure 3.1: MPPI node interactions. Each box represents an independent ROS node.

1. Model - as any MPC-type algorithm, a vehicle model is necessary for the forward-sampling. This is explored in Chapter 4.

[1]Mesh manipulation refers to the process of altering a collection of vertices, edges and faces that define the shape of a polyhedral object in 3D computer graphics

2. Costmap - in Section 2.4 it was seen that MPPI cost-weight averages multiple sampled trajectories. The most important of these costs is the map which gives spacial information to the algorithm. Such a map is usually provided by a SLAM algorithm, however, that is not available at the time of writing this. Thus, it was decided abstract that by 1) assuming that the SLAM will output 2D bird's eye cone locations. This "map data" is simulated in Section 3.4. The costmap generation is then tackled in Chapter 5.

3. Odometry - the car (and in extension the MPPI algorithm) must always be aware where it is in the world in order to plan a trajectory correctly. Estimating odometry is a difficult task and dependent on the hardware available. However, 2) odometry estimation is assumed to be handled elsewhere in the EUFS software stack, thus for the purposes of this project, the odometry is simulated which will be covered towards the end of this chapter.

After the algorithm has received all of the necessary inputs, it can output its control sequence and other auxiliary data which is handled by the nodes in yellow (Figure 3.1):

1. Simulated car - since the project is entirely based in simulation, one must be developed in order to evaluate the algorithm and its performance. This simulation development is handled for the remained of this chapter.

2. Visualisation - essential in debugging such a complex algorithm. Helpful visualisations are handled in Section 3.5.

3. Tools for *performance evaluation* of MPPI are also developed and coupled within a single node. This is elaborated on throughout Chapter 5.

3.3 Simulated vehicle model

Gazebo does not offer standard Ackermann robot models and constructing a new model altogether can be time-consuming and distracting from the goal of this project. Instead, it was chosen to base the vehicle model on existing implementations and then tailor them for the purposes of this project. The author opted for the `rb_car` simulation model developed by Robotnik Automation [Automation, 2018], which is created as a simulation for a self-driving golf cart.

A mesh file of the ADS-DV was provided by the competition organisers in STL format featuring 422,928 faces as seen in Figure 3.2(a). As the mesh represents a rigid body, it must be broken down into two separate components: chassis with bodywork and suspension with wheels.

Throughout the rest of this chapter, the model computation efficiency will be evaluated with the real-time factor metric of Gazebo. This metric is within the range [0, 1] with 1 meaning that the simulation is running at real-time with respect to the computer wall time.

(a) The raw ADS-DV vehicle mesh.

(b) The cleaned up and final mesh of the chassis and bodywork.

Figure 3.2: The ADS-DV vehicle mesh.

3.3.1 Chassis

First, it is needed to take the original mesh as seen in Figure 3.2(a) and strip it down to include only the chassis and bodywork of the vehicle. This was done in MeshLab, which is an open-source mesh manipulation tool. The resulting can be seen in Figure 3.2(b) containing 144,726 faces.

To insert this mesh into Gazebo, three important parameters are needed:

- Mass - overall vehicle mass is estimated to be 120 kg as seen in Table 2.1. The vehicle is equipped ULTRALITE Mini Wheels 13x6J weighing 4.7 kg and Hoosier tyres #43163 weighting 5kg. Based on this it is possible to approximate the mass the mesh by subtracting the mass of the four wheels from the mass of the vehicle:

$$120 - 4 \times (4.7 + 5) = 81.2kg$$

- Centre of mass - difficult to approximate without empirical results from the real vehicle. However, it is known that the batteries of the car are situated at the exact centre of the chassis in all x,y,z axes, it is possible to assume that the centre of mass of the batteries is at the origin of the chassis mesh. Furthermore, in electric Formula Student vehicles, batteries usually account for more than half of the total weight, therefore, it will be a safe 3) assumption that the centre of mass of the vehicle will be the origin of the chassis mesh.

- Moment of Inertia Matrix - for the continuous body the inertia matrix is given by:

$$\mathbf{I} = \int\int\int_{Q} \rho(\mathbf{r})((\mathbf{r}\cdot\mathbf{r})\mathbf{E} - \mathbf{r}\otimes\mathbf{r})dV$$

  Where $\mathbf{r}$ defines the coordinates of a point in the body, $\rho(\mathbf{r})$ defines the mass density at that point, $\mathbf{r}$ is the identity tensor and the integral is taken over the volume of the body.

  The available mesh does not have mass information for each vertex, therefore $\rho(\mathbf{r})$ is unknown. The author makes the 4) assumption that the mass density is

normally distributed along the x,y,z axes with mean $\mu = [0,0,0]$ and standard deviation $\sigma = [\frac{2L}{3}, \frac{2W}{3}, \frac{2H}{3}]$, where L is the vehicle height, W is the vehicle width and H is the vehicle height. This is a valid assumption due to the symmetric properties of the car.

With this assumption is possible to compute the inertia matrix, which the author calculates with MeshLab. It was found out to be:

$$\mathbf{I} = \begin{bmatrix} 0.2429 & 0.0969 & 0.0788 \\ 0.0969 & 0.2422 & 0.0787 \\ 0.0788 & 0.0787 & 0.3428 \end{bmatrix}$$

When this mesh as swapped in the `rb_car model` as both a visual and a collision mesh, its fidelity and detail proved to be too high for Gazebo and on each interaction between the environment and the model mesh, a real-time factor of ¡0.1 was reported.

Based on this bad performance, the author concluded that it is needed to reduce the fidelity of the chassis mesh as much as possible without deforming the overall shape of the car. This would not cause any issues with the simulation of the kinematics of the vehicle as long as the mass, centre of mass and inertia matrix from the original mesh is maintained.

The mesh complexity was then reduced by removing all of the internals of the mesh (batteries, electronics, motors, computers etc..) as they would not be the first parts of the mesh to collide with external objects. Then the fidelity of the mesh was reduced using Quadric Edge Collapse Decimation [Hoppe, 1999] to only 48,238 faces (with MeshLab). With this new minimised mesh, the Gazebo simulator maintained a real-time factor close to 1 even when colliding with multiple objects.

3.3.2 Suspension

The suspension of the vehicle is crucial to the simulation of the kinematics and dynamics of the vehicle. Ideally this should be validated on the real vehicle by collecting telemetry data, however that is not available at the time of the writing, therefore, the author can only approximate the suspension using empirical static tests with the suspension of SISU 3, the 2018 internal combustion car developed by EUFS and the vehicle dynamics analysis thesis by fellow EUFS team member Martin O'Connor [O'Connor, 2018].

Although the suspension system of a Formula Student vehicle is complex and difficult to analyse, it can be simplified to the 7 degree-of-freedom model as seen in Figure 3.3.

This model can then be split into 8 parts of interest to this project - the 4 suspension damping models which are shown as ZBLU, ZFLU, ZBRU and ZFRU and the 4 contact force and friction models between the tyres and the ground. Due to the symmetric properties of the ADS-DV vehicle as discussed in Section 3.3.1, it can be 5) assumed that the 4 suspension damping models are identical and the 4 tyre models are identical.

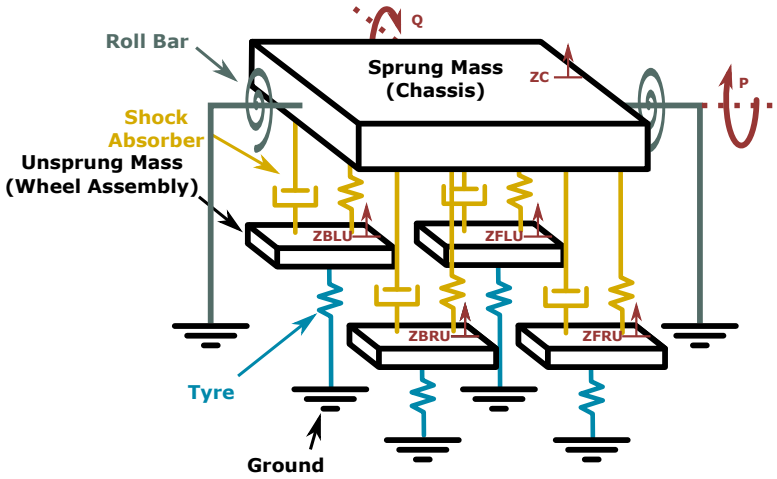


Figure 3.3: 7 D.O.F. dynamics model of a car [O'Connor, 2018]

The internal forcing on the springs can be broken down into the three standard equations which are produced as described by [O'Connor, 2018]:

$$\begin{aligned} Q_{damping} &= C \cdot (\dot{r}_k - \dot{r}_{k-1}) \\ Q_{gravity} &= -g \cdot I_k \\ Q_{forcing} &= Z_{CoM} \end{aligned} \tag{3.1}$$

$Q_{gravity}$ and $Q_{forcing}$ will be handled by the physics engine in Gazebo by using the `hardware_interface/EffortJointInterface` joint model[2] to simulate shocks which comes as standard with Gazebo. Therefore, we only need to calculate $Q_{damping}$. This is done using the MATLAB script in Appendix C of [O'Connor, 2018], and the resulting damping coefficient was $9.877N/m$, an additional frictional coefficient of $0.7N/m$ was also introduced as it stabilised the simulation model in empirical testing.

The next step is to create models of the tyres. Ideally, the tyres should be simulated as flexible body objects that can deform as the vehicle is subjected to different forces, however, Gazebo does not natively offer such functionality. It is possible to use the rigid body mesh of the tyres of the vehicle as shown in Figure 3.2(a), however, the complexity of the model will result in a slow simulation with minimal benefits. Instead, the author opted to use perfect cylindrical models for the tyres. Similarly to the procedure in Section 3.3.1, it is needed to find the mass, centre of mass and inertia matrix. The mass of each wheel is $9.7kg$, the centre of mass is fixed at the origin of the wheel due to its symmetric properties and the inertia matrix is:

$$\mathbf{I} = \begin{bmatrix} 0.2476 & 0 & 0 \\ 0 & 0.2476 & 0 \\ 0 & 0 & 0.4413 \end{bmatrix}$$

[2]Gazebo joints are an API for robot motor and actuator simulation.

3.3.3 Tire friction modelling

The next step in the process is to define correct friction models between the tyres and the ground. Car tyres typically consist of an outer shell of vulcanised rubber covering multiple layers of wrapped fabric reinforced with metal wire. They are incredibly complex to simulate and no off-the-shelf Gazebo plugin exists for this purpose, but simulating tyre dynamics is crucial to the project. For that reason, the author proceeded to create a custom tyre fiction model based on the Pacejka "Magic Formula" and brush model (known as ThreadSim) [Pacejka, 2005]. Below, the author will provide the background of tyre modelling which was originally authored in [O'Connor, 2018].

Contact forces from the tyre to the plane are transmitted through a "contact patch", however, due to the elastic behaviour of tyres, the contact patch varies greatly depending on the tyre pressure, load, lateral forces, longitudinal forces amongst other factors. A simplified analytical model of this called the "brush model" has been developed and is used to explain important parameters:

- slip angle α - the angle at which the tyre is orientated. This is the main factor determining the maximum lateral force a tyre can exert and is described by

$$\alpha = \arctan\left(\frac{\dot{x}_v}{\dot{y}_v}\right)$$

  where $\dot{x}_v$ is the longitudinal velocity and $\dot{x}_v$ is the lateral velocity; both in the vehicle frame.

- slip ratio λ - as the wheel accelerates there will be a tangential force on the brushes, causing an extension which increases the circumference of the tyre and forcing the wheel to turn slower for the same rotational velocity. The slip ratio is described by:

$$\lambda = \frac{r_e \cdot \omega - \dot{y}_v}{\dot{x}_v} = \left(\frac{r_e \cdot \omega}{\dot{x}_v}\right) - 1$$

  where r_e is the tyre radius and ω is the tyre rotational speed.

Multiple attempts have been made to model tyre behaviour, amongst which the most popular one is the Pacejka "Magic Formula" [Pacejka, 2005] which is derived completely from empirical results and fits the slip angle to lateral force and slip ratio to longitudinal force as described by:

$$y = D \cdot \sin\left(C \cdot \arctan\left(B \cdot x - E\left(\arctan B \cdot x\right)\right)\right).$$

A typical plot of this can be seen in Figure 3.4. 6) Note that this formula holds true only for dry friction which is assumed in simulation and when running the autonomous car in the real world. However, the "Magic Formula" has one major limitation - it does not hold true when the vehicle is standing still or moving at slow speeds. This is a well-known issue [Pacejka, 2005]. To overcome this issue the author decided to add a velocity threshold - when then model (or tyre in this case) is moving at less than the defined threshold, then simply give the object friction as if it was a rigid object.

The author used this tyre behaviour modelling to create a Gazebo plugin which overrides the contact forces of any Gazebo simulation model it is assigned to, in this case, the tyres of the vehicle. To comply with the standards of Gazebo, this plugin was developed in C++ with the Boost and Ignition libraries, while ensuring good re-usability. The plugin is initialised and parameterised within the URDF file[3] of the robot model. The parameters chosen for the Hoosier tyres #43163 used on this vehicle were provided by the FS Tyre Test Consortium [Kasprzak and Gentz, 2006], an organisation set up by a number of academics to provide access to high-quality test results for tyres commonly used in Formula Student competitions. The parameters used are $B = 40$, $C = 1.6$, $D = 1.4$ and $E = -1.2$.

This plugin was then tested empirically in simulation and produced odd behaviour, where the car was sliding sideways constantly after it went above the velocity threshold. This behaviour remained largely unexplained but was most likely due to the method of computing *arctan* and it's well-explored limitations when not provided with a distinct numerator and denominator[4]. As this limitation was not the main aim of this project, the author opted for a less accurate but simple solution - piece-wise linearly approximating the "Magic Formula" which is done as shown in Figure 3.4.

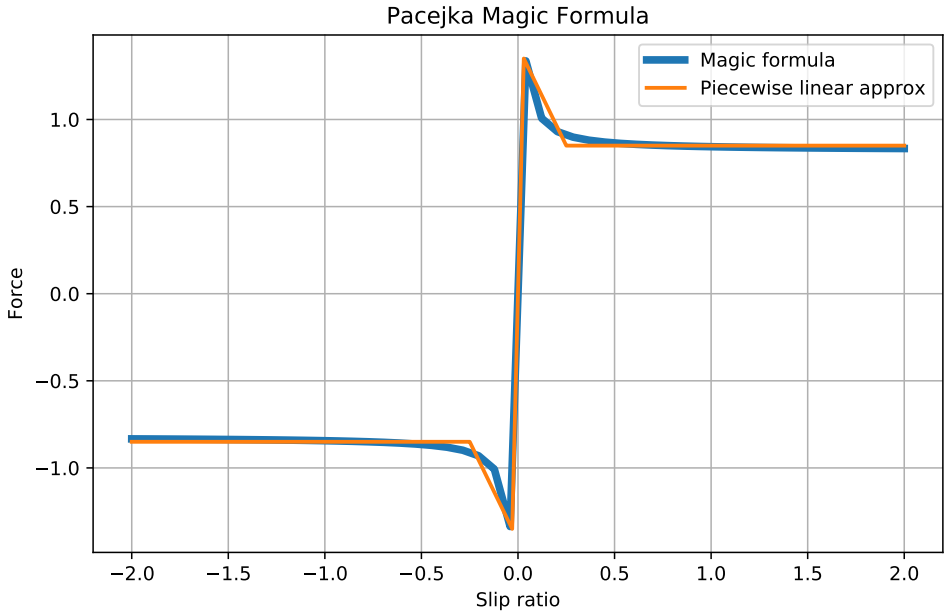


Figure 3.4: The Magic Formula and its piece-wise linear approximation

This can again be parameterised with only 4 variables - *static slip*, *dynamic slip*, *static friction* and *dynamic friction* as seen in Figure 3.5.

Based on this idea, the Gazebo plugin was then altered and empirically tested by trans-

[3] In the Gazebo simulation, URDF files are used to describe robot models on a high-level by composing them of multiple sub-modules

[4] Without access to the exact numerator and denominator, typical arctan functions in most programming languages can only output angles in the first or fourth quadrant, which has inherent limitations.

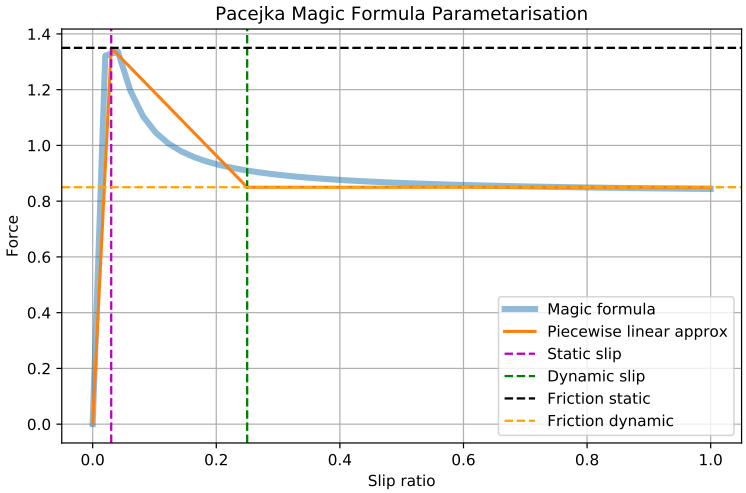


Figure 3.5: The piece-wise approximation of the Magic Formula for tyre friction.

lating the B, C, D and E parameters to the new parameter format. It is difficult to validate the performance of this plugin, but when the car was tested in simulation it was exhibiting behaviour similar to a car such as losing grip around tight corners and applying hard brakes at high speeds. These actions were not observed when the tyres were described as rigid body models.

3.3.4 Motors and control

Based on the vehicle setup as described in Section 2.2.2, the author has opted to simulate the vehicle control scheme as closely as possible to the real world. The ADS-DV employs a simplified control procedure. First, the software interface of the vehicle accepts 2 commands: the desired speed value in m/s and a desired steering angle in the range [-1, 1]. These commands are then sent to a low-level PID controller which attempts to achieve the desired commands as swiftly as possible.

To recreate this functionality, it was chosen to implement a ROS node due to better flexibility and debugging during development in comparison to Gazebo plugins. The purpose of this ROS node will be to translate the high-level driving commands to actuator commands which are then sent to Gazebo through the `joint_contoller` ROS package. The simulated vehicle model is then extended with the addition of:

- 4 Gazebo *velocity joints*[5] (one for each wheel) which simulate an electric motor with a PID loop. All of these joint interfaces were made identical due to the fact that the ADS-DV car features symmetrical transmission (Section 2.2.2). However, no information is available on the PID controller parameters of the real vehicle, therefore the P, I and D parameters for the simulation were tuned to offer similar acceleration performance to the ADS-DV.

[5] Analogous to motors which accepts a desired speed as input.

- 2 Gazebo *effort joints*[6] which simulate the steering actuator that controls the front wheels of the ADS-DV. However, there is one major difference here - the simulated model controls each front wheel steering independently. This was chosen due to the difficulty of implementing Ackermann steering as a physical link in the simulated vehicle model. Instead, the author chose to compute the steering angle of each wheel independently based on a standard Ackermann model.

Now it is only left to derive the commands for each actuator(joint).

Speed commands

To translate the desired vehicle speed to wheel speeds for the actuator it is only needed to translate linear velocity to angular velocity as seen in Figure 3.6. This is given by the equation:

$$V = r\omega$$

where V is the linear velocity, r is the wheel radius and ω is the angular velocity. Once angular velocities are derived, they are sent directly as input commands the actuators of the wheels in the simulation.

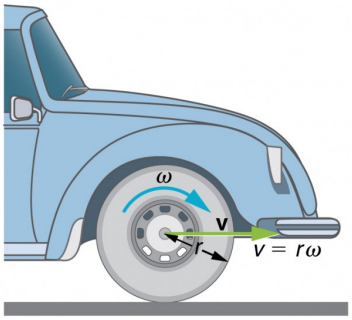


Figure 3.6: Relation between linear and angular velocities. Source: Lumen Learning

Steering commands

The idea of Ackermann steering is that the inside wheel should turn slightly sharper, resulting in better grip and control of cars. If α is defined as the desired steering command, α_L and α_R as the steering of the left and right wheel respectively, L as the wheelbase and W as track (width of the car); then on each turn there exists an *Instantaneous Centre of Curvature* (ICC) which is the point of intersection of the rear axle and the perpendiculars of both front wheels as seen in Figure 3.7.

The radius R around the ICC can be derived as:

$$R = \frac{L}{\tan(\alpha)}$$

with that α_L and α_R for $\alpha > 0$ can defined as:

$$\tan(\alpha_L) = \frac{L}{R - W/2}$$

$$\tan(\alpha_R) = \frac{L}{R + W/2}$$

which then lead to

$$\alpha_L = \arctan\left(\frac{L}{R - W/2}\right) \alpha_R = \arctan\left(\frac{L}{R + W/2}\right) \tag{3.2}$$

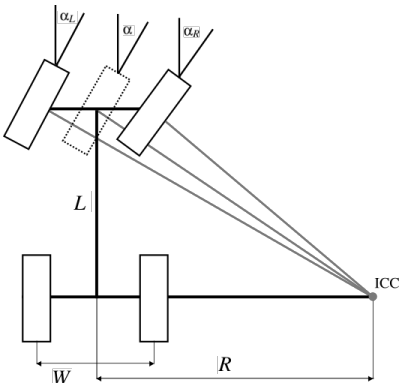


Figure 3.7: Ackermann steering model.

[6] Analogous to motors that accept torque values as input.

A special case not addressed by these equations is when $\alpha = 0$ as then R becomes infinite. In that case, we simply set $\alpha = \alpha_L = \alpha_R = 0$. Additionally, if $\alpha < 0$, it is still possible to use Equations 3.2, however, it is needed to subtract π from both α_L and α_R.

The final simulated vehicle model is then shown in Figure 3.8.

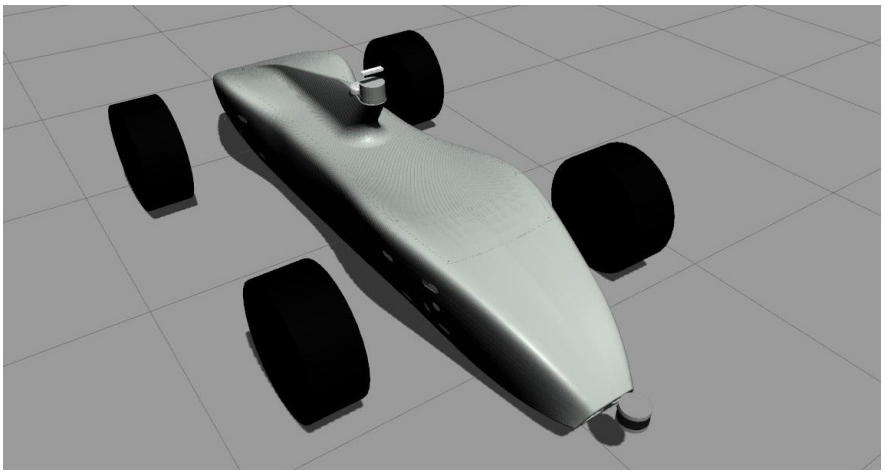


Figure 3.8: ADS-DV simulation model in Gazebo

3.4 Environment modelling

This section deals with providing the ”cone map” and accurate odometry of the vehicle to the MPPI algorithm as discussed previously in Section 3.2.

3.4.1 Tracks and map data

To create tracks in Gazebo, the author opted to use the Gazebo Model Editor which allows creating complex nested models with the standard GUI. Since the circuits in Formula Student consist of yellow, blue and orange traffic cones (Section 2), it is first needed to create these models of cones which will then be used to construct full tracks.

As the author of this project is not familiar with CAD, cone meshes were created by fellow EUFS team member Craig Martin. These were then converted to SDF files[7] with correct mass, the centre of mass and inertia matrices using the same procedure as in Section 3.3.1.

To construct racing tracks in simulation, one would ideally make an algorithm that generates random tracks based on the specifications in Section 2.1, construct SDF models in Gazebo and then create a plugin to automatically output the locations of cones in the world. However, the author deemed this option too complex and distracting from the overall goal of the project. Instead, it was chosen to manually construct tracks in Gazebo using the *Model Editor GUI* and then save them as SDF files. A Python script

[7] SDF is the standard Gazebo file format for defining simulated environment models.

was then developed to extract the cone locations in a 2D bird's eye view and save them as a CSV file. The benefit of this is that the cone locations can now be easily used and manipulated without directly involving the simulation, as seen later in Section 5.1, thus tackling the assumption that accurate cone locations are always available.

Two such tracks were constructed as seen in Figure 3.9.

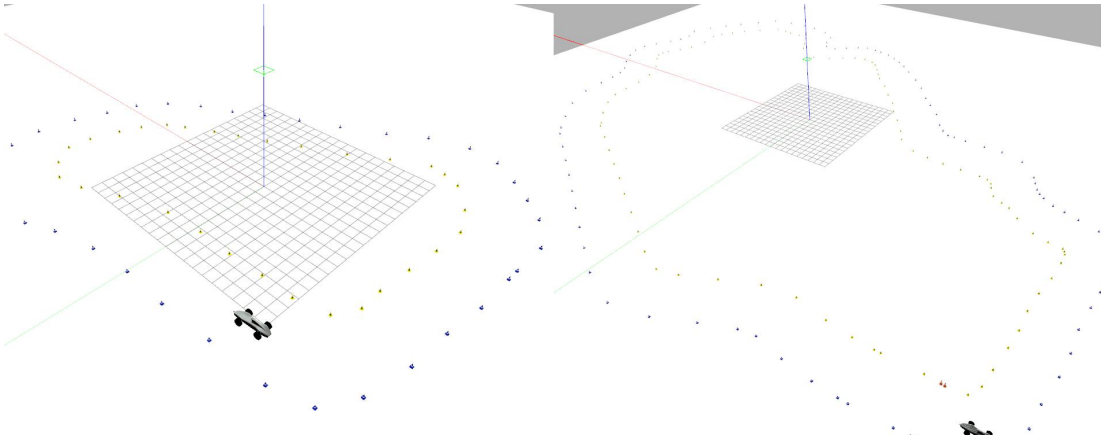


Figure 3.9: small_track (left) and big_track (right) simulated tracks in Gazebo

3.4.2 Odometry

For any type of model predictive control, it is required to have a representation of the state of the vehicle. For this project, this takes the shape of the ROS Odometry message format, which gives you information about the 3D position, orientation, linear velocity, angular velocity and uncertainty matrices for all of them. This is also the standard format adopted by the EUFS team, therefore integration with the team's system is simplified.

As this project is based on simulation, it is easy to obtain such data about the car. The author opted to use the `p3d_base_controller` Gazebo plugin which outputs exact odometry information relative to the origin of the simulation coordinate frame. It was also configured to perform similarly to the expected performance of the SBG Ellipse 2N INS (Appendix B): output odometry estimates at a rate of 200 Hz and induce Gaussian noise with $\sigma = 0.2$ to the position, orientation, linear and angular velocities.

3.5 Visualisation

Robots are complex systems which are difficult to debug in the typical sense of the word. Therefore, it is important to visualise the outputs modules and algorithms to ensure correct operation. The ROS ecosystem offers such a tool named RViz, which can visualise many of the standard message formats. The MPPI source code already outputs path planning data in the correct format, however, additional information is required for online debugging. For that purpose, the author developed a ROS node which

collects controller, pose, speed and command data from the simulation and visualises it as seen in Figure 3.10.

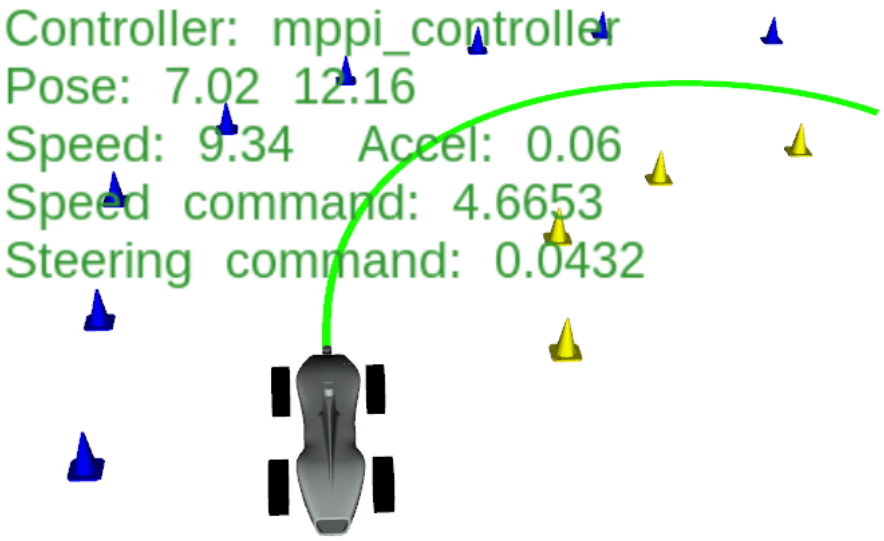


Figure 3.10: RViz visualisation of MPPI running in simulation.

Chapter 4

Model

To deploy the MPPI algorithm, it needs a model of the system dynamics. This is a well-explored topic, especially for an Ackermann-type car. However, instead of using traditional methods to identify the system, one can leverage the significant benefit of MPPI, the fact that it can use purely data-driven models.

In comparison to classical system identification [Kozlowski, 2012], data-driven model learning does not require hand-tailored equations; instead, machine learning models can derive them automatically, saving valuable design time. These models also have the benefit of exploiting non-linear relationships in the system dynamics model. One might argue that Ackermann-type vehicles are well-understood and straightforward, therefore a superior model can be handcrafted by an expert without the need for such black-box approaches; however, the empirical result in [Williams et al., 2017] suggest the opposite.

For the above reason, the author has chosen to leverage only data-driven models for this project, and the remainder of this chapter will be focused on learning the dynamics of the ADS-DV vehicle from the simulation designed in Chapter 3.

4.1 State representation

To forward-sample possible trajectories, the vehicle must be aware of the current position and orientation. Since it is a grounded robot, the state space can be limited to the x-position x_w, y-position y_w and yaw ψ_w in the world coordinate frame. These will be referred to as the kinematic state variables $\mathbf{x}_k$ and are described:

$$\mathbf{x}_k = \begin{pmatrix} x_w \\ y_w \\ \psi_w \end{pmatrix} \tag{4.1}$$

These variables can be seen in Figure 4.1 along with the associated coordinate frames of both the world and the vehicle.

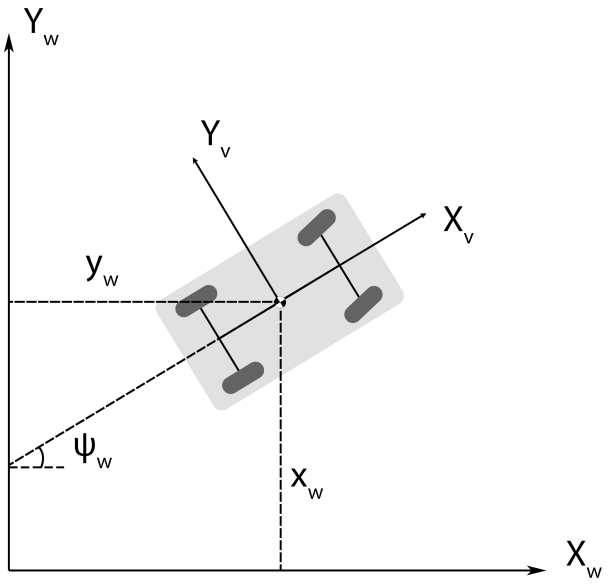


Figure 4.1: Coordinate frames of the vehicle and the kinematic state variables. X_w and Y_w represent the coordinate frame of the world. X_v and Y_v represent the coordinate frame of the vehicle. x_w, y_w and ψ_w are the kinematic state variables.

Given a **fixed timestep** Δt, it is possible to form the equation of motion as:

$$\mathbf{x}_k(t+1) = \mathbf{x}_k(t) + \mathbf{k}(\mathbf{x})\Delta t \tag{4.2}$$

where $\mathbf{k}(\cdot)$ gives the change of state for all kinematic state variables in the next timestep. The easiest method of calculating this is by simply calculating the distance between $\mathbf{x}_k(t)$ and $\mathbf{x}_k(t+1)$. Distance can easily be obtained with the formula $S = Vt$ where S is distance, V is velocity and t is time. This means that $\mathbf{k}(\cdot)$ must simple derive the velocities $\dot{x}_w$, $\dot{y}_w$, $\dot{\psi}_w$.

To homogeneously predict dynamics regardless of the absolute position of the car in the world frame, it is preferred to define the dynamic state in the vehicle frame. Given the velocities $\dot{x}_v$, $\dot{y}_v$ and the heading rate (yaw velocity) $\dot{\psi}_v$ in the vehicle frame, then we can apply an inverse 2D affine transformation to get their equivalents in the world coordinate frame:

$$\begin{aligned} \dot{x}_w &= \cos(\psi_w)\dot{x}_v - \sin(\psi_w)\dot{y}_v \\ \dot{y}_w &= \sin(\psi_w)\dot{x}_v + \cos(\psi_w)\dot{y}_v \\ \dot{\psi}_w &= \dot{\psi}_v \end{aligned} \tag{4.3}$$

Now that we have all of the velocities in the world coordinate frame, it is straightforward to define $\mathbf{k}(\cdot)$ and plug it back into Equation 4.2.

$$\mathbf{k}(\mathbf{x}) = \begin{pmatrix} \dot{x}_w \\ \dot{y}_w \\ \dot{\psi}_w \end{pmatrix} \tag{4.4}$$

Now, it is possible to define the dynamic state variables $\mathbf{x}_d$ which are defined as the variables that result in a change of the kinematics. These variables can include many different physical properties including linear and angular velocities. However, since the goal of this project is to predict the change of kinematics via the dynamic state variables $\dot{x}_v$, $\dot{y}_v$ and $\dot{\psi}$, it is possible to also include any other kinematic state variables that can help in these predictions. Additionally, the control signals sent to the car will also have an important effect on the dynamics and should also be taken into account here. Therefore, the equations of motion for $\mathbf{x}_d$ can be written as:

$$\mathbf{x}_d(t+1) = \mathbf{x}_d(t) + \mathbf{f}(\mathbf{x}_d, \mathbf{v}(t))\Delta t \tag{4.5}$$

where $\mathbf{v}(t) = (u_1(t) + \varepsilon_1(t), u_2(t) + \varepsilon_2(t)$ is the randomly perturbed control input. In this case u_1 is the speed and u_2 is the steering of the car; ε is noise. The function $\mathbf{f}(\cdot)$ gives the changes in the dynamic state variables $\mathbf{x}_d$ 1s ahead of timestep t. [Williams et al., 2017].

It is finally possible to define the full state space as:

$$\mathbf{x} = \begin{pmatrix} \dot{x}_k \\ \dot{x}_d \end{pmatrix} \tag{4.6}$$

where the exact definition of $\dot{x}_d$ will be explored later in Section 4.3.

It is finally possible to express the *state transition function* $\mathbf{F}(\cdot)$ introduced in Equation 2.1 by plugging in Equations 4.4 and 4.5:

$$\mathbf{x}(t+1) = \begin{pmatrix} \mathbf{x}_k(t) \\ \mathbf{x}_d(t) \end{pmatrix} + \begin{pmatrix} \mathbf{k}(\mathbf{x}(t)) \\ \mathbf{f}(\mathbf{x}_d(t), \mathbf{v}(t)) \end{pmatrix} \Delta t \tag{4.7}$$

The issue here is that $\mathbf{f}(\cdot)$ is not trivial to define as it is highly non-linear and dependent on several different factors. The remainder of this chapter will be focused on estimating this function using Neural Networks.

4.2 Data

This section deals with data collection, preprocessing and preparing it for learning algorithms.

4.2.1 Collection

Since the scope of this project is in simulation, kinematics and dynamics data were collected from the simulation as well. This was done by manually driving the car in simulation using an Xbox 360 controller while recording both the odometry from

Section 3.4.2 and the controls sent to the vehicle. The data was recorded in a *rosbag* which is a convenient tool for recording any data published by ROS.

Data collection in this project suffers from the typical problems of system identification, where one would like the data to explore as much as possible from the state space as possible [Kozlowski, 2012]. For that reason, one should carefully consider the state representation used, how the vehicle can be controlled to explore as much as possible from the state space and distribute data over the whole state space range. Inspired by the original work on MPPI [Williams et al., 2017], here the author has adopted the following data collection procedure:

1. Slow driving (3 - 9 m/s) around the track.
2. Zig-zag manoeuvres at slow speeds (3 - 6 m/s).
3. Aggressive driving, attempting to set the fastest lap times, while not hitting any cones. Average speed 12 m/s.
4. Gradually accelerating to 15 m/s and then gradually decelerating into random turns.
5. Full speed acceleration to 15 m/s and then stopping as fast as possible and occasionally sliding.
6. Driving in the figure of 8 from Section 2.1 as fast as possible. Involves significant sliding.

Manoeuvres in points 1-3 are performed for 3 minutes one way around the track and 3 minutes the other way around a track. Manoeuvres 4-6 are performed for 5 minutes each.

Using this procedure, a dataset was collected in simulation, and a pair plot was used to validate adequate state space exploration (seen in Figure 4.2). Noteworthy points:

1. The control variables are explored excellently.
2. There seems to be a bias towards positive values of $\dot{y}_v$. This might be possible due to outliers within the dataset, which have resulted from extreme sliding at high speeds (pairwise between $\dot{x}_v$ and $\dot{y}_v$). Since this is an outlier, hopefully it should not disturb later work and the models should learn to ignore it.
3. The speed command and $\dot{x}_v$ are biased towards positive values. This is to be expected since the car is moving forward the majority of the time.

4.2.2 Processing

The data type collected from the simulation is shown below in Table 4.1. Both the *odometry* and *control* data have timestamps accurate to 1 ns. The *odometry* data is in the world-frame.

The goal here is to preserve as much of the original data as possible as the author later intends to experiment with the variables to identify which combination of them results in the best dynamics prediction.

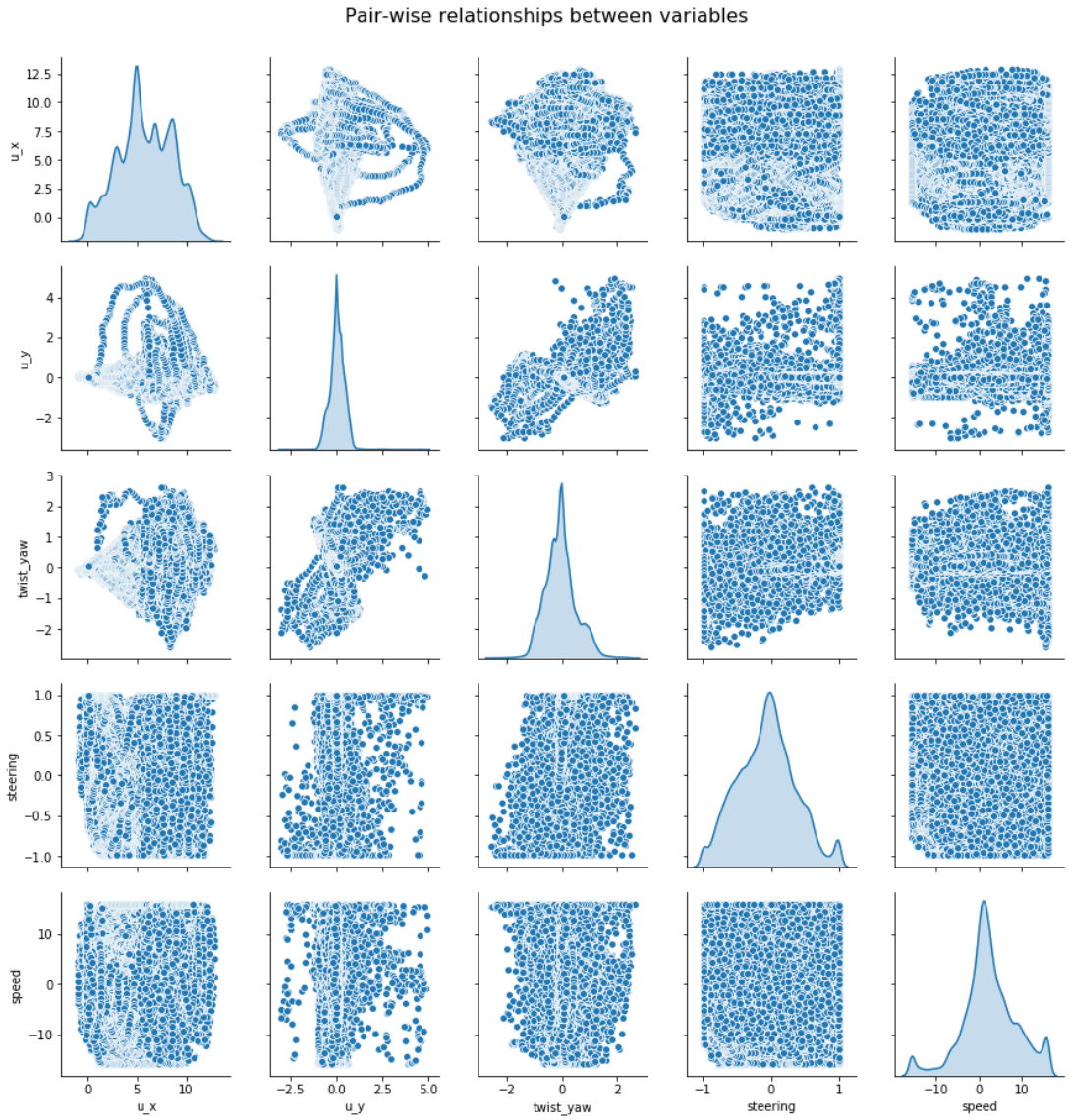


Figure 4.2: Both axis contain all dynamic state variables and control commands. Plots on the diagonal show the distribution of each of the 5 variables. The denser a plot is, the better the state space is explored. `u_x` and `u_y` are $\dot{x}_v$ and $\dot{y}_v$ respectively; `twist_yaw` is the heading rate $\dot{\psi}$.

Data processing was done with Python as it offers easy methods of loading rosbag data[3].

Data matching

The *odometry* and *control* data come from different sources and as such their timestamps are not matched as seen in Table 4.2. This can easily be solved with a full outer database join, which matched the row of our dataset; results can be seen in Table 4.3. However, the outer join operation also produces data entries with duplicated timestamps.

[3]rosbag data was loaded with the python package rosbag_pandas

odometry[1]	pose	pose	position	x
				y
				z
			orientation	x
				y
				z
				w
		covariance		
	twist	twist	linear	x
				y
				z
			angular	x
				y
				z
		covariance		
control[2]	speed			
	steering			

Table 4.1: Data type and format collected from simulation. Twist is the ROS name for velocities.

Data interpolation
The data types are collected come at different rates - 200 Hz and 50 Hz for *odometry* and *control* respectively. Ideally, we want to get as dense and accurate data as possible. From Chapter 3, we know that once a control command is sent to the car, it keeps executing it until it receives another command. Therefore, it is possible to piece-wise interpolate the *control* data to 200 Hz. Afterwards, it is also easy to remove rows with duplicated timestamps. The results of this is shown in Table 4.4.

Data re-sampling
Now that we have dense, high-frequency dataset, it is finally possible to re-sample the dataset to the running rate of the MPPI algorithm. For the purposes of this report, the controller is running at 50 Hz, which gives us a timestep of $\Delta t = 1/50 = 0.02s$. The re-sampled data is shown in Table 4.5. Additionally, the data timestamps were reset to start from $t = 0$ for convenience.

Please note that the data shown in the tables of this section is only part of all data collected. Most columns from the tables are omitted for brevity and simplicity.

The dataset is now fully reconstructed and it can be altered into the correct format for learning.

Quaternion to Euler conversion
The orientation data collected is in the quaternion format[4] as it is the standard in ROS and address some of the issues of Euler angles[5]. However, those issues do not affect the

[4]Quaternions are an alternative method of representing angles with complex numbers.

[5]Euler angle representations suffer from the problem known as Gimbal lock. This occurs when two of the three gimbals are driven into a parallel configuration, locking the rotation into only 2D space

timestamp	speed	steering	pose_x	pose_y	twist_x
00:02:28.545000	NaN	NaN	-57.505946	-3.786475	3.561731
00:02:28.545000	0.0	-0.005867	NaN	NaN	NaN
00:02:28.550000	NaN	NaN	-57.488142	-3.788593	3.560210
00:02:28.555000	NaN	NaN	-57.470346	-3.790711	3.558706
00:02:28.560000	NaN	NaN	-57.452557	-3.792829	3.557174
00:02:28.565000	NaN	NaN	-57.434775	-3.794947	3.555681
00:02:28.565000	0.0	-0.005867	NaN	NaN	NaN
00:02:28.570000	NaN	NaN	-57.417002	-3.797064	3.554153
00:02:28.575000	NaN	NaN	-57.399235	-3.799181	3.552597
00:02:28.580000	NaN	NaN	-57.381477	-3.801298	3.551097

Table 4.2: Raw dataset obtained from a rosbag. Only first 10 entries shown.

timestamp	speed	steering	pose_x	pose_y	twist_x
00:02:28.545000	NaN	NaN	-57.505946	-3.786475	3.561731
00:02:28.545000	0.0	-0.005867	-57.505946	-3.786475	3.561731
00:02:28.545000	0.0	-0.005867	NaN	NaN	NaN
00:02:28.550000	NaN	NaN	-57.488142	-3.788593	3.560210
00:02:28.555000	NaN	NaN	-57.470346	-3.790711	3.558706
00:02:28.560000	NaN	NaN	-57.452557	-3.792829	3.557174
00:02:28.565000	NaN	NaN	-57.434775	-3.794947	3.555681
00:02:28.565000	0.0	-0.005867	-57.434775	-3.794947	3.555681
00:02:28.570000	NaN	NaN	-57.417002	-3.797064	3.554153
00:02:28.575000	NaN	NaN	-57.399235	-3.799181	3.552597

Table 4.3: Dataset from Table 4.2 after an outer join. Only first 10 entries shown.

timestamp	speed	steering	pose_x	pose_y	twist_x
00:02:28.545000	0.0	-0.005867	-57.505946	-3.786475	3.561731
00:02:28.550000	0.0	-0.005867	-57.488142	-3.788593	3.560210
00:02:28.555000	0.0	-0.005867	-57.470346	-3.790711	3.558706
00:02:28.560000	0.0	-0.005867	-57.452557	-3.792829	3.557174
00:02:28.565000	0.0	-0.005867	-57.434775	-3.794947	3.555681
00:02:28.565000	0.0	-0.005867	-57.434775	-3.794947	3.555681
00:02:28.570000	0.0	-0.005867	-57.417002	-3.797064	3.554153
00:02:28.575000	0.0	-0.005867	-57.399235	-3.799181	3.552597
00:02:28.580000	0.0	-0.005867	-57.381477	-3.801298	3.551097
00:02:28.585000	0.0	-0.005867	-57.363726	-3.803415	3.549587

Table 4.4: Dataset from Table 4.3 with the speed and steering columns now interpolated to match the data rate of the rest of the columns. Only first 10 entries shown.

dynamics prediction; therefore it is simpler and easier to use Euler angles. To convert

timestamp	speed	steering	pose_x	pose_y	twist_x
00:00:00	0.0	-0.005867	-57.505946	-3.786475	3.561731
00:00:00.020000	0.0	-0.005867	-57.434775	-3.794947	3.555681
00:00:00.040000	0.0	-0.005867	-57.363726	-3.803415	3.549587
00:00:00.060000	0.0	-0.005867	-57.292798	-3.811878	3.543533
00:00:00.080000	0.0	-0.005867	-57.221991	-3.820338	3.537498
00:00:00.100000	0.0	-0.005867	-57.151304	-3.828791	3.531439
00:00:00.120000	0.0	-0.005867	-57.080739	-3.837237	3.525438
00:00:00.140000	0.0	-0.005867	-57.010293	-3.845674	3.519429
00:00:00.160000	0.0	-0.005867	-56.939967	-3.854101	3.513418
00:00:00.180000	0.0	-0.005867	-56.869761	-3.862515	3.507472

Table 4.5: The dataset re-sampled to the correct frequency (50 Hz). Only first 10 entries shown.

the angles, the *1-2-3 Euler angle conversion* was used. If the quaternion angles are represented with x, y, z and w then:

$$\begin{aligned} \phi &= \tan^{-1}\left(\frac{2zw+2xy}{w^2-z^2-y^2+x^2}\right) \\ \theta &= -\sin^{-1}\left(2yw-2xz\right) \\ \psi &= \tan^{-1}\left(\frac{2yz+2xw}{y^2+x^2-z^2-w^2}\right) \end{aligned} \tag{4.8}$$

This conversion is not needed for the angular velocities, as they already are in Euler angles[6].

Roll and pitch detection

During aggressive driving data collection, it frequently happened that the car rolled over while attempting to corner. Usually, this would render the whole dataset collected until that point useless. To avoid that, the author incorporated *bad roll and pitch detection* which checks the data for $|\phi_v| > 40°$ and $|\theta_v| > 20°$. If such data is found then, the user is warned and given the option to remove that data and any other data following it. This has the benefit of utilising previously unusable datasets, reducing the cost of collecting bootstrapping data.

Vehicle frame velocities

The velocities of the car are needed in the vehicle frame for Equation 4.5, then it is needed to convert $\dot{x}_w$ and $\dot{y}_w$ from the dataset into $\dot{x}_v$ and $\dot{x}_v$. This is done by simply applying a 2D rotation matrix which results in:

$$\begin{aligned} \dot{x}_v &= \cos(\psi)\dot{x}_w + \sin(\psi)\dot{y}_w \\ \dot{y}_v &= -\sin(\psi)\dot{x}_w + \cos(\psi)\dot{y}_w \end{aligned} \tag{4.9}$$

[6]ROS documentation for the odometry message type - `http://docs.ros.org/melodic/api/nav_msgs/html/msg/Odometry.html`

A plot is used to verify the correct conversion - Figure 4.3.

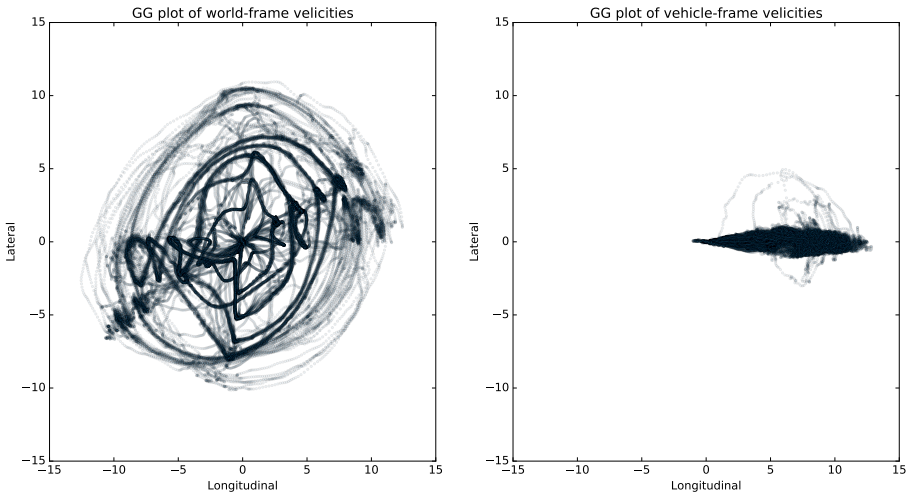


Figure 4.3: Plot of the velocities in the vehicle frame. Longitudinal and lateral velocities are $\dot{x}_v$ and $\dot{y}_v$ respectively.

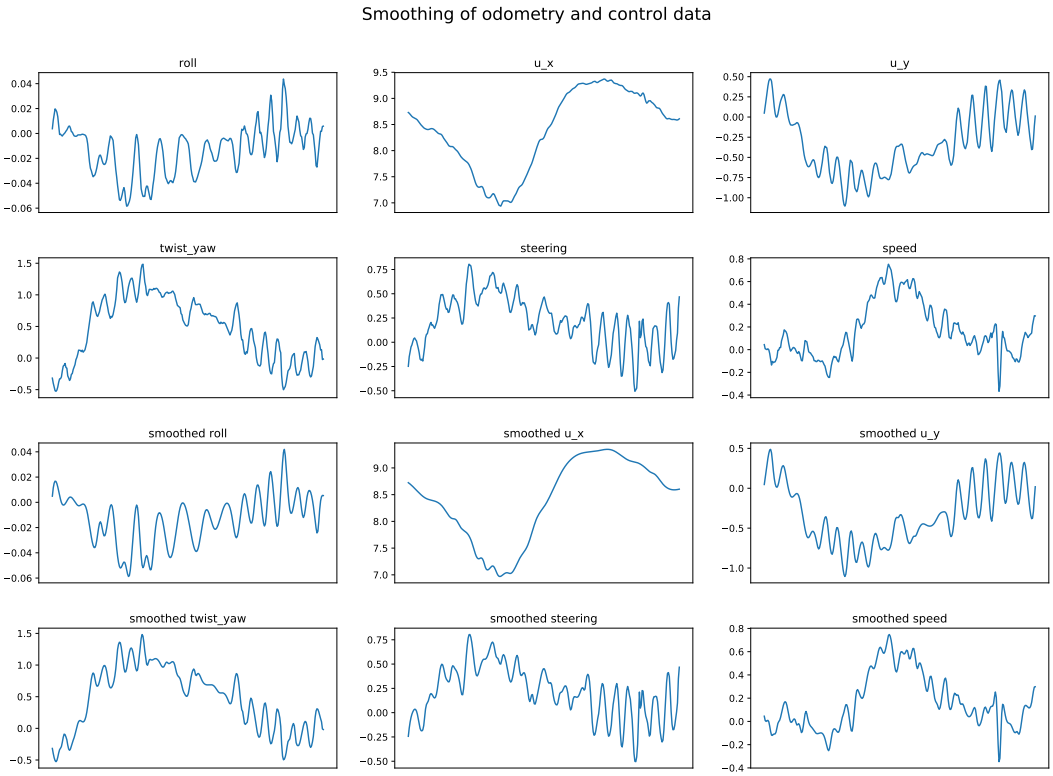


Figure 4.4: Data plotted against 2s of time. Top two rows show the raw data, whereas the bottom two rows are the smoothed version. Note this only shows part of the full dataset. `u_x` is forward velocity in the vehicle frame, `u_y` is lateral velocity in the vehicle frame, `twist_yaw` is the heading rate.

Data smoothing

One of the issues with the data collection process is the noise from the physical controller and the noise induced by the odometry simulation (Section 3.4). Empirical results suggest that this noise significantly decreases the ability of neural networks to learn the dynamics of the car accurately. As suggested by Grady Williams, one of the authors of MPPI, the collected data was smoothed with one-dimensional splines fit through time. This can be seen in Figure 4.4. The scipy spline library[7] was used for this with an empirically estimated smoothing factor $s = \sigma/10$ where σ is the standard deviation for each data column.

Data preparation for learning

To use the data for machine learning, it must be shaped into the correct format. Keeping to the standard notation, the inputs are defined as:

$$\mathbf{X}(t) = \big(\mathbf{x}_d(t), \mathbf{v}(t)\big)$$

Then, the targets are defined as:

$$\mathbf{Y}(t) = \big(\mathbf{X}(t+\Delta t) - \mathbf{X}(t)\big)\Delta t$$

Δt is needed in this case to scale up the outputs into a suitable range. Without that, the targets have low magnitude which decreases the performance of NN.

4.3 Dynamic variables identification

Although Ackermann-type vehicle dynamics are well explored, they still remain highly non-linear systems which are difficult to identify. This section will focus on exploring how to best represent the dynamic state variables $\mathbf{x}_d$. From Section 4.1, Equation 4.5, it is known that at least $\dot{x}_v, \dot{y}_v$ and $\dot{\psi}_v$ are needed, which will now be called the **base dynamic state variables**. Additionally, it is possible to extend $\mathbf{x}_d$ with all other kinematics and dynamic state variables.

It is possible to analyse which dynamics variables to include with classical dynamics analysis. However, since this project utilises NN to learn the dynamics for this project, it is possible to identify

Dynamic State Variables	MSE
$\dot{x}$, $\dot{y}$, ψ	2.8815
$\dot{x}$, $\dot{y}$, ψ, ϕ	1.5916
$\dot{x}$, $\dot{y}$, ψ, θ	2.5183
$\dot{x}$, $\dot{y}$, ψ, ψ	2.7266
$\dot{x}$, $\dot{y}$, ψ, $\dot{\phi}$	2.7371
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ	2.7786
$\dot{x}$, $\dot{y}$, ψ, ϕ , ψ	2.3529
$\dot{x}$, $\dot{y}$, ψ, ϕ , $\dot{\phi}$	2.3484
$\dot{x}$, $\dot{y}$, ψ, ϕ , $\dot{\theta}$	2.4891
$\dot{x}$, $\dot{y}$, ψ, θ, ψ	2.7267
$\dot{x}$, $\dot{y}$, ψ, θ, $\dot{\phi}$	2.7724
$\dot{x}$, $\dot{y}$, ψ, θ, $\dot{\theta}$	2.7666
$\dot{x}$, $\dot{y}$, ψ, ψ, $\dot{\phi}$	2.8512
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ, $\dot{\theta}$	2.4802
$\dot{x}$, $\dot{y}$, ψ, ψ, $\dot{\phi}$, $\dot{\theta}$	2.8883
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ, ψ, $\dot{\theta}$	2.6387
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ, ψ, $\dot{\phi}$	2.5487
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ, ψ, $\dot{\phi}$	2.7914
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ, ψ, $\dot{\phi}$, $\dot{\theta}$	2.7342
$\dot{x}$, $\dot{y}$, ψ, ψ, $\dot{\theta}$	2.8749
$\dot{x}$, $\dot{y}$, ψ, $\dot{\phi}$, $\dot{\theta}$	2.7725
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ, ψ	2.4032
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ, $\dot{\phi}$	2.3587
$\dot{x}$, $\dot{y}$, ψ, θ, ψ, $\dot{\phi}$	2.8040
$\dot{x}$, $\dot{y}$, ψ, θ, ψ, $\dot{\theta}$	2.9034
$\dot{x}$, $\dot{y}$, ψ, ϕ , θ, $\dot{\phi}$, $\dot{\theta}$	2.6430

Table 4.6: Training of different NN for identifying the best dynamics representation. All variables here are in the vehicle-frame.

[7] Scipy is one of the most popular packages for Python, which adds a wide set tools for scientific computing. Its *interpolation* module provides an easy to use splining and smoothing toolset.

the most appropriate dynamic state variables by training several networks and empirically analyse their performance. This black-box approach allows for rapid testing and evaluation of several different state representations and identifying the best one.

Experiments were set up using PyTorch[8] and a big neural network with 2 hidden layers of 128 neurons each was used which allows for learning complex non-linearities even with all possible dynamic state variables. All combinations of state variables were explored as inputs and outputs to the network. However, results are calculated only by finding the Mean Square Error (MSE) between the base dynamic state variables. All hyper-parameters used for these experiments are presented in Appendix C.

From the results in the Table 4.6, it can be seen that that the best variables for $\mathbf{x}_d$ are $\dot{x}_v$, $\dot{y}_v$, ψ and ϕ. Coincidentally, these are the same dynamic state variables used in [Williams et al., 2016] [Williams et al., 2017]; however, the choice in those papers is justified by classical dynamics system identification. This proves that neural networks are capable of identifying dynamic state variables without requiring costly development time and hand-analysing dynamic systems.

4.4 Learning dynamics

The focus of this section is to explore learning the dynamics of the vehicle with neural networks. Throughout this section fully-connected Neural Networks will be referred to as NN for brevity.

4.4.1 Framework

Machine learning is still a relatively young field, and as such, no standardised full toolset has been developed. However, as it was found throughout the development of this project, that a capable and feature-rich framework is key to fast development, evaluation and storing reproducible results. For that reason, the author has progressively developed the **sysid dynamics learning framework**, based on Python and PyTorch.

Instead of listing the functionality of the framework, an outline of a typical experiment process for a NN is provided below:

1. Each experiment is automatically labelled with a timestamp. Names have the format `system_id_net,[timestamp]`.
2. The preprocessed data from Section 4.2 shaped into the correct format for learning and converted into PyTorch tensors. At this point, the user can select which data columns he would like to use as well as specify Δt. Then target outputs $\mathbf{Y}$ are generated, and the user also has the option to normalise the data. It is possible to load multiple datasets and combine them into one for training..

[8] PyTorch is one of the most popular modern machine learning tool-kits which is specialised towards neural networks

3. The dataset is then split into training, validation and testing sets.
4. An NN model is then created by specifying the number of inputs, number of outputs, number of hidden layers and their dimensions, whether to use bias parameters and the choice of activation functions.
5. The experiment is started be specifying the number of epochs, batch size, the optimiser (RMSProp or Adam), the learning rate and weight decay factor.
6. An experiment folder is automatically generated, and all hyper-parameters are saved in a YAML file. On every epoch, the model is evaluated on the full training and validation sets, errors and the model itself are saved within the experiment folder.
7. At the end of the experiment, all errors on every epoch are saved as a statistic ,and a plot is generated plotting testing and validation errors against epochs.
8. The model with the lowest validation error is selected as the best model and is evaluated on the test set.
9. The best model is then loaded into the simulation and automatically tested for five laps against small_track (Section 3.4.1). Average lap times are then saved within the experiments folder. If the car takes more than 1 minute to complete a lap, then it is labelled as a failed model as it most likely has went outside the track. Note: the first lap of evaluation is discarded as the car starts from a full stop.

All of this functionality is embedded into the files `sysid_datasets.py`, `sysid_models.py` and `sysid_framework.py`. Then they are combined in a high-level script which control all of the hyper-parameters - `NN_training.py`.

4.4.2 Setting a Baseline

Before running any serious machine learning experiments, it is vital to set a baseline to which all further experiments can be compared. Since this project is based on previous work, a reasonable baseline can be set by reproducing the NN of the paper [Williams et al., 2017].

The hyper-parameters used for this are:

- 2 fully-connected hidden layers with 32 neurons with biases.
- tanh activation function.
- No normalisation.
- RMSProp optimiser with learning rate 10^{-3}, $\alpha = 0.99$, $\varepsilon = 10^{-8}$ and L2 weight normalisation $wd = 10^{-6}$ [Tieleman and Hinton, 2012].
- 1000 epochs with batch size 100.
- 80% /10% /10% train/validation/test dataset split.

Results from this experiment are seen in Table 4.7 and Figure 4.5. The hyper-parameters used here will be considered the default ones for all future experiments.

Best epoch index	Best val error	Average lap time
325	1.2519	$11.07 \pm .14$

Table 4.7: Baseline NN results. Lap time is averaged over 5 laps of small_track. The standard deviation for the lap times is also given.

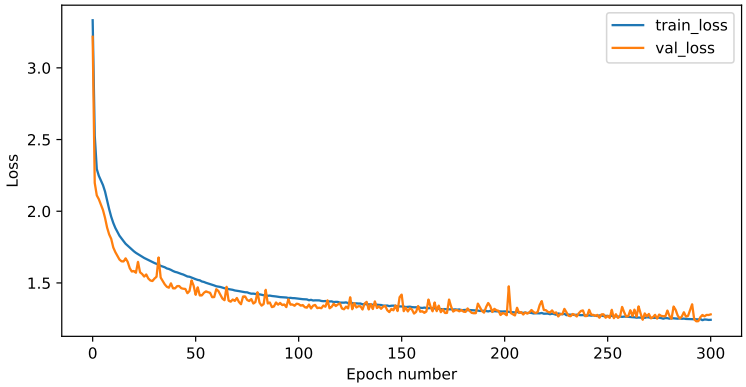


Figure 4.5: Training and validation error plotted against epochs for the baseline neural network.

4.4.3 Neural Network Experiments

With the automatic framework developed in the last section, a series of experiments were tailored and left them running for long periods of time. A common random seed was used in all experiments to guarantee reproducible results. Select hyper-parameters and their effects on learning will be explored below.

Network architecture
The choice of this is always an open question in the field of machine learning. In principle, the bigger the network, the more complex functions it can learn; however, in the case of this project, computational time must also be taken into account. The primary benefit of NNs in the context of dynamics is that they can learn non-linearities, however, for that to work, a minimum of two hidden layers are needed. On the other hand, more than three hidden layers introduce heavy computational requirements. With that in mind, the author chose to explore hidden layers of dimensions seen in Table 4.8, results shown are the ones for the hyper-parameters that achieved the best results with not normalised data. Although less complex networks achieve larger validation errors, they produce good results. However, that is only true, when they manage to finish laps at all as they also have higher failure rates. Therefore the NN architecture that strikes a good balance between lap time, low computational complexity and low failure rates is the one with two hidden layers with 48 neurons each.

Evaluation metrics
Given the results previously shown in Table 4.8, a keen reader would naturally ask the question whether the two main evaluation metrics - validation error and lap times are

Hidden layer size	Number of multiplications	Best epoch index	Best validation error	Average lap time	Failure rate
8, 8	1536	953	1.6203	$10.90 \pm .10$	37.5%
16, 16	6144	999	1.3861	$10.97 \pm .07$	37.5%
32, 32	24576	983	1.3521	$10.89 \pm .11$	25%
48, 48	55296	496	1.191	$11.07 \pm .04$	12.5%
64, 64	98304	996	1.2496	$11.08 \pm .04$	12.5%
8, 8, 8	12288	859	1.423	$10.72 \pm .12$	37.5%
8, 16, 8	24576	641	1.4751	$10.91 \pm .15$	25%
8, 16, 16	49152	993	1.3766	$10.78 \pm .09$	12.5%
16, 16, 16	98304	993	1.3730	$11.01 \pm .16$	25%
16, 32, 16	196608	998	1,3387	$10.88 \pm .14$	25%
32, 32, 32	786432	995	1.2710	$10.81 \pm .10$	12.5%

Table 4.8: Analysis of different NN architectures. The number of multiplications refers to the number of multiplications required to execute one forward propagation for the network (a crude measure of computational complexity). Lap time is averaged over five laps. The failure rate is defined by the percentage of time the car steers off the track.

correlated at all. To aid exploration of this question, Figure 4.6 plots validation loss against lap time. From it, one can derive that there is virtually no correlation, which is further confirmed by the derived Pearson correlation coefficient -0.0835[9].

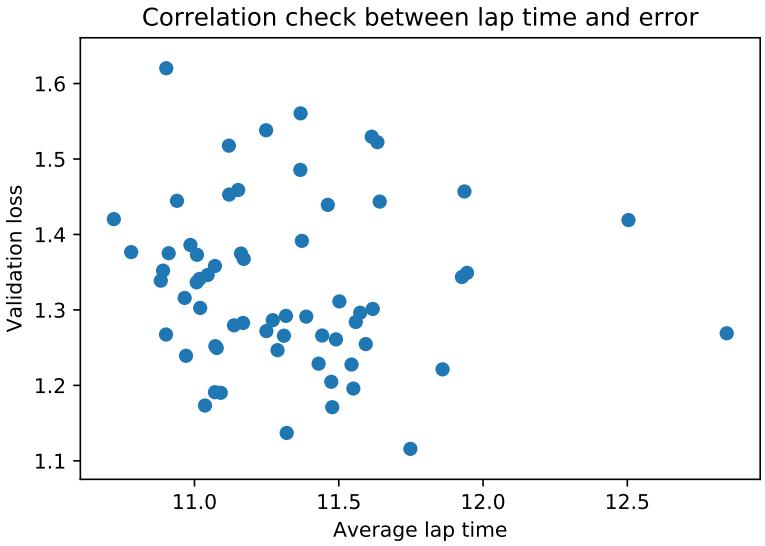


Figure 4.6: Investigating possible correlation between validation loss and lap times. This uses data of all experiments without data normalisation.

In this case, what is the correct evaluation metric? Well, both are correct in their own right. The validation loss is the absolute best metric, given the dataset provided,

[9]Correlation close to 0 means that there is no correlation between the variables.

however, as the model is trying to learn a high-dimensional continuous state space, the dataset used is only a fraction of the full state space. Ideally, one would obtain a dataset that is representative of the full state space; however, that is difficult and costly, especially in the real world when dealing with real hardware. On the other hand, the average lap time metric is even more empirical. In theory, this should be the best metric for the given task; however, this is also prone to issues. The MPPI algorithm is not trying to minimise lap times directly, it is trying to reduce lap times while aiming to minimise its cost function (covered in Chapter 5) which also ensures that the car stays on track. In the cases of dynamic models with high validation losses but good lap times, this is usually the result of the vehicle attempting risky manoeuvres, which might decrease lap time but also pose the risk of throwing the car outside of the track.

In conclusion, there is no single *best evaluation metric* to be used for the dynamics model; therefore the author has chosen to use both.

Activation function

The most popular activation functions in NN are the linear, sigmoid, hyperbolic tangent (tanh) and ReLU activation functions as shown in Figure 4.7. Since both the inputs and outputs of the NNs used in this project are symmetric and do not follow the classical meaning of activation functions[10], learning dynamics requires a symmetrical activation function along the x and y axis. This immediately rules out the sigmoid and ReLU activation functions. The linear activation function is usually not used within hidden layers as it results in exploding gradients. Thus, I have opted to use the tanh activation function for all hidden layers and the linear activation function for the output layers.

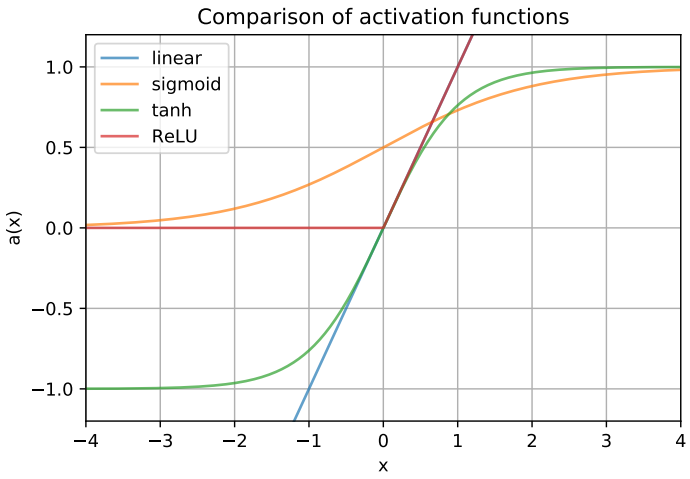


Figure 4.7: Common activation functions used in neural networks.

Learning rule

In this project I have considered the two most popular learning rules - RMSProp [Tieleman and Hinton, 2012] and Adam [Kingma and Ba, 2014]. Both of these are ran for every different NN architecture considered above with varying learning rates. RMSProp was used with L2 weight normalisation of $wd = 10^{-6}$, $\alpha = 0.99$ and $\varepsilon = 10^{-8}$.

[10]Usually, activation functions are ”turned on” by a certain combination of inputs, otherwise, they remain inactive.

Adam was used with L2 weight normalisation of $wd = 10^{-6}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$. Results were overall similar across the different NN architectures; an example of those for the network with two hidden layers of 48 neurons each is shown in Figure 4.8. Adam is faster to converge in most cases and therefore will be the proffered option throughout this report.

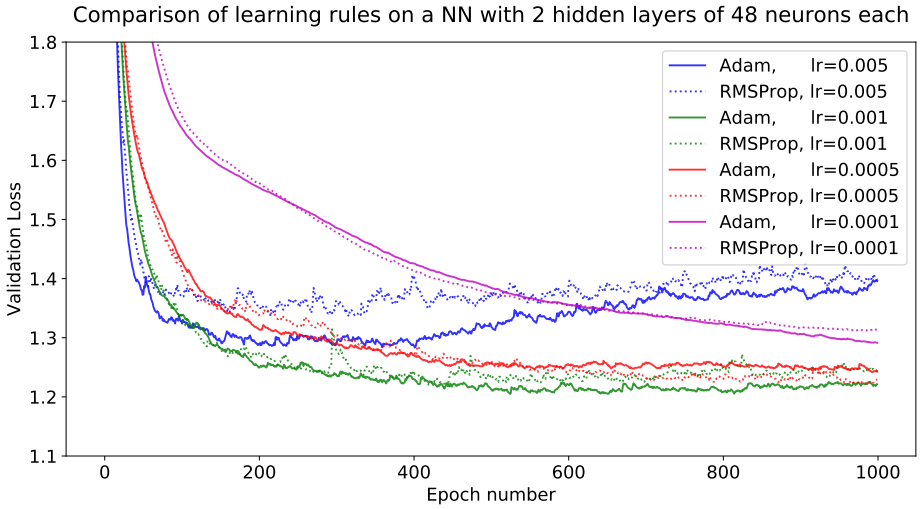


Figure 4.8: RMSProp and Adam learning rules compared with different learning rates. Plots are smoothed for readability.

Normalisation

This is entirely standard when it comes to the field of machine learning. Normalisation allows neural networks to take into account all input variables equally [Renals, 2018]. However, an interesting observation is that in the original work on MPPI [Williams et al., 2017] no normalisation was used.

Adding normalisation to the dynamics learning is not trivial as it is difficult to estimate the overall range of the dynamic variables since the dataset obtained is not necessarily a representative of the full state space. Regardless, the author chose to normalise every dynamic state input according to:

$$\hat{x}_i = \frac{x_i - mean(x_i)}{std(x_i)}$$

The steering was left as is since it is already in unit scale. The speed command was scaled down by 15 to be within the range [-1, 1].

This normalisation was also implemented within the CUDA source code to validate lap time performance. To simplify the process, the author created a custom NN model saving structure in an NPZ compressed format. The saved model consists of full details of the normalisation which are then loaded in CUDA and then all input variables are normalised in parallel on each NN forward-propagation

Then the same experiments as in Table 4.8 are run but now with normalised data. From the results in Table 4.9, it can be seen that the validation error is reduced across the

board which should in theory translate to better performance but average lap times are slightly worse. However, the failure rates reveal a different picture - they are all significantly reduced, with most two hidden layer networks having a perfect 0% failure rate. Normalisation has appeared to stabilise the dynamics models and has made them significantly more consistent, which is of immense value.

Hidden layer size	Number of multiplications	Best epoch index	Best validation error	Average lap time	Failure rate
8, 8	1536	764	1.3469	$10.89 \pm .09$	25%
16, 16	6144	985	1.2142	$10.98 \pm .09$	25%
32, 32	24576	974	1.1975	$10.89 \pm .09$	0%
48, 48	55296	977	1.1586	$10.85 \pm .05$	0%
64, 64	98304	952	1.1209	$11.14 \pm .18$	0%
8, 8, 8	12288	523	1.3154	$10.98 \pm .09$	0%
8, 16, 8	24576	960	1.2679	$11.18 \pm .21$	25%
8, 16, 16	49152	987	1.2304	$11.18 \pm .11$	25%
16, 16, 16	98304	982	1.2481	$11.14 \pm .09$	16.7%
16, 32, 16	196608	940	1,1476	$11.12 \pm .17$	25%
32, 32, 32	786432	820	1.1316	$10.91 \pm .07$	16.7%

Table 4.9: Analysis of different NN architectures with normalised input variables. The number of multiplications refers to the number of multiplications required to execute one forward propagation for the network (a crude measure of computational complexity). Lap time is averaged over five laps. The failure rate is defined by the percentage of time the car steers off the track.

Best model

In conclusion to this subsection, The best model identified achieves a validation error of 1.1586 and an average lap time 10.85 s. The hyper-parameters used for this model are:

- Name: `system_id_net,2019-02-20T05:57:26.263482`
- 2 fully-connected hidden layers with 48 neurons with biases.
- tanh activation function for the hidden layers.
- With normalisation.
- Adam optimiser with learning rate 10^{-4}.
- 1000 epochs with batch size 100.
- 80% /10% /10% train/validation/test dataset split.

All of the dynamics models in this section were evaluated using the MPPI algorithm. As the cost function of the algorithm was not yet explored at the time of writing this, the standard cost function available in the Autorally repository was used. A more detailed exploration of this can be found in Appendix C along from full results from the experiments of this section.

4.4.4 Online learning

A fairly obvious improvement that can be made on the work in [Williams et al., 2017] is to add online learning. It is still possible to bootstrap the dynamics model with a dataset and train it online; then when the MPPI algorithm is running, it can improve its model of the dynamics online. In theory, this sounded perfect. If the car is currently in $\mathbf{x}_t$, on timestep t, the car will plan a trajectory and predict $\hat{\mathbf{x}}_{t+1}$; then when it reaches $t+1$, it can compare the real state $\mathbf{x}_{t+1}$ and the prediction it made $\hat{\mathbf{x}}_{t+1}$. With that, it can backpropagate through the neural network and further optimise it.

The author proceeded to implement this into the source code provided by GeorgiaTech. To ensure real-time performance, it had to be implemented in CUDA, which proved a considerable challenge due to the lack of experience with the language. In particular, the author implemented the mean square loss computation, the gradient computation and the actual backpropagation through the network. Although better results were thus far obtained with the Adam optimiser, it was decided to implement the RMSProp optimiser for the online learning task as it is simpler. Default hyper-parameters were used for it with L2 weight normalisation $wd = 10^{-6}$. For debugging purposes, a model saving feature was also added which saved the model on every 100 timesteps, with the idea to use those models to evaluate results.

An experiment was then set up using the best-trained model from the previous section as a bootstrapping dynamics model. Then the car was left to drive around small_track for 20 laps. Knowing that each lap on average takes 11s to complete, this resulted in 129 new models saved. These models were then loaded and analysed with the validation dataset used in the previous section. Results are shown in Figure 4.9 hint that no further improvement occurs and in some cases, lap times are reduced in comparison to the same model without online learning.

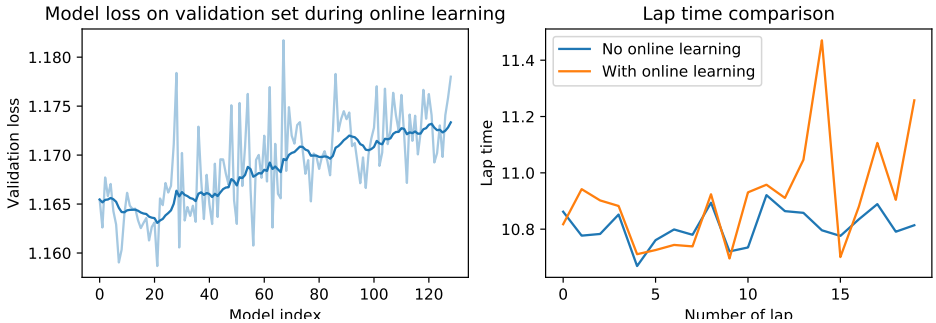


Figure 4.9: Performance of online learning. Left plot is smoothed.

The best reasoning for this behaviour is that the model continues to overfit as it is learning online; however, without a validation set to be evaluated on, the model can now overfit indefinitely. This behaviour is not the worst case of overfitting as the vehicle is always exploring different state space. However, this results in oscillating and unpredictable behaviour, which is undesirable. No solution to this problem that can run in real-time without affecting the performance of the algorithm was found; thus online learning was abandoned as an idea.

Chapter 5

Cost

Referring back to Figure 3.1, the costs are an essential part of the MPPI algorithm. This chapter will focus on optimising the cost for this project; however, unlike the previous chapter, here the focus will be on achieving a reliable working system, instead of exploring possible improvements of the basis of this project. The best model found in the previous chapter will be used throughout this chapter.

Interestingly, the cost functions reported in all of the MPPI papers does not match the source code available in the online repository. For this project, the cost function is based on the one found in the repository, which takes the shape of:

$$Cost = \alpha_1 C_{track} + \alpha_2 C_{control} + \alpha_3 C_{speed} + C_{crash} \tag{5.1}$$

5.1 Costmap

The costmap as represented in [Williams et al., 2017] is a 2D grayscale image which has 0 value in the centre of the track which gradually increases to 1 at the edge of the track. The pixel values outside of the track are 100. As the image is a discrete data structure, it needs to be interpolated so that it fits in the continuous state space of the problem. Then the cost of the vehicle is computed as the average of track cost of the front of the car and the track cost at the rear of the car. Figure 5.2(c) shows an example.

To create a costmap for this project, the author reverse-engineered the costmaps available on the Autorally repository. A Python tool was implemented for loading different Formula Student tracks (either from CSV or SDF files) and generating costmaps given parameters.

Track plotting
From Section 3.4, CSV files which define the locations of cones within the track are available. Afterwards, midpoints are generated by iterating over every left (blue) cone and finding the closest yellow point to it; then a midpoint is generated by averaging their x and y coordinates.

Before making a complete costmap, it is first needed to interpolate the track boundaries since the cones are sparse. This can be done piece-wise but produces bad results; instead, it was necessary to smooth to track boundaries using the scipy spline library. However, for this to work the cones must be sequentially ordered, which is not the case by default. To ensure correct order, an iterative closest point ordering algorithm was developed (Algorithm 2). Afterwards, a cubic spline is piece-wise fit through all ordered points with a smoothing factor of $s = 1$. This procedure is also done for the midpoints of the track. The final plotted result is seen in Figure 5.1. It is evident why smoothing is required.

Algorithm 2: Iterative closest point ordering

Given: **X** sequence of N points x_i where $i = 1, 2, 3..N$
Initialise: empty array **Y** for already ordered points
for x_i *in* X **do**
    $p \leftarrow$ find_closest(x_i, X)
    Append p to Y
    Remove p from X
end

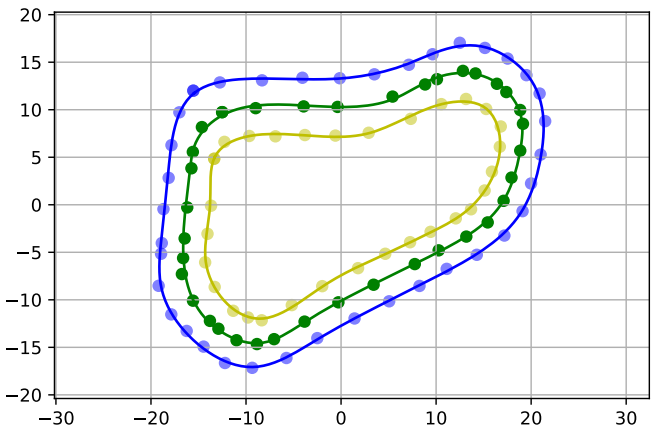


Figure 5.1: A plot of the interpolated track boundaries for small_track. Blue and yellow points represent cones; green points represent midpoints between the closest blue/yellow pairs. The lines are cubic splines fit through the points.

Costmap generation

The costmap must be saved in the correct format so that it is accepted in the Autorally source code, which accepts an NPZ file[1] with the keys from Table 5.1.

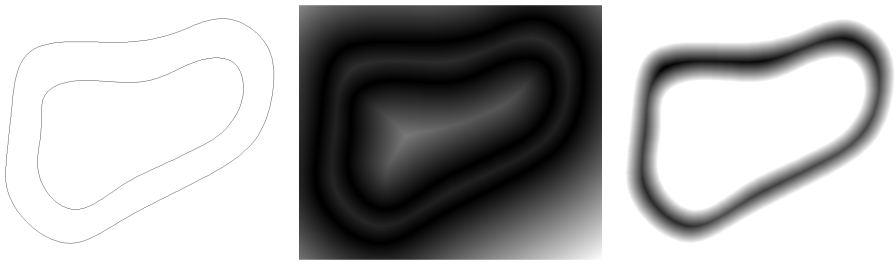
The process of creating the costmap is the following:

1. xBounds and yBounds are created by identifying the absolute maximum and minimum values along both x and y dimensions. Padding of 10 meters is also added to ensure that the track is always contained within the image.

[1] NPZ is a standard file for saving compressed data in NumPy. It has the structure of a disctionary

Key	Description	Example
xBounds	X limits of the track	[-25.0 10.0]
yBounds	Y limits of the track	[-15.0 20.0]
pixelsPerMeter	Number of pixels to be read per pixel	[20.0]
channel3	pixel values	[0. 0. 0. ... 0.]
channel2	pixel values	[0. 0. 0. ... 0.]
channel1	pixel values	[0. 0. 0. ... 0.]
channel0	pixel values	[0. 0. 0. ... 0.]

Table 5.1: The parameters and example values of a costmap used with the Autorally source code.



(a) Outline of track (b) Distance transformed (c) Final costmap

Figure 5.2: The process of creating a costmap

2. Since images have only positive values in their coordinates, the track boundaries and midline are shifted so that they are centred in the image coordinate frame. The track boundaries and midline are then scaled by the pixelsPerMeter parameter required for the costmap.

3. A blank white image with the correct dimensions is created.

4. The pixels corresponding to the track boundaries are set to 255. Figure 5.2(a).

5. A Huber distance transform[2] is then performed which results in smooth gradients extending outwards from both track boundaries. Figure 5.2(b).

6. Map the areas outside of the track and set them to 0.

7. Invert the image and normalise it in unit range [0, 1].

8. Set the areas outside of the track to pixel value 100. Costmap now finished - Figure 5.2(c).

Although the above process can be parameterised and optimised, that strategy is not the best as it requires offline changes to the costmap. On the other hand, such parameters can be embedded in the computation of the cost function during the MPPI optimisation.

[2] https://en.wikipedia.org/wiki/Huber_loss

Costmap calculation

Now that there is a costamp, it is finally possible to compute the cost of the car being in a certain position within the track C_{track}. In [Williams et al., 2017] this is done by calculating the value of the costmap under the centre of mass of the vehicle. This strategy did not produce good results in the case of this project - when the car was pressed to achieve a faster speed, it started diving into corners[3] and inevitably hits the cones of the track boundary. From the perspective of MPPI, this is completely valid behaviour as it is not completely aware of the dimensions of the vehicle.

To address this issue, the author altered the calculation of C_{track} to take into account the dimensions of the car. This is best explained by Figure 5.3. The parameter α_1 is left to be parametarised when MPPI is running.

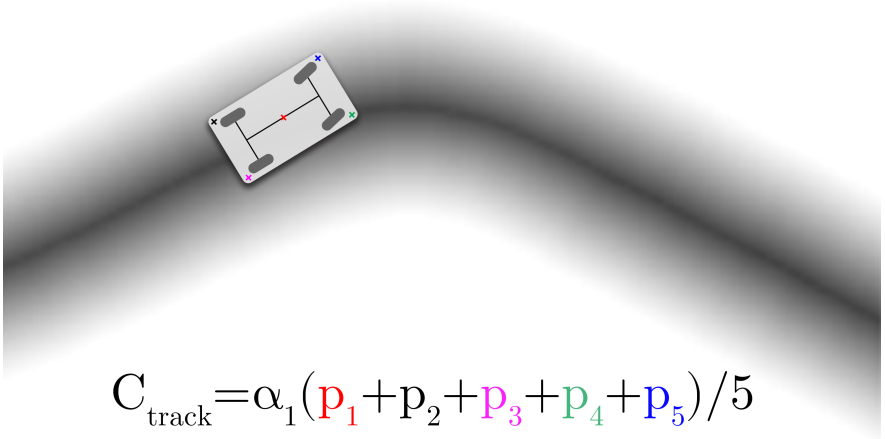


Figure 5.3: Dimensional calculation of track cost.

Experiments were run to compare the results of both track cost methods. Cost parameters used were $V_{des} = 12m/s$, $\alpha_1 = 300$, $\alpha_2 = 0$ and $\alpha_3 = 4.25$. The paths the car follows are seen in Figure 5.4. The dimensional track cost method produces more stable trajectories and thus better lap times. However, it is interesting that both methods have instabilities in different parts of the track.

5.2 Cost function

From Equation 5.1, $C_{control}$, C_{speed} and C_{crash} are yet to be defined.

Control cost $C_{control}$

[3] Diving into corners refers to getting in close proximity to the edge (apex) of a corner.

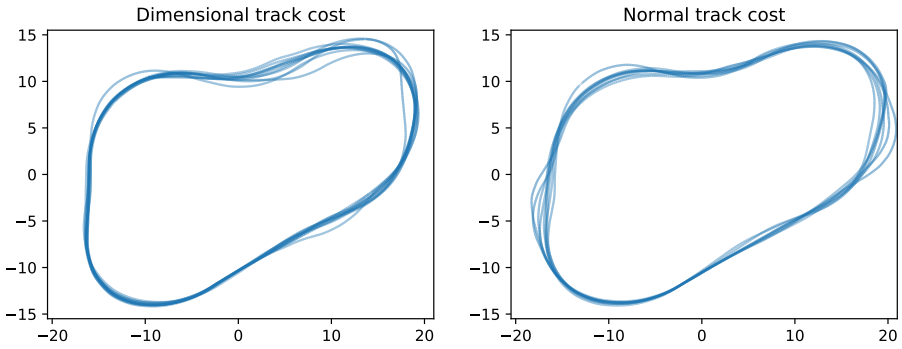


Figure 5.4: Traces of the paths that the car followed while running the MPPI controller with different methods of evaluating track costs. The more solid a line is, the more times, the vehicle has passed through that point.

Defined as in [Williams et al., 2017] as:

$$C_{control}(t) = \frac{\mathbf{u}(t) - \mathbf{u}(t-1)}{(\mathbf{v}(t) - \mathbf{u}(t))^2} \tag{5.2}$$

Increasing this cost will force the controller to change the control sequence less rapidly and as a result, stabilise trajectories.

Speed cost C_{speed}
One of the most critical parameters, this encompasses the target speed of the vehicle. The speed cost is defined as:

$$C_{speed} = (\dot{x}_v - V_{des})^2 \tag{5.3}$$

where V_{des} is the desired speed.

Crash cost C_{crash}
As noted in [Williams et al., 2017], as the unrolled trajectory is forward-sampled, the error of the predicted dynamics becomes more significant as the errors of each consecutive timestep stacks on top of the previous error - Figure 5.5

To tackle this issue, the original authors of MPPI introduced a *crash cost* which is also used in this project and is defined as:

$$C_{crash} = 0.9^t(10000I(C_{track} > 0.99)) \tag{5.4}$$

where $I(\cdot)$ is an impulse function and 0.9^t is the discount factor.

Cost parameter tuning
Parameter tuning for an algorithm such as MPPI is extremely difficult and particular to each different use case. Instead of searching for the *most optimal cost parameters*, it was chosen to minimise the number of tunable parameters and to provide tools for systematically evaluating performance.

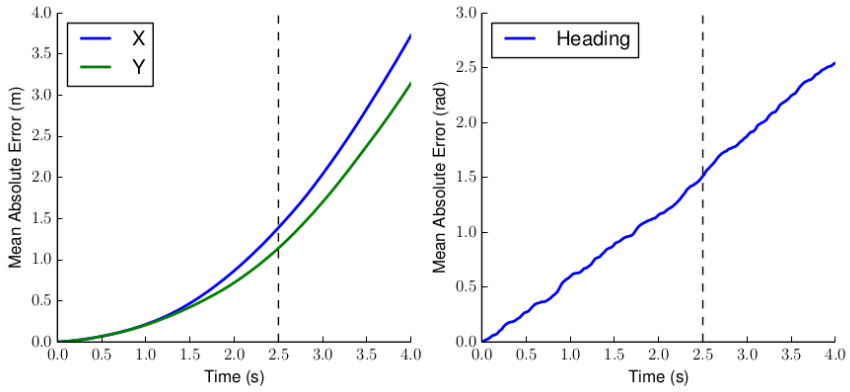


Figure 5.5: Prediction error for Autorally dynamics. the vertical bar denotes the prediction horizon. [Williams et al., 2017]

In comparison to the ten tunable parameters in the original work in [Williams et al., 2017], users of this project are given the option to tune only four parameters as seen in Table 5.2. It is possible to change these parameters through the ROS dynamic reconfigure tool[4], which allows for parameter tuning while the controller is operating on the vehicle.

Parameter	Name	Description	Default Value
V_{des}	Desired velocity	The target velocity the car should strive to achieve.	10
α_1	Track coefficient	Weight of how close the car should stay to the track.	300
α_2	Control coefficient	Higher values penalise the controller for rapidly changing actions.	0
α_3	Speed coefficient	Weight of how important it is to achieve the target velocity.	4

Table 5.2: Tunable cost function parameters.

An additional tool was developed to track lap statistics such as lap times, average speed, number of slips, number of occasions where the car went outside the track, as well as all parameters of the MPPI controller. These statistics are published as a ROS message and also stored in a YAML file for future reference. This tool comes as a ROS node called `lap_stats.py` within the controller package.

[4]Dynamic reconfigure is a tool within ROS which allows to change parameters in C++/CUDA code as it is being executed, instead of needing to recompile the code after each parameter change

5.3 Experiments

highlighting the effects of parameter changes. These are presented in Table 5.3 and Figure 5.6. The algorithm manages to surpass easily surpass the best lap time of 11.24 s done by the human expert used to generate the dynamics data. This should give the reader intuition regarding the complex relationships and trade-offs between the different cost parameters. The control cost parameter α_2 does not provide any perform acne improvements, thus it is not used in these experiments. Failed experiments are defined when the car hits a cone (track boundary) or simply goes outside of the track.

Interesting is the relationship between experiments #2 and #5 which achieve comparable performance but have different traces in Figure 5.6. This is related to the track cost as MPPI is more prone to go off-centre. A safe trajectory is also maintained in experiment #3 but that systematically fails in the same location. This is most likely due to the fact that the vehicle reaches not well-explored states (high speeds) and fails to plan a correct trajectory to tackle the next corner. In term, this highlights the issue with learning system dynamics with neural networks - they are more susceptible to never-before-seen data in comparison to classical system identification.

Finally, it is recommended to use the cost parameters of experiment #2 as it offers the best balance between reliability and performance.

ID	V_{des}	α_1	α_2	α_3	Lap time	Speed
#1	10	300	0	4	$11.18 \pm .07$	$9.14 \pm .05$
#2	13	300	0	4	$10.08 \pm .10$	$10.14 \pm .15$
#3	15	300	0	4	fail	
#4	15	400	0	4	$10.23 \pm .18$	$10.05 \pm .15$
#5	13	150	0	4	$9.9 \pm .14$	$10.28 \pm .14$
#6	13	150	0	5	fail	

Table 5.3: Experiments with different cost parameters. Lap times and speed is averaged over 5 laps. A failed experiment is defined when the car hits a traffic cone.

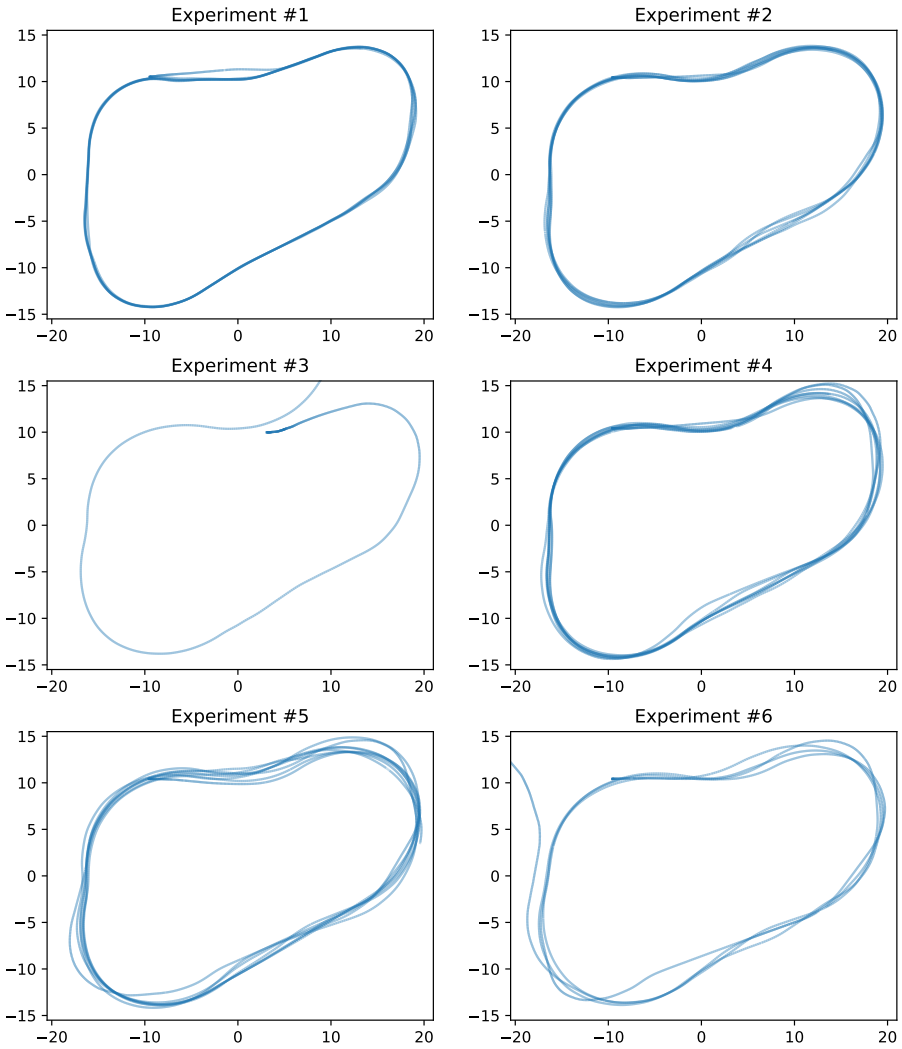


Figure 5.6: Traces of paths that the car has followed during cost parameter tuning. Lap traces consist of 5 normal laps + 1 warm-up one.

Chapter 6

Conclusions

6.1 Summary

The aim of this section is to summarise the work carried out during this project, the authors' contributions and achievements and to outline lessons learned.

Recent technological advancements have enabled the rise of autonomous cars, which are promising to alleviate traffic congestion, reduce road incidents and lower pollution. As a result, the popular student competition - Formula Student was extended to challenges students to develop, build and test an autonomous racecar. This project focuses on developing a path planning and control algorithm for such a vehicle, which is intended to be used by the Edinburgh University Formula Student (EUFS) team.

Initially, this project, surveyed the fields of path planning and control for Ackermann-type vehicles to find out that limited development has been made towards aggressive autonomous driving. The most suitable option chosen was the Model Predictive Path Integral (MPPI) algorithm, which unlike most other solutions, optimises trajectory and control inputs simultaneously in an open-loop. This is done exploiting a fundamental relationship between the information theoretic notations of free energy and relative entropy resulting in an optimal control solution which takes the form of an iterative update law which can be stochastically forward sampled in parallel on a GPU. The algorithm, first developed in [Williams et al., 2017], is then adapted for the purposes of this project and integrated into the autonomous driving solution of EUFS.

To facilitate the work of this project, a simulation has been developed, capable of accurately simulating vehicle dynamics, the Formula Student environment and tackling the necessary assumptions made during development.

Thereafter, the vehicle model needed for MPPI is explored with entirely data-driven machine learning methods. Sufficient state space for the task at hand is defined, during which it was shown that neural networks can be used in the development and definition of dynamics models. A framework is then developed for collecting and processing system identification data, which is extended with a fully-automated method of learning dynamics models and empirically evaluating them in the simulation. With the help of

this framework, the most suitable neural network dynamics model is identified, optimised for real-time computation. Additionally, the option of online learning of these dynamics is explored and implemented as part of the original MPPI algorithm, however, this also showcased the inherent limitations of the approach.

Finally, spatial information regarding the race tracks is embedded into the cost function of the algorithm in the form of a *costmap*, which is then integrated into the project and ready for real-world applications. The cost function of MPPI is then explored and simplified to facilitate easier cost parameter tuning and experimentation. To aid in evaluation, a set of tools have been developed for logging the performance, parameters and configurations of the algorithm as well as visualising its outputs. Using these tools a set of experiments have been run to evaluate the culmination of this project and give the reader understanding of how MPPI can be employed. A video of results of this project can be found online[1] and all the code and datasets used are hosted in a public repository[2].

In addition to the contributions above, the author would also like to highlight lessons learned throughout the duration of this project.

- Learning about optimal control theory, path integrals and the challenges to bringing theoretically-sound solutions to real implementations.
- Familiarisation with the industry standard robotics toolkit ROS and modelling accurate simulations with Gazebo
- Tailoring and comparing different machine learning experiments, while tackling their stochastic behaviour.
- Testing and evaluation performance of complex algorithms such as MPPI which do not have an easy to understand and intuitive goal. Further to that, the necessity of easy to use, robust and deterministic evaluation tools.
- The importance of time management and automation of trivial tasks. It was learned the hard way that throughout and carefully crafted experiments are more valuable than brute-force quantity orientated ones.

6.2 Discussion

In this report, MPPI was integrated into EUFS software stack and was shown that it is more than capable of aggressive racing in a *simulated* Formula Student environment. However, deploying the algorithm in the real world is a different challenge which breaks the assumptions made throughout this report. Namely 1) a SLAM algorithm exists which outputs 2D bird's eye cone locations and 2) accurate odometry estimation (localisation) is provided. Although care was taken while tackling these assumptions and even though noise was added in the simulation, that is still no guarantee for correct operation in the real world. In principle, it is possible to impose *minimal requirements*

[1] Video of MPPI in action - `https://youtu.be/xLGUA4ECP_I`

[2] Public repository of this project - `https://gitlab.com/eufs/it-mpc/tree/devel`

for 1) and 2) in terms of accuracy; however, those will be of nearly no value, since the result of this project is not a complete *preconfigured* solution.

The statement above brings up the next issue MPPI - it is difficult to configure and operate. The algorithm requires throughout understanding of the theory behind it in order to be able to operate it (and troubleshoot issues with it). Furthermore, it relies on two major building blocks - the model and the cost which will be different for every application. As such MPPI is not a *plug-and-play* algorithm, and its users should carefully consider both its benefits and drawbacks before committing to using it.

This report extensively explored an innovative alternative to classical system identification - learning the dynamics with a neural network, however, it was presented in a one-dimensional viewpoint. This method has one major limitation - it is significantly more susceptible to previously unexplored state space. An attempt to address this issue was presented in Chapter 4.4.4 by extending the algorithm to further learn its model online, however, that did not prove fruitful.

Throughout the report, several tools are developed for tuning and evaluating MPPI to achieve optimal performance. However, the same exact strategy of pushing the vehicle to its limits until it goes outside the track or flips over is not feasible in the real-world as it will incur risking damaging expensive hardware. Therefore, even when the algorithm is applied in real-life, the simulation developed in this report will prove to be a valuable asset. It allows for bootstrapping the vehicle model with data from simulation and then allowing the vehicle to explore and verify it in the real world. Furthermore, if the simulation proves accurate enough, it can be used to estimate the optimal cost function without having to risk damaging hardware.

6.3 Future Work

A look is finally taken at suggestions for further work that aims to possibly improve the results of this report and address some of the short-comings found.

MPPI is an innovative controller which can utilise neural networks in its dynamics model, however that topic is not sufficiently explored in the research community due to its novelty. As controls are executed sequentially through time, instead of using a standard fully-connected neural network to learn the dynamics as in Chapter 4, it is possible to use a Recurrent Neural Network (RNN) which is specialised towards time-series data. These networks have connections between nodes to form a directed graph along a temporal sequence. However, unlike traditional implementations, it would not be possible to predict a sequence of states, instead such a network will be limited to predicting only the next timestep $\mathbf{x}_{t+1}$ while having access to the current and previous states as seen in Figure 6.1. A further addition to this can be a novel hybrid approach to modelling the dynamics, where the well-understood part of the model can be designed analytically utilising classical *system identification approaches* and then the other non-linear parts which are difficult to analyse can be left for a neural network to learn. In theory, this will reap the benefits of both approaches.

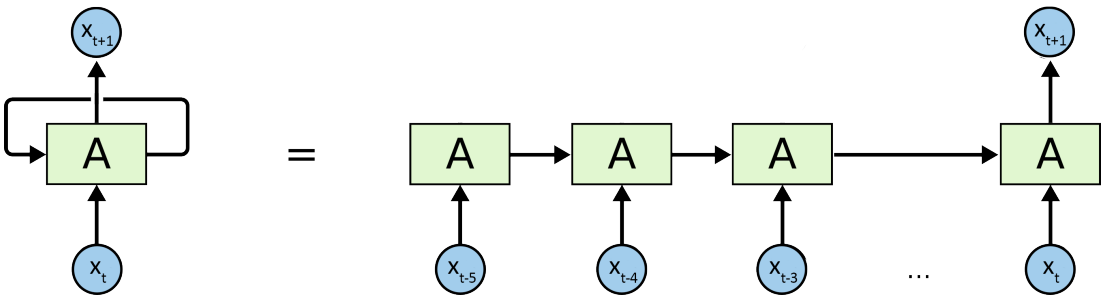


Figure 6.1: An example of how a RNN can be used for the predicting dynamic, while looking at the previous 6 timesteps.

Further to the idea above, it is possible to extend the idea of online learning as shown in Section 4.4.4 to work reliably and address the issues of neural network dynamics learning. An approach to this might be to represent the exploration of each state variable as a dense multidimensional histogram. Each bin would represent *how explored that state is* and if it not as explored as other states, then the model can learn from those particular interactions with the environment. This approach would deal with the overfitting problem found during online learning in Section 4.4.4, while ensuring that previously unseen states are handled correctly.

Throughout this paper, the goal has always been to predict only the next step of the algorithm. However, in principle, this is not the best method, as, in the end, the algorithm predicts several timesteps during a rollout. In theory, doing multi-step prediction would be more correct, however, that is not trivial, especially with the real-time requirements of this project. Approaches to solving this and potentially improving MPPI altogether is by utilising the Dataset Aggregation algorithm [Ross et al., 2011] or even better yet, multi-step predictions can be generated by RNNs and then optimised for multi-step error minimisation [Mohajerin and Waslander, 2019].

6.4 MInf Part 2

The focus of this report has been understanding and integrating MPPI into the EUFS autonomous racing solution. However, for part 2, I would like to contribute to the improvement and development of MPPI altogether by addressing some of its limitations.

In particular, I would like to improve the vehicle model of the algorithm and optimise it for better dynamics prediction, utilising the suggestions from the previous section and potentially generalising and automating it any type of robot.

However, due to the novelty of MPPI, there are many more opportunities for improvement which have not been highlighted in this report. Therefore the ideas explored here should not necessarily be considered the absolute goals for part 2 of the project.

Bibliography

[Aho and Hopcroft, 1974] Aho, A. V. and Hopcroft, J. E. (1974). *The design and analysis of computer algorithms*. Pearson Education India.

[Alonso et al., 2013] Alonso, L., Perez-Oria, J., Al-Hadithi, B. M., and Jimenez, A. (2013). Self-tuning pid controller for autonomous car tracking in urban traffic. In *System Theory, Control and Computing (ICSTCC), 2013 17th International Conference*, pages 15–20. IEEE.

[Automation, 2018] Automation, R. (2018). Rbcar simulation. `http://wiki.ros.org/rbcar_sim`. Accessed: 2019-04-04.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

[Dolgov et al., 2008] Dolgov, D., Thrun, S., Montemerlo, M., and Diebel, J. (2008). Practical search techniques in path planning for autonomous driving. *Ann Arbor*, 1001(48105):18–80.

[Dosovitskiy et al., 2017] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017). Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*.

[Fagnant and Kockelman, 2015] Fagnant, D. J. and Kockelman, K. (2015). Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167–181.

[Funke et al., 2012] Funke, J., Theodosis, P., Hindiyeh, R., Stanek, G., Kritatakirana, K., Gerdes, C., Langer, D., Hernandez, M., Müller-Bessler, B., and Huhnke, B. (2012). Up to the limits: Autonomous audi tts. In *Intelligent Vehicles Symposium (IV), 2012 IEEE*, pages 541–547. IEEE.

[Hart et al., 1968] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

[Hoppe, 1999] Hoppe, H. (1999). New quadric metric for simplifying meshes with appearance attributes. In *Proceedings Visualization'99 (Cat. No. 99CB37067)*, pages 59–510. IEEE.

[Kasprzak and Gentz, 2006] Kasprzak, E. M. and Gentz, D. (2006). The formula sae tire test consortium-tire testing and data handling. Technical report, SAE Technical Paper.

[Keivan and Sibley, 2013] Keivan, N. and Sibley, G. (2013). Realtime simulation-in-the-loop control for agile ground vehicles. In *Conference Towards Autonomous Robotic Systems*, pages 276–287. Springer.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Koenig and Howard, 2004] Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE.

[Kozlowski, 2012] Kozlowski, K. R. (2012). *Modelling and identification in robotics*. Springer Science & Business Media.

[Kullback and Leibler, 1951] Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86.

[Leonard et al., 2008] Leonard, J., How, J., Teller, S., Berger, M., Campbell, S., Fiore, G., Fletcher, L., Frazzoli, E., Huang, A., Karaman, S., et al. (2008). A perception-driven autonomous urban vehicle. *Journal of Field Robotics*, 25(10):727–774.

[Liniger et al., 2015] Liniger, A., Domahidi, A., and Morari, M. (2015). Optimization-based autonomous racing of 1: 43 scale rc cars. *Optimal Control Applications and Methods*, 36(5):628–647.

[Luo et al., 2016] Luo, Y., Bai, H., Hsu, D., and Lee, W. S. (2016). Importance sampling for online planning under uncertainty. *The International Journal of Robotics Research*, page 0278364918780322.

[Mohajerin and Waslander, 2019] Mohajerin, N. and Waslander, S. L. (2019). Multi-step prediction of dynamic systems with recurrent neural networks. *IEEE transactions on neural networks and learning systems*.

[Nilsson, 1969] Nilsson, N. J. (1969). A mobile automaton: An application of artificial intelligence techniques. Technical report, SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER.

[O'Connor, 2018] O'Connor, M. (2018). Formula student lap time simulator.

[O'Kane, 2014] O'Kane, J. M. (2014). A gentle introduction to ros.

[Pacejka, 2005] Pacejka, H. (2005). *Tire and vehicle dynamics*. Elsevier.

[Paden et al., 2016] Paden, B., Čáp, M., Yong, S. Z., Yershov, D., and Frazzoli, E. (2016). A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55.

[Pepy and Lambert, 2006] Pepy, R. and Lambert, A. (2006). Safe path planning in an uncertain-configuration space using rrt. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 5376–5381. IEEE.

[Pepy et al., 2006] Pepy, R., Lambert, A., and Mounier, H. (2006). Path planning using a dynamic vehicle model. In *Information and Communication Technologies, 2006. ICTTA'06. 2nd*, volume 1, pages 781–786. IEEE.

[Pinciroli et al., 2011] Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., et al. (2011). Argos: a modular, multi-engine simulator for heterogeneous swarm robotics. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5027–5034. IEEE.

[Renals, 2018] Renals, S. (2018). Lecture 6—RmsProp: Dropout, Initialisation, Normalisation. `https://www.inf.ed.ac.uk/teaching/courses/mlp/2018-19/mlp06-enc.pdf`. Accessed: 2019-04-04.

[Rohmer et al., 2013] Rohmer, E., Singh, S. P., and Freese, M. (2013). V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE.

[Ross et al., 2011] Ross, S., Gordon, G., and Bagnell, D. (2011). A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635.

[Shah et al., 2018] Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2018). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer.

[Siegwart et al., 2011] Siegwart, R., Nourbakhsh, I. R., Scaramuzza, D., and Arkin, R. C. (2011). *Introduction to autonomous mobile robots*. MIT press.

[Thrun, 2010] Thrun, S. (2010). Toward robotic cars. *Communications of the ACM*, 53(4):99–106.

[Thrun et al., 2006] Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., et al. (2006). Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning. Accessed: 2019-04-04.

[Tsiotras and Diaz, 2014] Tsiotras, P. and Diaz, R. S. (2014). Real-time near-optimal feedback control of aggressive vehicle maneuvers. In *Optimization and optimal control in automotive systems*, pages 109–129. Springer.

[Urmson et al., 2008] Urmson, C., Anhalt, J., Bagnell, D., Baker, C., Bittner, R., Clark, M., Dolan, J., Duggins, D., Galatali, T., Geyer, C., et al. (2008). Autonomous

driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466.

[Valls et al., 2018] Valls, M. d. l. I., Hendrikx, H. F. C., Reijgwart, V., Meier, F. V., Sa, I., Dubé, R., Gawel, A. R., Bürki, M., and Siegwart, R. (2018). Design of an autonomous racecar: Perception, state estimation and system integration. *arXiv preprint arXiv:1804.03252*.

[Velenis et al., 2007] Velenis, E., Tsiotras, P., and Lu, J. (2007). Modeling aggressive maneuvers on loose surfaces: The cases of trail-braking and pendulum-turn. In *Control Conference (ECC), 2007 European*, pages 1233–1240. IEEE.

[Williams, 2019] Williams, G. (2019). Autorally repository. `https://github.com/AutoRally/autorally`. Accessed: 2019-04-04.

[Williams et al., 2016] Williams, G., Drews, P., Goldfain, B., Rehg, J. M., and Theodorou, E. A. (2016). Aggressive driving with model predictive path integral control. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 1433–1440. IEEE.

[Williams et al., 2017] Williams, G., Wagener, N., Goldfain, B., Drews, P., Rehg, J. M., Boots, B., and Theodorou, E. A. (2017). Information theoretic mpc for model-based reinforcement learning. In *International Conference on Robotics and Automation (ICRA)*.

[Zhao et al., 2012] Zhao, P., Chen, J., Song, Y., Tao, X., Xu, T., and Mei, T. (2012). Design of a control system for an autonomous vehicle based on adaptive-pid. *International Journal of Advanced Robotic Systems*, 9(2):44.

Appendices

Appendix A

Formula Student Environment

The FS-AI competition consists of 3 major events to challenge the autonomous cars. To aid vision systems, all layouts are marked with blue traffic cones on the left side and yellow traffic cones on the right side. The events include:

- Skidpad - Figure A.1 - a track consisting of two pairs of concentric circles in a figure of eight pattern. The cars are required to make four laps in total around it - two in one direction and two in the opposite direction.
- Acceleration - Figure A.2 - a straight line with a length of 75m from starting to finish line. The track is at least 5m wide, and cones are placed at intervals of 5m.
- Trackdrive - Figure 2.1 - closed loop circuit of length between 200m and 500m. Cars are required to complete ten laps overall and are scored according to their average lap time. The autonomous cars have no prior knowledge about the layout of the track and thus must learn in during the initial laps. The track consists:
  - Straights: No longer than 80m.
  - Constant Turns: up to 50m diameter.
  - Hairpin Turns: Minimum of 9m outside diameter

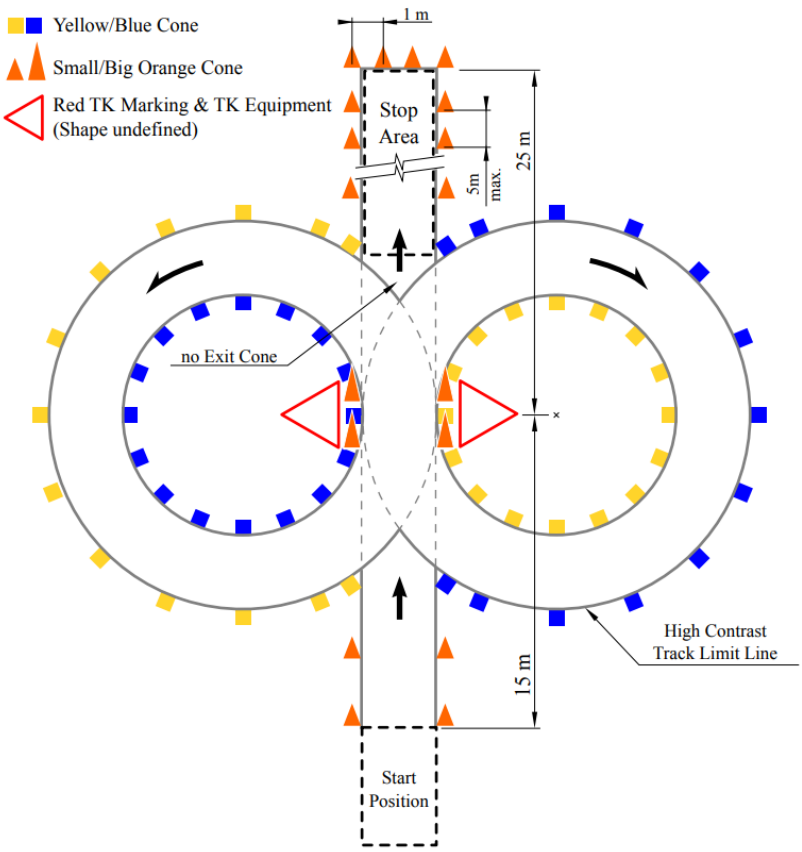


Figure A.1: Skidpad © Formula Student Germany

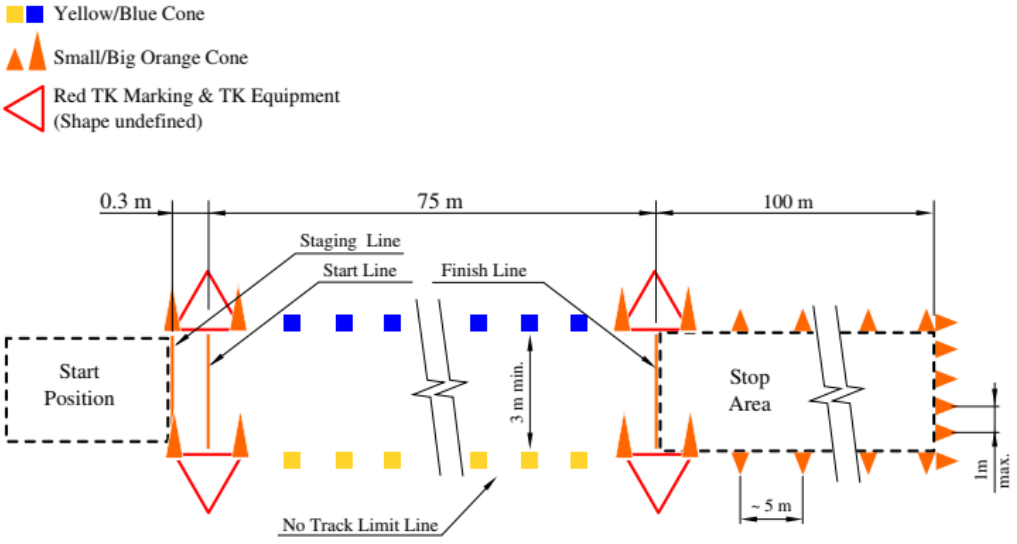


Figure A.2: Acceleration © Formula Student Germany

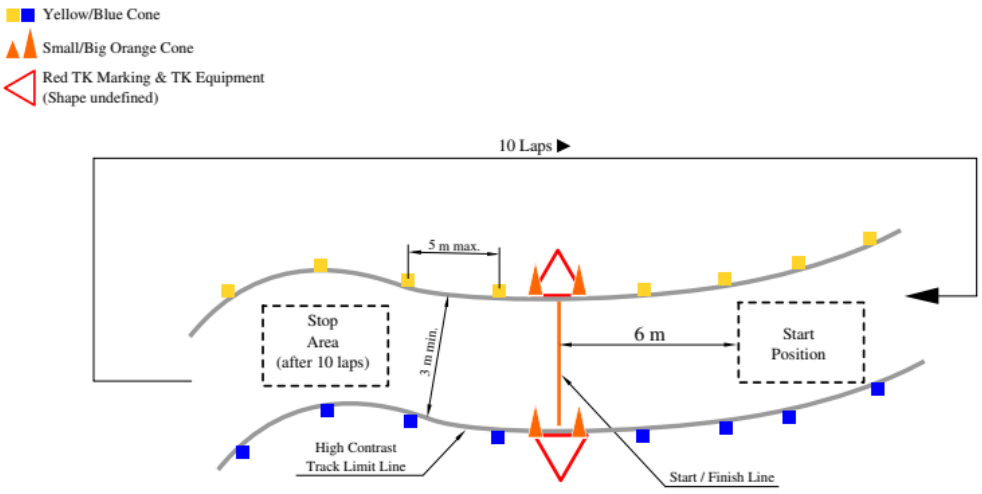


Figure A.3: Trackdrive © Formula Student Germany

Appendix B

ADS-DV Sensors and Computing

The ADS-DV vehicle is equipped with:

- Wheel encoders - track the rotation of each of the four wheels.
- Steering encoder - tracks the steering angle of the vehicle.
- INS - Ellipse 2N - an IMU and a GPS fused with a Kalman Filter outputting data at a rate of up to 400 Hz.
- Stereo camera - ZED Camera - a cost-effective camera system capable of perceiving depth from 0.5m to 20m.
- Lidar - Velodyne VLP-16 - 16 beam multi-planar Lidar capable of creating a 360° view of its surroundings up to a distance of 100m at a rate of 20 Hz
- NVidia Drive PX2 - a mobile computing equipped with 2x ARM Cortex-A57 CPUs, 2x Tegra Pascal GPUs and 64GB RAM. Aimed at parallel deep learning computation, capable of processing 16 FP16 TFLOPS
- NVidia Jetson TX2 - a small 10W system aimed for small-scale parallelisable computation, equipped with NVidia Pascal GPU with 256 CUDA cores, ARM Cortex A57 CPU and 8GB of RAM.

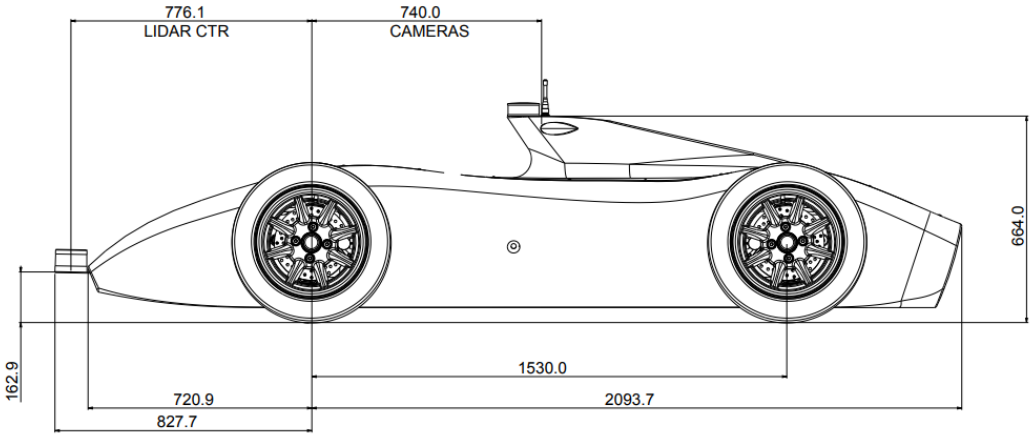


Figure B.1: Schematic side view of the ADS-DV. © IMechE

Appendix C

Neural network experiments

In Section 4.3, a neural network was used to identify the best possible state variables. The full details of those experiments are:

- 2 fully-connected hidden layers with 128 neurons with biases.
- tanh activation function.
- No normalisation.
- RMSProp optimiser with learning rate 10^{-4}, $\alpha = 0.99$, $\varepsilon = 10^{-8}$ and L2 weight normalisation $wd = 10^{-6}$ [Tieleman and Hinton, 2012].
- 500 epochs with batch size 100.
- 80% /10% /10% train/validation/test dataset split.

Here are the raw results from all NN experiments of Section 4.4 are shown. The results will be split into subsections according to their architecture. The name of each experiment was the timestamp when it was started. The learning rules use default parameters as described in Section 4.3. The best model is selected as the one with the best validation error. The lap time is evaluated as the 5 lap average on small_track (Section 3.4.1).

The MPPI cost parameters used for these tests are:

- $V_{des} = 10m/s$
- $\alpha_1 = 300$
- $\alpha_2 = 0$
- $\alpha_3 = 4.25$

Neural Network with 2 hidden layers of 8 neurons each

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
2019-02-16T 20:08:59.957713	0.005	Adam	TRUE	FALSE	178	1.38582623	13.855	Very unstable. Barely managed to complete laps.
2019-02-17T 04:54:34.460164	0.005	RMSProp	TRUE	FALSE	897	1.362227798	fail	Went backwards. Trajectories were very unstable. Went outside the track.
2019-02-17T 14:17:17.631820	0.001	Adam	TRUE	FALSE	764	1.346910357	10.898875	WOW. This small network rocked. It was very smooth.
2019-02-18T 01:02:42.404401	0.001	RMSProp	TRUE	FALSE	953	1.355379105	10.81366667	WOW. This was great as well.
2019-02-18T 12:09:41.935351	0.0005	Adam	TRUE	FALSE	527	1.349972248	11.189875	Went back.
2019-02-19T 00:13:54.100482	0.0005	RMSProp	TRUE	FALSE	983	1.385073066	fail	It didn't like corners and overshoot them

2019-02-19T 12:54:48.756392	0.0001	Adam	TRUE	FALSE	997	1.38134253	11.411	Pretty stable overall.
2019-02-20T 02:34:30.465528	0.0001	RMSProp	TRUE	FALSE	818	1.412572026	11.0268	Wen backwards. Was pretty damn aggressive and loved sliding. Trajectories were a bit unstable.
2019-02-20T 16:51:55.747314	0.005	Adam	FALSE	FALSE	869	1.444589138	10.9394	This was pretty stable.
2019-02-21T 08:56:22.288959	0.005	RMSProp	FALSE	FALSE	762	1.522273898	11.634	Trajectory unstable. Usually overshoots corners.
2019-02-22T 01:10:42.632965	0.001	Adam	FALSE	FALSE	932	1.458914876	11.1514	Quite stable and drifty. I like it.
2019-02-22T 18:00:02.831160	0.001	RMSProp	FALSE	FALSE	382	1.492998123	fail	Trajectories not very smooth. Overshoots corners a lot. Hit a lot of cones
2019-02-23T 11:27:14.027688	0.0005	Adam	FALSE	FALSE	950	1.485577226	11.36683333	Overshoots corners. Likes sliding

2019-02-24T 05:50:14.765479	0.0005	RMSProp	FALSE	FALSE	967	1.529542089	11.61425	Overshoots corners but always recovers
2019-02-25T 01:10:07.242527	0.0001	Adam	FALSE	FALSE	995	1.620259523	10.9022	Quite smooth and stable. Very surprising considering the high validation error
2019-02-25T 21:17:59.186151	0.0001	RMSProp	FALSE	FALSE	998	1.560499907	11.36775	Smooth and stable. sometimes overshoots corners

Neural Network with 2 hidden layers of 16 neurons each

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
2019-02-16T 20:48:04.694342	0.005	Adam	TRUE	FALSE	383	1.228810191	fail	The car just went backwards all the way??
2019-02-17T 05:36:38.962345	0.005	RMSProp	TRUE	FALSE	874	1.286914229	fail	
2019-02-17T 15:04:28.806350	0.001	Adam	TRUE	FALSE	910	1.215444088	fail	

2019-02-18T 01:53:54.323227	0.001	RMSProp	TRUE	FALSE	990	1.170702219	fail	
2019-02-18T 13:05:53.925774	0.0005	Adam	TRUE	FALSE	985	1.214161873	10.98183333	
2019-02-19T 01:13:15.628010	0.0005	RMSProp	TRUE	FALSE	964	1.218294382	10.9252	
2019-02-19T 13:58:45.452097	0.0001	Adam	TRUE	FALSE	989	1.303282857	11.31133333	Solid.
2019-02-20T 03:41:47.938707	0.0001	RMSProp	TRUE	FALSE	997	1.297703385	11.29533333	
2019-02-20T 18:03:53.614757	0.005	Adam	FALSE	FALSE	93	1.367636442	11.17066667	Drifty but stable.
2019-02-21T 10:14:14.759400	0.005	RMSProp	FALSE	FALSE	730	1.346057653	fail	Overshoots corners.
2019-02-22T 02:30:20.220834	0.001	Adam	FALSE	FALSE	825	1.341103315	11.01775	Overshoots; happy to slide.
2019-02-22T 19:23:51.379193	0.001	RMSProp	FALSE	FALSE	971	1.336584568	11.0076	
2019-02-23T 12:54:54.795323	0.0005	Adam	FALSE	FALSE	986	1.358161807	11.0708	
2019-02-24T 07:21:12.954987	0.0005	RMSProp	FALSE	FALSE	976	1.364845991	fail	

2019-02-25T 02:46:31.383939	0.0001	Adam	FALSE	FALSE	999	1.386052132	10.986
2019-02-25T 22:57:21.397268	0.0001	RMSProp	FALSE	FALSE	4	1.473080277	fail

Neural Network with 2 hidden layers of 32 neurons each

Name	**Learning rate**	**Learning rule**	**Normalisation**	**Dropout**	**Best epoch index**	**Best validation error**	**Average lap time**	**Comments**
2019-02-16T 21:27:21.098564	0.005	Adam	TRUE	FALSE	872	1.17107749	11.47733333	Relatively jerky trajectory further away in the future. Didn't want to start initially.
2019-02-17T 06:18:59.461141	0.005	RMSProp	TRUE	FALSE	130	1.173335075	11.03644444	Overall well. Still did the going backwards thing.
2019-02-17T 15:55:21.744302	0.001	Adam	TRUE	FALSE	246	1.115912318	11.74825	Overshoots corners and hits cones.
2019-02-18T 02:45:33.378016	0.001	RMSProp	TRUE	FALSE	939	1.136975408	11.31957143	Overshot the second corner quite badly. It was fine afterwards.

2019-02-18T 14:02:23.426135	0.0005	Adam	TRUE	FALSE	779	1.11740756	11.27466667	Unstable trajectory and overshoots corners.
2019-02-19T 02:12:52.856426	0.0005	RMSProp	TRUE	FALSE	400	1.161110282	11.27466667	Pretty stable and solid.
2019-02-19T 15:03:12.412580	0.0001	Adam	TRUE	FALSE	976	1.218885541	11.07266667	Overshot first corner on first lap.
2019-02-20T 04:49:29.754045	0.0001	RMSProp	TRUE	FALSE	974	1.197523594	10.88683333	Very stable.
2019-02-22T 20:47:34.849892	0.001	RMSProp	FALSE	FALSE	325	1.251967311	11.07233333	Pretty damn solid. It is very consistent and manages to predict dynamics by itself. It overshoots corners quite a lot but then again always recovers without hitting cones.
2019-02-23T 14:22:51.890805	0.0005	Adam	FALSE	FALSE	785	1.239231467	10.9706	Pretty solid.
2019-02-24T 08:52:25.052218	0.0005	RMSProp	FALSE	FALSE	999	1.227626324	11.54425	Almost went outside.
2019-02-25T 04:23:19.582385	0.0001	Adam	FALSE	FALSE	983	1.352055073	10.891	

2019-02-26T 00:37:06.488581	0.0001	RMSProp	FALSE	FALSE	996	1.315896749	10.966	

Neural Network with 2 hidden layers of 48 neurons each

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
2019-02-16T 22:07:10.078061	0.005	Adam	TRUE	FALSE	87	1.173938751	11.94675	Significantly more confident and capable in sliding. Worth noting that it overfits very swiftly
2019-02-17T 07:01:38.887076	0.005	RMSProp	TRUE	FALSE	143	1.186850429	12.4475	Always overshoots the first corner. One slow track because it hit cones
2019-02-17T 17:01:11.095973	0.001	Adam	TRUE	FALSE	390	1.126226068	12.0094	Always overshoots the first corner. One slow track because it hit cones
2019-02-18T 03:37:30.428864	0.001	RMSProp	TRUE	FALSE	421	1.101166487	11.2782	Pretty good. Always within the track

2019-02-18T 14:59:12.601841	0.0005	Adam	TRUE	FALSE	337	1.118475437	11.53185714	Overshot first corner
2019-02-19T 03:12:48.544229	0.0005	RMSProp	TRUE	FALSE	278	1.102432132	11.159	
2019-02-19T 16:07:45.383980	0.0001	Adam	TRUE	FALSE	986	1.153297663	11.50183333	Overshot first corner
2019-02-20T 05:57:26.263482	0.0001	RMSProp	TRUE	FALSE	977	1.158667326	10.85	
2019-02-20T 20:30:53.886824	0.005	Adam	FALSE	FALSE	179	1.246657252	11.28771429	
2019-02-21T 12:50:53.687846	0.005	RMSProp	FALSE	FALSE	257	1.271978378	11.24925	It learned to cool drift around the 1st corner
2019-02-22T 05:10:37.483340	0.001	Adam	FALSE	FALSE	797	1.175311208	fail	Went outside of the track on the 3rd lap. Did a cool final moment rescue drift
2019-02-22T 22:11:15.739796	0.001	RMSProp	FALSE	FALSE	496	1.19097662	11.071	
2019-02-23T 15:51:07.220968	0.0005	Adam	FALSE	FALSE	998	1.22126627	11.85975	Unstable; overshoots

Neural Network with 2 hidden layers of 64 neurons each

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
2019-02-16T 22:47:20.976826	98304	0.005	Adam	TRUE	FALSE	100	11.51	Drives on the outside of the track but hasn't hit the cboundaries
2019-02-17T 07:45:01.053057	98304	0.005	RMSProp	TRUE	FALSE	49	11.81766667	
2019-02-17T 17:50:15.838240	98304	0.001	Adam	TRUE	FALSE	132	11.7326	
2019-02-18T 04:29:51.144209	98304	0.001	RMSProp	TRUE	FALSE	153	12.151	Kind-a unstable. Overshoots corners
2019-02-18T 15:56:22.293874	98304	0.0005	Adam	TRUE	FALSE	424	12.63366667	Almost went out of the track
2019-02-19T 04:13:04.911001	98304	0.0005	RMSProp	TRUE	FALSE	262	11.13825	Good
2019-02-19T 17:12:47.333057	98304	0.0001	Adam	TRUE	FALSE	952	11.1825	Not much drifting but looks very efficient
2019-02-19T 17:12:47.333057	98304	0.0001	RMSProp	TRUE	FALSE	994	11.49033333	Overshoots a bit
2019-02-20T 21:57:52.905511	98304	0.005	Adam	FALSE	FALSE	319	11.317	

2019-02-21T 14:09:37.295583	98304	0.005	RMSProp	FALSE	FALSE	215	fail	overshot
2019-02-22T 06:31:20.784286	98304	0.001	Adam	FALSE	FALSE	260	11.4302	overshot
2019-02-22T 23:35:05.671679	98304	0.001	RMSProp	FALSE	FALSE	508	11.091	
2019-02-23T 17:19:37.951105	98304	0.0005	Adam	FALSE	FALSE	925	11.5506	
2019-02-24T 11:55:59.924841	98304	0.0005	RMSProp	FALSE	FALSE	662	11.4425	
2019-02-25T 07:38:05.452944	98304	0.0001	Adam	FALSE	FALSE	987	11.0772	

Neural Network with 3 hidden layers of 8 neurons each

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
2019-02-16T 23:27:59.189439	0.005	Adam	TRUE	FALSE	874	1.30386436	fail	Hmm this refuses to run for some reason. It just wants to go backwards indefinately. Investigate further!

2019-02-17T 08:28:28.456834	0.005	RMSProp	TRUE	FALSE	935	1.36380744	11.40933333	
2019-02-17T 08:28:28.456834	0.001	Adam	TRUE	FALSE	942	1.323821425	11.30016667	
2019-02-18T 05:22:28.078102	0.001	RMSProp	TRUE	FALSE	897	1.313225746	14.8705	Didn't do well at all
2019-02-18T 16:53:50.740923	0.005	Adam	TRUE	FALSE	966	1.324402571	11.8675	
2019-02-19T 05:13:35.705911	0.005	RMSProp	TRUE	FALSE	523	1.315394521	10.979	
2019-02-19T 18:18:08.462321	0.0001	Adam	TRUE	FALSE	999	1.378853083	11	
2019-02-20T 08:14:14.880964	0.0001	RMSProp	TRUE	FALSE	997	1.367005825	11.021	
2019-02-20T 23:21:06.452954	0.005	Adam	FALSE	FALSE	859	1.42032814	10.7205	
2019-02-21T 15:28:39.419489	0.005	RMSProp	FALSE	FALSE	869	1.45261991	failed	got stuck in corners?
2019-02-22T 07:52:18.252642	0.001	Adam	FALSE	FALSE	974	1.457005501	11.93566667	overshoots hard; wiggly trajectories
2019-02-23T 00:59:16.341257	0.001	RMSProp	FALSE	FALSE	954	1.419065952	12.504	went backwards first

2019-02-23T 18:48:27.747379	0.0005	Adam	FALSE	FALSE	989	1.420649648	fail
2019-02-24T 13:28:15.897294	0.0005	RMSProp	FALSE	FALSE	855	1.473297238	fail
2019-02-25T 09:15:58.638387	0.0001	Adam	FALSE	FALSE	993	1.517655134	11.11916667
2019-02-26T 05:38:38.081908	0.0001	RMSProp	FALSE	FALSE	997	1.53807044	11.24825

Neural Network with 3 hidden layers of 8, 16, 8 neurons each respectively

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
2019-02-17T 00:13:41.534454	0.005	Adam	TRUE	FALSE	387	1.293500662	12.65028571	Was quite hesitant to start but run fairly smoothly once started
2019-02-17T 09:17:06.750574	0.005	RMSProp	TRUE	FALSE	96	1.302935958	fail	overshoots corners gracefully (i.e. stops)
2019-02-17T 19:31:52.763615	0.001	Adam	TRUE	FALSE	959	1.263950586	11.4195	
2019-02-18T 06:19:38.292823	0.001	RMSProp	TRUE	FALSE	798	1.240846515	11.27128571	

2019-02-18T 17:55:43.994697	0.005	Adam	TRUE	FALSE	981	1.256322384	fail	started going backwards; fails because it is hesitant around corners
2019-02-19T 06:18:30.664719	0.005	RMSProp	TRUE	FALSE	960	1.267922878	11.175	overshots, failed final lap
2019-02-19T 19:27:59.265716	0.0001	Adam	TRUE	FALSE	986	1.305367827	11.69025	
2019-02-20T 09:27:04.714057	0.0001	RMSProp	TRUE	FALSE	990	1.373493671	11.199	
2019-02-21T 00:42:52.712404	0.005	Adam	FALSE	FALSE	541	1.311275959	11.50216667	hesitatnt around corners
2019-02-21T 16:52:12.219855	0.005	RMSProp	FALSE	FALSE	641	1.375123739	10.9106	
2019-02-22T 09:17:57.956053	0.001	Adam	FALSE	FALSE	900	1.329272032	fail	unstable trajectory
2019-02-23T 02:27:50.565919	0.001	RMSProp	FALSE	FALSE	988	1.343602777	11.9268	went backwards
2019-02-23T 20:22:00.785667	0.0005	Adam	FALSE	FALSE	988	1.358335614	fail	

2019-02-24T 15:04:15.749005	0.0005	RMSProp	FALSE	FALSE	929	1.391509414	11.37225	
2019-02-25T 10:58:32.182780	0.0001	Adam	FALSE	FALSE	983	1.443559885	11.642	Kind-of does the weigh shifting but it's very unstable
2019-02-26T 07:23:43.016182	0.0001	RMSProp	FALSE	FALSE	986	1.452852011	11.12016667	

Neural Network with 3 hidden layers of 8, 16, 16 neurons each respectively

Name	**Learning rate**	**Learning rule**	**Normalisation**	**Dropout**	**Best epoch index**	**Best validation error**	**Average lap time**	**Comments**
2019-02-17T 00:59:54.354921	0.005	Adam	TRUE	FALSE	547	1.235517025	fail	Fails to copmlete laps. Starts off well but trajectory is very jerky and someimtes also wants to go backwards
2019-02-17T 10:06:01.224962	0.005	RMSProp	TRUE	FALSE	418	1.224464655	fail	went backwards; went outside the track hard

2019-02-17T 20:25:36.332771	0.001	Adam	TRUE	FALSE	643	1.222990155	11.60371429	went backwards, but pretty crazy. Didn't hit any cones first 3 laps. Then went outside
2019-02-18T 07:17:08.734911	0.001	RMSProp	TRUE	FALSE	994	1.20611465	11.25866667	
2019-02-18T 18:58:01.422825	0.005	Adam	TRUE	FALSE	999	1.23376441	fail	Goes outside after the first corner
2019-02-19T 07:23:43.218177	0.005	RMSProp	TRUE	FALSE	987	1.230389476	11.1764	Pretty good
2019-02-19T 20:38:06.131867	0.0001	Adam	TRUE	FALSE	991	1.279715061	11.413	Went outside track on 3rd lap
2019-02-20T 10:40:11.338615	0.0001	RMSProp	TRUE	FALSE	964	1.316835642	11.38166667	
2019-02-21T 02:04:22.661055	0.005	Adam	FALSE	FALSE	462	1.349086404	11.9445	only 2 laps
2019-02-21T 18:16:15.700100	0.005	RMSProp	FALSE	FALSE	400	1.374710679	11.16125	
2019-02-22T 10:44:01.272384	0.001	Adam	FALSE	FALSE	894	1.301314116	11.6178	

2019-02-23T 03:56:51.080720	0.001	RMSProp	FALSE	FALSE	595	1.322384715	fail	went backwards; went outside on first corner
2019-02-23T 21:55:54.605137	0.0005	Adam	FALSE	FALSE	987	1.267397285	10.90133333	
2019-02-24T 16:54:40.208216	0.0005	RMSProp	FALSE	FALSE	986	1.291109681	11.38733333	
2019-02-25T 12:41:25.402917	0.0001	Adam	FALSE	FALSE	993	1.376596332	10.7806	went backwards; afterwards pretty consistent
2019-02-26T 09:09:17.963565	0.0001	RMSProp	FALSE	FALSE	995	1.43926549	11.46214286	

Neural Network with 3 hidden layers of 16 neurons each

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
2019-02-17T 01:46:11.781761	0.005	Adam	TRUE	FALSE	689	1.240614295	11.73733333	This one if a paradox. The trajectory is very unstable but in a different way to before. The overall end goal position is pretty constant, but the path to it changes. Even with this, it's lap times are one of the most stable ones seen thus far.
2019-02-17T 10:55:07.138046	0.005	RMSProp	TRUE	FALSE	160	1.234059811	11.4206	overshot corners slightly
2019-02-18T 08:14:56.074454	0.001	Adam	TRUE	FALSE	492	1.175912738	11.01366667	looks pretty slick
2019-02-18T 20:00:35.005333	0.0005	Adam	TRUE	FALSE	974	1.217239141	11.21428571	Slidy
2019-02-19T 08:29:17.035040	0.0005	RMSProp	TRUE	FALSE	777	1.20420897	fail	went overboard, fast
2019-02-19T 21:48:42.446901	0.0001	Adam	TRUE	FALSE	982	1.248136044	11.1415	Pretty stable

2019-02-20T 11:53:49.621846	0.0001	RMSProp	TRUE	FALSE	965	1.227286935	11.20475
2019-02-21T 03:26:12.917727	0.005	Adam	FALSE	FALSE	367	1.31892705	fail
2019-02-21T 19:40:46.492096	0.005	RMSProp	FALSE	FALSE	91	1.36363709	fail
2019-02-22T 12:10:19.588625	0.001	Adam	FALSE	FALSE	710	1.302651882	11.01975
2019-02-23T 05:26:23.349224	0.001	RMSProp	FALSE	FALSE	898	1.28401792	11.559
2019-02-23T 23:30:07.785842	0.0005	Adam	FALSE	FALSE	785	1.279577494	11.137
2019-02-24T 18:34:56.744696	0.0005	RMSProp	FALSE	FALSE	932	1.286406398	11.27133333
2019-02-25T 14:23:55.141839	0.0001	Adam	FALSE	FALSE	981	1.346297622	11.04575
2019-02-26T 11:03:57.239833	0.0001	RMSProp	FALSE	FALSE	993	1.373029113	11.0086

Neural Network with 3 hidden layers of 16, 32, 16 neurons each respectively

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
2019-02-17T 02:32:55.597030	0.005	Adam	TRUE	FALSE	625	1.18298912	11.19916667	Initially started planning backwards, but quickly recovered. Plans well but tends to do a snail trail at the end of the path
2019-02-17T 11:44:37.214774	0.005	RMSProp	TRUE	FALSE	512	1.182944179	11.71975	Unstable trajectory
2019-02-17T 22:15:54.293445	0.001	Adam	TRUE	FALSE	440	1.183642745	fail	Goes backwards
2019-02-18T 09:13:09.892217	0.001	RMSProp	TRUE	FALSE	331	1.124680638	fail	overshoots badly
2019-02-18T 21:03:23.127862	0.0005	Adam	TRUE	FALSE	609	1.147288442	12.9856	overshot once
2019-02-19T 09:35:04.851005	0.0005	RMSProp	TRUE	FALSE	940	1.147627354	11.1244	pretty good
2019-02-19T 22:59:29.040584	0.0001	Adam	TRUE	FALSE	988	1.19195807	11.13266667	
2019-02-20T 13:07:46.953479	0.0001	RMSProp	TRUE	FALSE	989	1.228970289	11.4632	failed to start correctly?

2019-02-21T 04:47:58.585569	0.005	Adam	FALSE	FALSE	148	1.26360786	fail	went outside at the end
2019-02-21T 21:02:57.907659	0.005	RMSProp	FALSE	FALSE	112	1.296214461	11.57425	a bit unstable
2019-02-22T 13:36:55.848769	0.001	Adam	FALSE	FALSE	809	1.265810251	11.31	
2019-02-23T 06:56:03.988476	0.001	RMSProp	FALSE	FALSE	344	1.288014054	fail	
2019-02-24T 01:04:31.166066	0.0005	Adam	FALSE	FALSE	745	1.254714251	11.5935	overshoots
2019-02-24T 20:13:00.654535	0.0005	RMSProp	FALSE	FALSE	704	1.269054651	12.8445	went backwards; overshoots
2019-02-25T 16:06:43.023125	0.0001	Adam	FALSE	FALSE	998	1.33874321	10.88325	pretty stable
2019-02-26T 12:49:15.094620	0.0001	RMSProp	FALSE	FALSE	982	1.370982289	fail	

Neural Network with 3 hidden layers of 32 neurons each

Name	Learning rate	Learning rule	Normalisation	Dropout	Best epoch index	Best validation error	Average lap time	Comments
0.005	Adam	TRUE	FALSE	54	1.176932096	1.18298912	11.2235	Trajectory looks stable but hit 3 cones.
0.005	RMSProp	TRUE	FALSE	117	1.188133597	1.182944179	11.45325	
0.001	Adam	TRUE	FALSE	207	1.134605408	1.183642745	fail	overshot hard on the 1st corner
0.001	RMSProp	TRUE	FALSE	621	1.129894614	1.124680638	11.38075	
0.0005	Adam	TRUE	FALSE	445	1.103417635	1.147288442	11.17825	
0.0005	RMSProp	TRUE	FALSE	257	1.125794888	1.147627354	11.4388	
0.0001	Adam	TRUE	FALSE	959	1.15176034	1.19195807	fail	overshot first corner
0.0001	RMSProp	TRUE	FALSE	820	1.13157773	1.228970289	10.90733333	
0.005	Adam	FALSE	FALSE	92	1.312399387	1.26360786	11.566	overshot first corner but managed to get back
0.005	RMSProp	FALSE	FALSE	320	1.305542827	1.296214461	11.55533333	Not so good at keeping to the middle
0.001	Adam	FALSE	FALSE	234	1.223459721	1.265810251	fail	
0.001	RMSProp	FALSE	FALSE	721	1.211175799	1.288014054	11.592	unstable
0.0005	Adam	FALSE	FALSE	966	1.197242379	1.254714251	11.18333333	much more stable!
0.0005	RMSProp	FALSE	FALSE	478	1.240801692	1.269054651	11.749	overshoots hard

0.0001	Adam	FALSE	FALSE	951	1.28052187	1.33874321	11.218	went straight left?
0.0001	RMSProp	FALSE	FALSE	995	1.271024346	1.370982289	10.8104	went backwards