Adaptive motion control for an autonomous race car

Ignat Georgiev

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2020

Abstract

Controlling an autonomous vehicle near the limits of its physical capabilities is a challenging task which has the potential to contribute to road safety. This is possible with model-based control approaches which harness a representation of the vehicles' physical properties to compute the optimal controls. However, these approaches suffer from inaccurate modelling due to computational restrictions and changes in vehicle dynamics over time. This thesis explores adaptive motion control of a racecar using a Model Predictive Controller (MPC). We develop three different modelling techniques using classical and novel methods and compare their accuracy and generalisation capabilities. Then we develop an adaptive learning system which can learn the changes in vehicle dynamics and adapt to them during operation. We evaluate these results in simulation and show that the model successfully learns long-term changes and improves its performance and stability. To achieve the best possible results, we also extend the MPC to track its nominal trajectory between optimisation cycles and introduce a new control sampling procedure which allows exploring feasible controls under different state spaces better. Finally, we reformulate the optimisation objective for the best possible lap time. We show that it achieves comparable performance to classical offline methods while optimising trajectories online and adapting to changing dynamics.

Acknowledgements

First and foremost, I would like to thank my supervisor Michael Mistry for his continuous support and guidance. I thank Christoforos Chatzikomis and Timo Völkl for their help with this project and keeping me grounded in reality. I thank Joshua Smith for sharing his robot dynamics expertise. Finally, I thank my family and friends for their unconditional support.

Table of Contents

Glossary

EUFS Edinburgh University Formula Student. 9

GPU Graphics Processing Unit. 55, 56

LQR Linear Quadratic Regulator, a classic controller for various applications. 48, 49, 63

MPC Model Predictive Controller, a family of common control algorithms. 2, 10, 64

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, 10, 18, 19, 30, 33, 42, 44, 46, 48–51, 54, 55, 57–60, 62–65, 73

MSE Mean Square Error. 15, 30, 32, 39

PID Proportional Integral Derivative, a classic controller for various applications. 48

ROS The Robotics Operating System - the defacto standard for software system building in robotics. 7, 17, 55, 62, VII

Mathematical Notation

$\mathbf{x}_t$	Vehicle state at time t. This vector consists of both kinematic and dynamic state variables. Shorthand for $\mathbf{x}(t)$.
$\mathbf{x}^k$	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. Shorthand for $\mathbf{u}(t)$.
$\mathbf{v}_t$	Noisy vehicle control signals at time t. Shorthand for $\mathbf{v}(t)$.
κ	Curvature.
δ	Steering angle.
u_κ	Curvature command.
u_δ	Steering command.
u_a	Acceleration command.
F_x	Longitudinal tyre force.
F_y	Lateral tyre force.
β	Slip angle.
$f(\cdot)$	State transition function.
$\phi(\cdot)$	Terminal cost function.
$\mathcal{L}$	Quadratic cost term.
$[*]_v$	the v denotes that the variable is in the vehicle coordinate frame.
$[\dot{*}]$	Derivative of any variable $*$.
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.
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.

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.

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.

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.

Matlab - a multi-paradigm numerical computing environment and proprietary programming language

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.

RViz - a data visualisation tool which comes prepackaged with ROS.

Iterative Learning - a machine learning scenario where the model is continuously trained on new data when available without an explicit notion of validation or test sets.

Generative models] - unsupervised learning methods which aim to learn the true data distribution with the aim of afterwards generating new data points with some variation.

Chapter 1

Introduction

Robots are one of the most fascinating human-made machines. They raise hopes of exploring extraterrestrial planets, replacing human workers in dangerous environments, assisting us with laborious tasks, caring for the sick and even saving human lives [Siegwart et al., 2011]. Although there are real examples of all of these feats, robots are still not commonplace. This is because these complex machines involve the integration of many different bodies of knowledge - mechanical construction, actuation, battery development, dynamics, artificial intelligence, computer vision and endlessly many others. All of this combined makes robotics as interdisciplinary a field as there can be. By mastering robotic arms in recent decades, interest has now shifted towards a more interesting albeit more difficult topic - mobile autonomous robots.

One of the first instances of potential for massive adaption of this technology has been autonomous cars which are now promising to alleviate traffic congestion, reduce road incidents and fatalities, and lower pollution [Fagnant and Kockelman, 2015]. These robocars operate similarly to any other mobile autonomous robot. They have to perceive the environment, localise within it and plan their next actions based on those inputs. As such, the software behind an autonomous car can be classified into three different categories - perception, localisation and motion control, the so-called *see-think-act cycle* shown in Figure 1.1. The focus of this report is on the last stage - motion control.

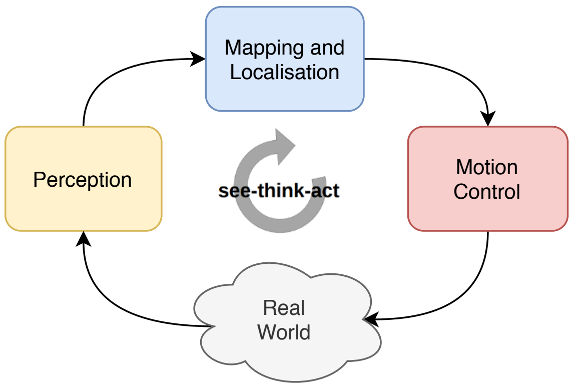


Figure 1.1: The see-think-act mobile robot cycle.

The goal of this project is to develop a motion control system which optimises trajectories online, adapts to changing dynamics and is ultimately capable of beating human lap time performance in an autonomous racing scenario. This is done in collaboration with Roborace - the world's first self-driving racing series, created to accelerate the development of self-driving vehicle technology. As such, the target vehicle for this project is the Roborace DevBot V2, as seen in Figure 1.2.

Figure 1.2: DevBot V2. Source: Roborace

This project is a direct continuation of the authors' previous work [Georgiev, 2019] in which the problem of autonomous racing was tackled using a novel Model Predictive Control (MPC) approach for trajectory optimisation. In classical literature, these types of controllers consist of a model which is used to forward simulate a trajectory which is then numerically optimised to find the optimal solution. Ideally, this optimisation should be carried out in real-time; however, that is impossible due to limited computation which forces the applications of these controllers to reduce model fidelity and the optimisation criteria [Lam et al., 2010] [Gao, 2014] [Kong et al., 2015]. Instead, we take a different approach and sample multiple possible trajectories using randomly perturb controls and cost-wise average them to obtain the optimal trajectory. This approach is known as Model Predictive Path Integral (MPPI) Control, first pioneered by [Williams et al., 2016].

1.1 Objectives and Contributions

To achieve the end goal of this project - develop a motion control system to beat human lap-times, this is into several sub-objectives:

1. Develop a standalone, easy-to-use simulator for rapid motion control development.
2. Review vehicle dynamic literature, research, implement and experiment with more generalisable and accurate vehicle models.
3. Research and apply iterative model learning and adaption.
4. Integrate and tune the algorithm for maximum performance.

Contributions

1. Packaging of a portable standalone simulator for motion control applications.
2. Integration of the MPPI controller into the target platform
3. Research, development and evaluation of a novel semi-parametric vehicle model with superior generalisation capabilities.
4. Development of an iterative learning system which adapts the model to changing dynamics online.
5. Development of a trajectory tracking module for the MPPI controller.
6. Introduction of dynamic control sampling based on the current state of the vehicle.
7. Reformulation of the controller cost and optimisation for minimum lap times.
8. CUDA and C++ close to real-time implementation of the above-stated work; ready to be applied to a real vehicle.

1.2 Report outline

The following chapters document the course of the project. Chapter 2 lays out the problem of motion control, the target environment for this project and summaries the base MPPI algorithm used throughout this project. Furthermore, a section of the chapter summarises our prior work in [Georgiev, 2019]. Chapter 3 outlines the developed simulation for this project and the overall integration in the target environment. Chapters 4,5 and 6 deal with parts of the MPPI algorithm which, when combined, produce the final product of this report. They tackle the topics of vehicle modelling (Chapter 4), trajectory optimisation (Chapter 5) and costs (Chapter 6); however, the results between them are incomparable due to the timing of the experiments. The overall performance of the algorithm is shown only in Chapter 6. Finally, Chapter 7 presents a summary of the work undertaken, the results obtained and further steps that can be taken towards refining path-integral control and iterative dynamics learning.

Chapter 2

Background and Related Work

In recent years there has been a significant rise of interest towards aggressive autonomous driving in cars. Controlling a vehicle to the limits of its capabilities is a challenging robotics problem which has the potential to contribute to the safety of road-legal driverless vehicles which have recently started emerging in the commercial field.

2.1 The Motion Control Problem

Fundamentally, the field of motion control asks the question of how can we control a system to achieve the desired goal. This goal can be of various abstractions and can be continuously changing. Examples in automotive include ”Drive to London”, ”Do a right turn”, ”Reach target position”, ”Achieve given velocity” or even ”Rotate motor at desired torque”. In this example ”Drive to London” is the core problem, and the following examples are issues that arise from it in a cascade fashion.

One can attempt to solve the problem with a so-called end-to-end algorithm which accepts the overall goal (”Drive to London”) and control the car directly to achieve it. This problem is usually considered in the field of General Artificial Intelligence or Machine Learning. There are many approaches which attempt to solve similar general problems, the most famous of which are Imitation Learning [Bojarski et al., 2016], [Chen and Huang, 2017] and Reinforcement Learning [Kendall et al., 2019], [Folkers et al., 2019] approaches. Although there are examples of successful applications, all of them are confined to the environments used to train them, require vast amounts of data and lack any explainability and as a result safety.

Instead of trying to solve the many problems of motion control simultaneously, we can dissect the overall goal into sub-problems and tackle all of them individually. Continuing the abstraction examples shown in the previous paragraph, we present an example motion control system in Figure 2.1. Note that this is by no means exhaustive or definite but only an example. Many more (or less) abstraction levels can be introduced depending on the task. However, dissecting a motion control system in this fashion has two key advantages: (1) the system is more explainable and (2) the hard motion control problems are simplified.

In the case of this project, the high-level intention of the vehicle is clear - achieve the best

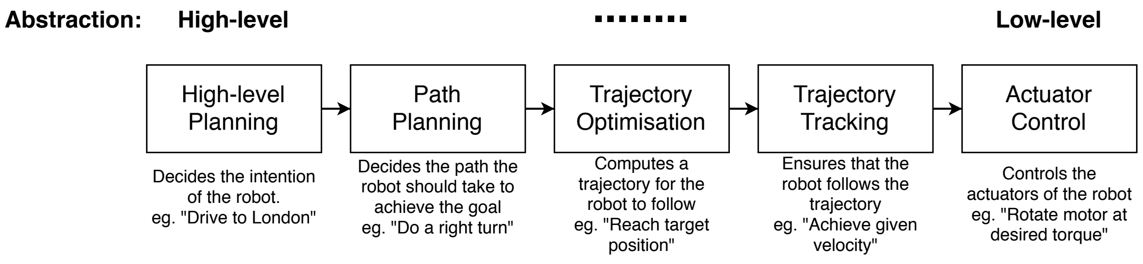

Abstraction: High-level
Low-level
High-level Planning
Path Planning
Trajectory Optimisation
Trajectory Tracking
Actuator Control
Decides the intention of the robot. eg. "Drive to London"
Decides the path the robot should take to achieve the goal eg. "Do a right turn"
Computes a trajectory for the robot to follow eg. "Reach target position"
Ensures that the robot follows the trajectory eg. "Achieve given velocity"
Controls the actuators of the robot eg. "Rotate motor at desired torque"

Figure 2.1: An example motion control stack dissected into five different abstraction levels. Each module decides the next action in a higher abstracted level and sends that as inputs to the next module, which has a lower level of abstraction.

possible lap times without crashing, and the low-level actuator control is assumed to be a working black box. Thus, we are left to solve problems in between which we chose to formalise path planning, trajectory optimisation and trajectory tracking (similar to Figure 2.1 above). Relevant here is the difference between a path and a trajectory. The first is a set geometric positions, and the latter is the full state of the vehicle[1] propagated thought time.

2.1.1 Formal definition

Now we can define the problem mathematically as an optimal control problem. Given a dynamical system described by the following differential equation

$$d\mathbf{x} = f(\mathbf{x}, \mathbf{u})dt \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state and $\mathbf{u} \in \mathbb{R}^m$ is the control. The function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ is the state transition function. The finite-horizon control problem is then defined as finding a sequence of states and a control policy π that minimise the cost function:

$$J(\mathbf{x}(t_0)) = \mathbb{E}_{\mathbf{x}}\left[\phi(\mathbf{x}(T)) + \int_{t_0}^{T} \mathcal{L}(\mathbf{x}(t), \pi(\mathbf{x}(t)))dt\right] \tag{2.2}$$

where $\phi : \mathbb{R}^n \to \mathbb{R}$ is the terminal cost function and $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}$ is the instantaneous cost. The control policy $\mathbf{u}(t) = \pi(\mathbf{x}(t), t)$ is a function which maps the state and time to a control vector. Finally, the cost $J(\mathbf{x}(t_0))$ for a starting state t_0 is defined as the expectation of the total cost accumulated from timestep t_0 to T. Now to find the optimal control solution, we simply have to minimise the cost subject to the controls:

$$\mathbf{u}^* = \underset{U}{\operatorname{argmin}}(J(\mathbf{x}(t_0))) \tag{2.3}$$

however, this minimisation is not trivial as we are to find out later.

[1]full state can refer to position, orientation, velocity, acceleration or any quantity describing the state of the car

To simplify notation and keep the mathematical formulation brief and concise for the remainder of this report, we will present derivatives as $\dot{\mathbf{x}} = d\mathbf{x}/dt$ and time-varying variables as $\mathbf{x}_t = \mathbf{x}(t)$. With this new simplified notation, we can rewrite Equations 2.1 - 2.3 to:

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) \tag{2.4}$$

$$J(\mathbf{x}_0) = \mathbb{E}_{\mathbf{x}}\left[\phi(\mathbf{x}_T) + \int_{t_0}^{T} \mathcal{L}(\mathbf{x}_t, \pi(\mathbf{x}_t))dt\right] \tag{2.5}$$

$$\mathbf{u}^* = \underset{U}{\operatorname{argmin}}(J(\mathbf{x}_0)) \tag{2.6}$$

2.2 Problem Definition

2.2.1 Environment

The target environments for this project are standard Formula 1 race circuits such as Monteblanco (Spain, Figure 2.2), Modena (Italy) and Silverstone (UK). These are circuits with high-quality tarmac, usually of width between 10-20 meters and total distances of 4-7 km. At such tracks, it is common for Formula 1 or comparable cars to reach speeds of more than 300 kph (83 m/s).

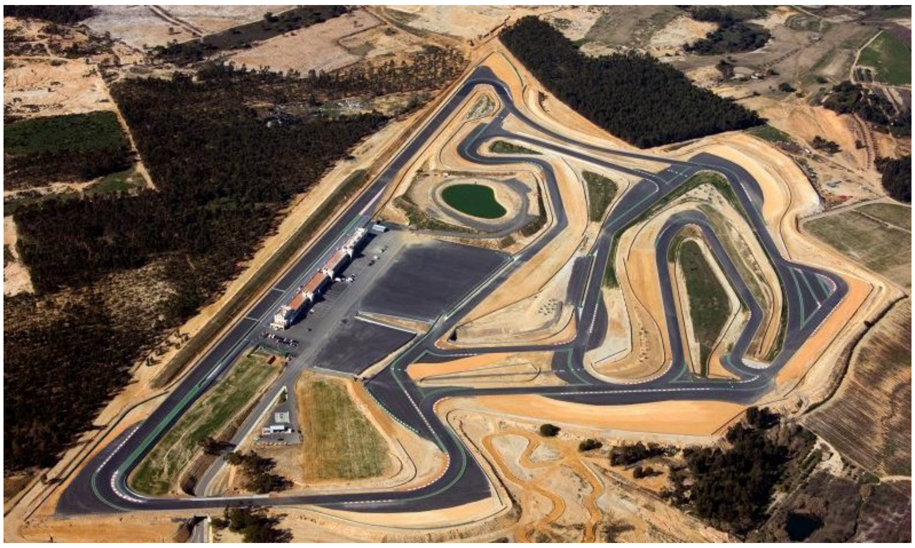


Figure 2.2: Monteblanco racing circuit.

2.2.2 Vehicle

The platform used for this project is an Ackermann-type vehicle, developed for easier autonomous software development instead of raw performance. As such, the vehicle is more orientated for ease-of-use and reliability. Its main parameters are shown in Table 2.1. Note that these vehicle parameters are in later chapters.

Parameter	Value
Wheelbase	2.9 m
Track front	1.54 m
Track rear	1.58 m
Overall width	2.0 m
Overall length	4.8 m
Max steering angle	0.48 rads
Car weight	1350 kg
Max. velocity	est. 240 km/h (66.67 m/s)
Max. lateral acceleration	13.734 m/s^2
Weight distribution	0.481% - 0.519% (Front - Rear)
Front axle to CoG	1.5 m
Rear axle to CoG	1.4 m
CoG height	0.275 m

Table 2.1: Vehcile Key Technical Specifications

Two inboard electric motors power the car, one on each rear wheel with 135 kW peak power. The batteries supplying this power are rated at 730V with a capacity of 32kWh. These batteries power all electrical systems on the computer. Two additional actuators are installed to enable autonomous control of the vehicle. An electric motor attached to the steering rack controls the steering, and an electrically controlled brake control unit activates the brakes of the car.

The car is equipped with the following sensors allowing it to perceive the environment and localise within it:

- 5 Lidars
- 8 ultrasonic sensors
- 6 monocular cameras
- 1 front-mounted radar
- 1 combined GPS and IMU sensor
- 1 slip-angle sensor
- 4 wheel speed sensors; one per wheel
- 1 steering displacement sensor

Additionally, the car sports three different computing platforms used for various applications. These are listed below, along with a supplementary visual guide in Figure 2.3.

1. Nvidia Drive PX2 - capable desktop computer, equipped with multi-core processors and a 512-core graphics card. This machine runs a non-real-time version of Linux with ROS and is generally used for running of C++ and Python programs.
2. Speedgoat - 2-core real-time capable computer running Simulink programs. Used to safety-critical systems in a real-time manner including driving support, vehicle models, stabilisation and path matching.
3. TAG 320 - Standard racing Engine Control Unit (ECU). This serves as the interface to the car providing access to the actuators of the vehicle and feeding back information from some sensors to the other computing systems.

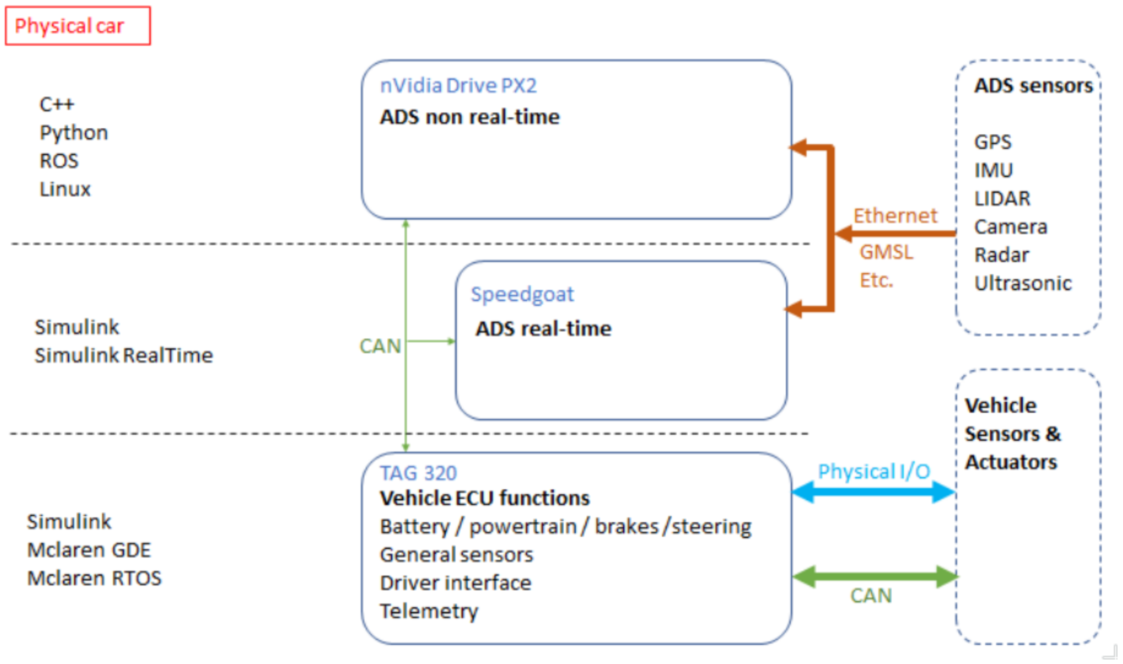


Figure 2.3: The compute architecture, along with vehicle communications and sensor communications. Both the Nvidia Drive PX2 and the Speedgoat computers obtain sensor information from sensors through UDP packets over Ethernet. Additionally, both of these computers communicate with each other via the same method. All three machines are connected on the same CAN network and can send vehicle commands to the ECU through that. The TAG 320 system communicates to vehicle-specific sensors and actuators through proprietary communication protocols and an independent CAN network.

2.2.3 Software Base and Assumptions

As this project focuses on the problem of motion control which is the last stage in the ”pipeline” of an autonomous car, it depends on several layers of software beforehand. As such, some assumptions must be made. First and most importantly, the development of this project builds on top of the existing autonomous driving stack developed by Roborace. It features a robust localisation system and a feedforward-feedback steering controller with an offline calculated trajectory. Based on that, we make several assumptions for this project:

1. The vehicle dynamics can be modelled in simulation with such a high accuracy that overall performance of the outcomes of this project will not be affected by the small difference between simulation and real-world. Although this assumption is ambitious at best, it is closer to truth given the accurate simulation software presented in Chapter 3 and any inaccuracies are assumed to be handled via the model adaption showed later in Section 4.7 when this work is applied to a real-world application.

2. The actuator delay for the motors, steering and braking is insignificant and would not affect performance. This assumption regarding the motors is proved by experiments run at Roborace, and the associated delay is lower than 10ms which is lower than the maximum running time of this project. However, this assumption does not hold for

the steering and braking actuators. Since there are no accurate modelling of these systems, zero actuator delay is assumed for simulation.

3. Accurate state estimation is given at 250 Hz using the sensors provided in the previous subsection. The accuracy of this system has not been verified; however, due to the high-quality sensors, this assumption is not unrealistic.
4. Map boundary, middle line and racing line coordinates are provided for each racing track. This is based off prior work by the team and is provided as a working framework for this project.

2.3 Summary of MInf Part 1

This report is part of our Master of Informatics (MInf) dissertation at the University of Edinburgh. This dissertation is carried over four semesters and is split into two parts. MInf Part 1 has successfully been carried out in the 2018/2019 academic year [Georgiev, 2019], and this report is about MInf Part 2. Since this report is a direct continuation of Part 1, this subsection provides a detailed review and discussion of our work.

The overall goal of Part 1 of our dissertation was similar to the one in this report - develop a path planning and control algorithm for an autonomous racecar. However, the environment for that was different as it was set in the world of Formula Student - a competition which challenges students to develop a driverless racecar. A significant contribution of MInf Part 1 was the development of a Gazebo-based simulation environment which served its purpose but has proven to be inaccurate by other members of Edinburgh University Formula Student Edinburgh University Formula Student (EUFS). For that reason and the drastic environmental changes between MInf Part 1 and Part 2, Chapter 3 of [Georgiev, 2019], which focuses on Simulation and Integration, will not be summarised here.

2.3.1 Core algorithm choice

In Section 2.3 of [Georgiev, 2019], we laid out prior work on path planning and control approaches to autonomous driving and reason our final choice of approach. In that section we start from the foundation of the field dating back to the DARPA Grand Challenge [Thrun et al., 2006] and review the whole field in its classical hierarchical approach (similar to Figure 2.1 shown previously). In most cases, the approach to autonomous driving is to have a high-level path planner which provides a plan of action and then we would have a motion controller which executes that plan. We review classical approaches to path planning such as Dijkstra's algorithm, A* and incremental search approaches [Paden et al., 2016]. Then we take a look at the control side and investigate the possibilities of instantaneous feedback controllers such as PID controllers and the more advanced family of Model Predictive Controllers (MPC), which can forward simulation vehicle motion.

We continue to reason our way that the traditional approach of the path planning and motion control split is flawed. 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. Furthermore, we reason that classical MPCs are operating at a slow rate (10-20 Hz) and are severely limited by simple models.

The approach that addresses all of these concerns is the Model Predictive Path Integral (MPPI) Controller developed by researches at Georgia Institute of Technology [Williams et al., 2017]. Unlike the approaches stated previously, this algorithm focuses on the Reinforcement Learning (RL) task of computing the most optimal trajectory and control sequence simultaneously in an open-loop by forward-sampling possible trajectories in parallel. Furthermore this approach is based on the path-integral framework which is derivative free, allowing the algorithm to use more complex and non-linear models than possible with traditional MPCs.

2.3.2 Model Predictive Path Integral (MPPI)

This subsection presents an adapted summary of the base algorithm used throughout this project. For a more complete description, the reader should refer to Section 2.4 of [Georgiev, 2019] or the original work in [Williams et al., 2017].

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 unexecuted portion of the previous control sequence as the importance sampling trajectory for the next iteration while utilising a dynamics transition model.

If given access to a perfect model, then doing a single 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 challenging 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 2.2 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 racecourse. The sampling process is visualised in Figure 2.4. 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 2.4 shows a simplified example. In practice, it is needed to sample significantly more than ten trajectories in order to find the optimal one consistently.

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
$(\mathbf{u}_0, \mathbf{u}_1 ... \mathbf{u}_{T-1})$: Initial control sequence
$\Sigma, \phi, q, \lambda$: 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 $\mathcal{E}_k = (\varepsilon_0^k, \varepsilon_1^k, ... \varepsilon_{T-1}^k)$
        **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**
```

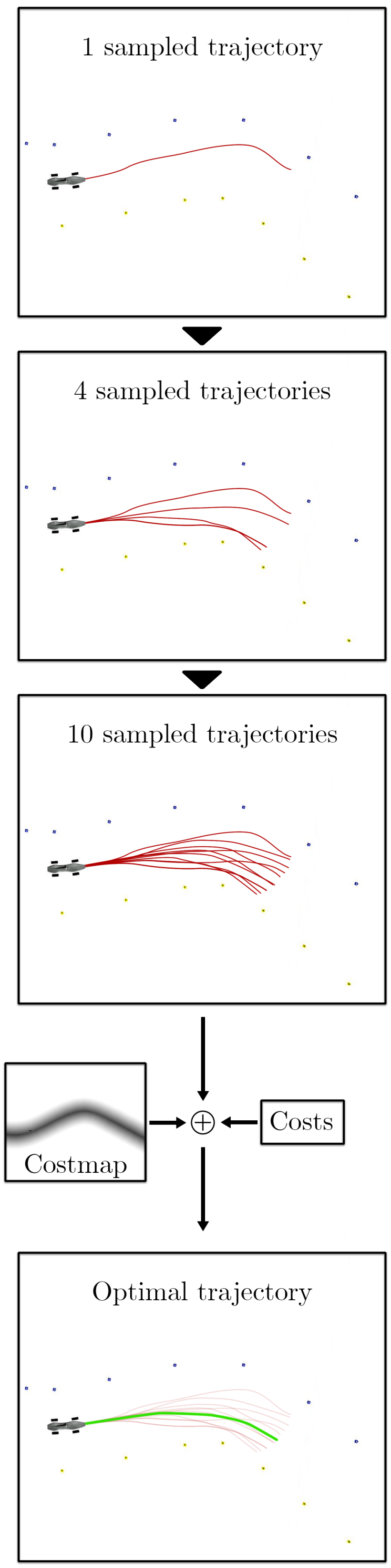


Figure 2.4: MPPI Sampling and cost-wise averaging

2.3.3 Model

In Chapter 4 of [Georgiev, 2019] the notion of kinematic ($\mathbf{x}^k$) and dynamic ($\mathbf{x}^d$) states were introduced. The kinematic state describe the position and orientation of the robot. In the case of a car, these are x-position x, y-position y and yaw ψ in the global coordinate frame. Shown mathematically in Equation 2.7 and visually in Figure 2.5.

$$\mathbf{x}^k = \begin{pmatrix} x \\ y \\ \psi \end{pmatrix} \tag{2.7}$$

$$\mathbf{x}^k_{t+1} = \mathbf{x}^k_t + k(\mathbf{x}_t)\Delta t \tag{2.8}$$

where $k : \mathbb{R}^3 \to \mathbb{R}^3$ 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 $\mathbf{x}^k_{t+1}$. This assumes linearity in the short-term. The distance can easily be obtained with the formula $S = Vt$ where S is the distance, V is velocity, and t is time. This means that $k(\cdot)$ must simple derive the velocities $\dot{x}$, $\dot{y}$, $\dot{\psi}$. Note that the slight abuse of notation here as $\dot{x}$ is the symbol both velocity and x position derivative.

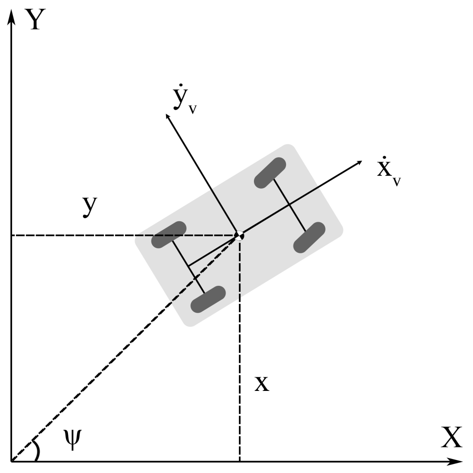


Figure 2.5: Coordinate frames of the vehicle. X_w and Y_w represent the coordinate frame of the world. $\dot{x}_v$ and $\dot{y}_v$ represent the coordinate frame of the vehicle. x, y and ψ are the kinematic state variables.

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 (using the subscript v), then we can apply an inverse 2D affine transformation to get their equivalents in the world coordinate frame:

$$\begin{aligned} \dot{x} &= \cos(\psi)\dot{x}_v - \sin(\psi)\dot{y}_v \\ \dot{y} &= \sin(\psi)\dot{x}_v + \cos(\psi)\dot{y}_v \\ \dot{\psi} &= \dot{\psi}_v \end{aligned} \tag{2.9}$$

Now that we have all of the velocities in the world coordinate frame, it is straightforward to define $k(\cdot)$ and plug it back into Equation 2.8.

$$k(\mathbf{x}) = \begin{pmatrix} \cos(\psi)\dot{x}_v - \sin(\psi)\dot{y}_v \\ \sin(\psi)\dot{x}_v + \cos(\psi)\dot{y}_v \\ \dot{\psi}_v \end{pmatrix} \tag{2.10}$$

It is now 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 ψ, it is possible also to include any other kinematic state variables that can help in these predictions. Additionally, the control signals sent to the car will also have a substantial 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:

$$\mathbf{x}_{t+1}^d = \mathbf{x}_t^d + f(\mathbf{x}_t^d, \mathbf{u}_t)\Delta t \tag{2.11}$$

where $\mathbf{u}(t)$ is the control input at time t. It is finally possible to express the *state transition function* $F(\cdot)$:

$$\mathbf{x}_{t+1} = F(\mathbf{x}_t, \mathbf{u}_t) = \begin{pmatrix} \mathbf{x}_t^k \\ \mathbf{x}_t^d \end{pmatrix} + \begin{pmatrix} k(\mathbf{x}_t) \\ f(\mathbf{x}_t, \mathbf{u}_t) \end{pmatrix} \Delta t \tag{2.12}$$

Note that this is the discrete case of the state transition function presented earlier in the definition of the motion control problem (Section 2.1).

Now the only missing piece of the puzzle is the dynamics function $f(\cdot)$ which we chose to learn with a neural network. This will be the focus for the remainder of this subsection.

Dynamics learning framework
In order to learn the dynamics, we developed a Python and PyTorch[2]-based framework which allows for easy data loading and conducting of experiments. In short, it allowed for:

- Data loading, preprocessing, normalisation and train-validate-test splitting.
- Setting up fully connected and recurrent neural networks of any size and shape.
- Running learning experiments with a set of optimisers, learning rates, weight decay and automatic result generation.

Dynamic state identification
In Section 4.3 of [Georgiev, 2019] we introduce the notion of *core dynamic state variables* which are the variables vital to computing the kinematic state in Equation 2.8. In the case of this project, these are $(\dot{x}_v, \dot{y}_v, \dot{\theta}_v)$. However, we can also introduce other dynamic state variables that aid in the calculation of $f(\cdot)$. In our thesis last year, we discovered that it was beneficial to add the roll ϕ, which gave a 181% improvement in dynamic state prediction.

Model experiments
In Section 4.4 of [Georgiev, 2019] we set out to find the best possible model by training multiple fully-connected neural networks and doing a hyper-parameter search. For that we tried different network sizes, activation functions, learning rules and learning rates. The best model we found was:

- 2 fully-connected hidden layers with 48 neurons with biases.
- tanh activation function for the hidden layers (other options are not explored due to the results in Section 4.4.3 of [Georgiev, 2019]).

[2]PyTorch is a popular Python library for deep learning

- Adam optimiser with learning rate 10^{-4}.
- 1000 epochs with batch size 100.
- 80% /10% /10% train/validation/test dataset split.

However, the most significant contribution of our modelling last year was the introduction of data normalisation (in both the inputs and outputs) which resulted in increased learning speed, better lap times and made the car significantly more stable.

2.3.4 Cost

As with any controller of the MPC family, there is an associated cost that gives the algorithm its optimisation objective. In Chapter 5 of [Georgiev, 2019], we introduced the full cost of the MPPI algorithm as:

$$Cost = \alpha_1 C_{track} + \alpha_2 C_{control} + \alpha_3 C_{speed} + C_{crash} \tag{2.13}$$

- C_{track} - based on the position of the car in the track. This was done using an offline generated costmap which was a grayscale image with pixel values in the range of [0; 1]. The lowest cost of this map was in the middle of the track.
- $C_{control}$ - cost for drastically changing controls between optimisation timesteps. This cost was used to stabilise the vehicle.
- C_{speed} - cost to reach a desired speed. This was the main incentive for the car to go fast and achieve better lap times.
- C_{crash} - incentives the car to not go in positions which result in a crash. A crash is defined by the car going outside of the track or rolling over more than 90 degrees.

2.3.5 Discussion

The view of path planning and control fields we outlined in our previous work was incorrect. We stated that the fields are separate and that in most applications, problems are solved separately; however, we not consider that view short-sighted. A better view on this is to view path planning and control as the same problem of motion control as presented in Section 2.1.

Portability or the lack thereof. All of our work last year was done in a very encapsulated way and removed from any deployment and portability concerns. As a result, the outcomes of my project last year have been very scattered, difficult to read and deploy without extensive knowledge of the work.

Use of roll in vehicle model has worked purely by luck as the car was driving in an oval-shaped track with 0,0 coordinates in its centre. This means that the roll of the car was always orthogonal to the world coordinate frame and worked in that specific case. However, if transferred to most other tracks where that would not have been the case, this vehicle model would have failed.

Normalisation results were flawed. In our report, we said that we have ”proved” that normalisation produces better results by comparing the mean square error (MSE) of normalised and non-normalised experiments. The issue with this is that the results are not directly comparable as they are in entirely different scales. Non-normalised experiments used SI unit scale, whereas normalised ones used scales of the standard deviation of 1.0. As such, the raw MSE comparison should not be taken as proof, but the lap time results should.

Costmaps generated in our report last year were sub-optimal and incentivised the controller to steer the car into the centre of the track. Sadly the centre of the track is far from the optimal path the car should take to achieve the best possible lap time; therefore, one should consider alternatives which will be discussed later in this report. Furthermore, another issue with the costmaps is that they cannot be scaled as they have $O(n^2)$ requirements in terms of memory. For a real-world implementation, this is not usable as we cannot fit the whole world into a single image.

Chapter 3

Simulation and Integration

This project is based entirely in simulation, and as such building, an accurate and reliable one is of utmost importance. Before the beginning of this project, Roborace had only a hardware-in-the-loop simulator which physically included all of the compute platforms (from Figure 2.3) and a fully simulated vehicle ECU coupled with an off-the-shelf racing track simulator and an outsourced vehicle model. This full simulation guarantees that all software which runs on it will also run on the vehicle in the real world. However, due to its complexity, it also has inherent issues such as slow load times, difficulty to operate due to the multiple machines and instability issues with all of the safety systems which limit rapid development. Due to these issues and the unnecessary simulation of perception and localisation sensors, it was decided to develop a new portable, motion control specialised simulator called the **BLC Simulator**.

3.1 BLC Simulator

The concept of this new simulator is simple - it includes the same vehicle model mentioned above and the so called stabilisation layer. The inputs are vehicle control commands and the outputs are the emulated state estimation which includes position, velocities, accelerations, slip angle measurements, wheel speed data and feedback from the vehicle controls. A high-level overview can be seen in Figure 3.1. For convenience of understanding, it can be dissected into two layers: the **Core Simulation** and the **Full Simulation**.

The Core Simulation was developed prior to this project in Matlab as a standalone module. It includes three main modules:

1. **Stabilisation Layer** - receives control inputs of the form of curvature and longitudinal acceleration (u_κ, u_a) and translates them to actuator commands while trying to achieve the desired commands and ensuring the stability of the vehicle. This is done through a torque vectoring and traction control system, which can be seen in Figure 3.2. This system is also used on the real vehicle.
2. **Vehicle Model** - determines and interpolates the vehicle model over time based on the incoming control sequence. This module is the same one used in the hardware-in-the-loop simulator introduced at the beginning of this section in order to maximise

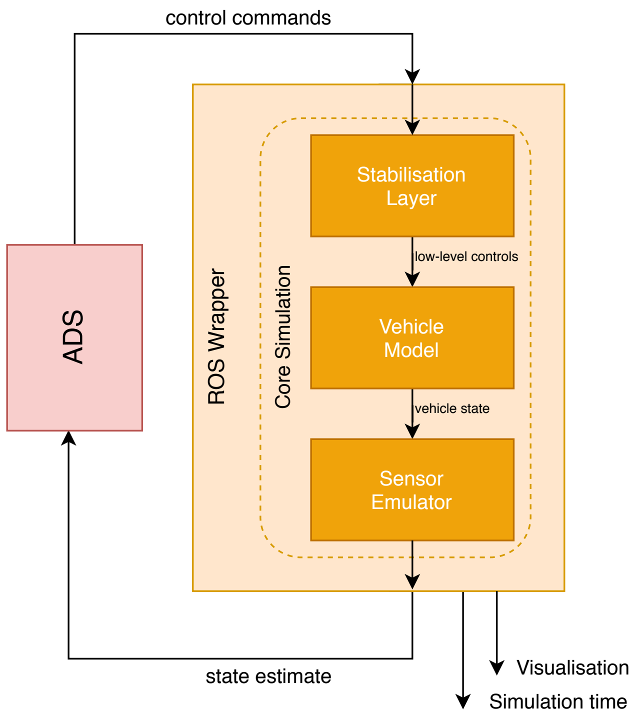


Figure 3.1: A high-level architecture of the standalone BLC Simulator.

compatibility.

3. **Sensor emulator** - translates the vehicle state output from the vehicle model into emulated sensor measurements in the format they are received on the real car.

Since the stabilisation layer and the vehicle model were initially developed in Matlab, it only made sense for the Core Simulation also to be developed in the same environment. However, all development in this report is done in a ROS environment. ROS and Matlab are compatible, but the communication between both is inefficient, slow and a resulting system is challenging to use and automate for experiments. Instead, we opted for a more ROS-based solution which tackles the fore-mentioned issues. We call this the *BLC Simulator*.

To achieve the desired result, we first compiled the Core Simulation as an executable C program from which was created a standalone ROS package. Afterwards, a new *ROS Wrapper node* was developed, which serves as an interface from ROS to the Core Simulation, as seen in Figure 3.1. This wrapper feeds the command inputs to the stabilisation layer and handles the outputs of the Core Simulation. Namely, it converts the state estimate outputs into the correct coordinate frames according to the REP 103 standard[1]. Additionally, it also publishes a ROS coordinate transform (`tf`) and adds visualisations of the car and the racing track using RViz as seen in Figure 3.3. This is the only visual feedback of the BLC Simulator, and it is dynamically turned off if not used.

[1]REP 103 is the ROS standard for coordinate frames. It can be found online `https://www.ros.org/reps/rep-0103.html`

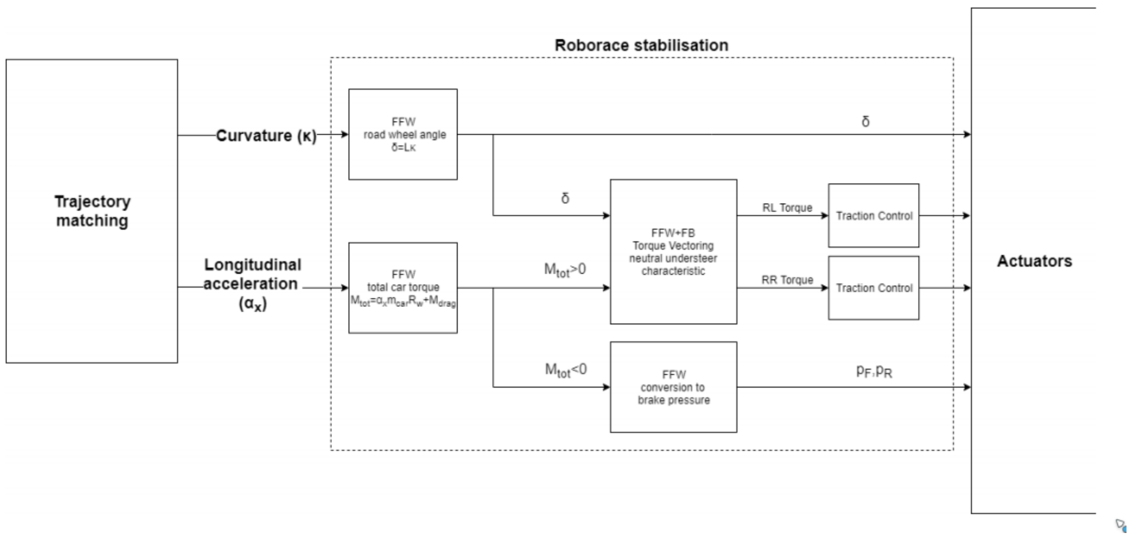

Roborace stabilisation
Trajectory matching
Curvature (κ)
Longitudinal acceleration (αx)
FFW road wheel angle δ=Lκ
FFW total car torque Mtot=αxmcarRw+Mdrag
δ
δ
Mtot>0
Mtot<0
FFW+FB Torque Vectoring neutral understeer characteristic
FFW conversion to brake pressure
RL Torque
RR Torque
Traction Control
Traction Control
pF,pR
Actuators

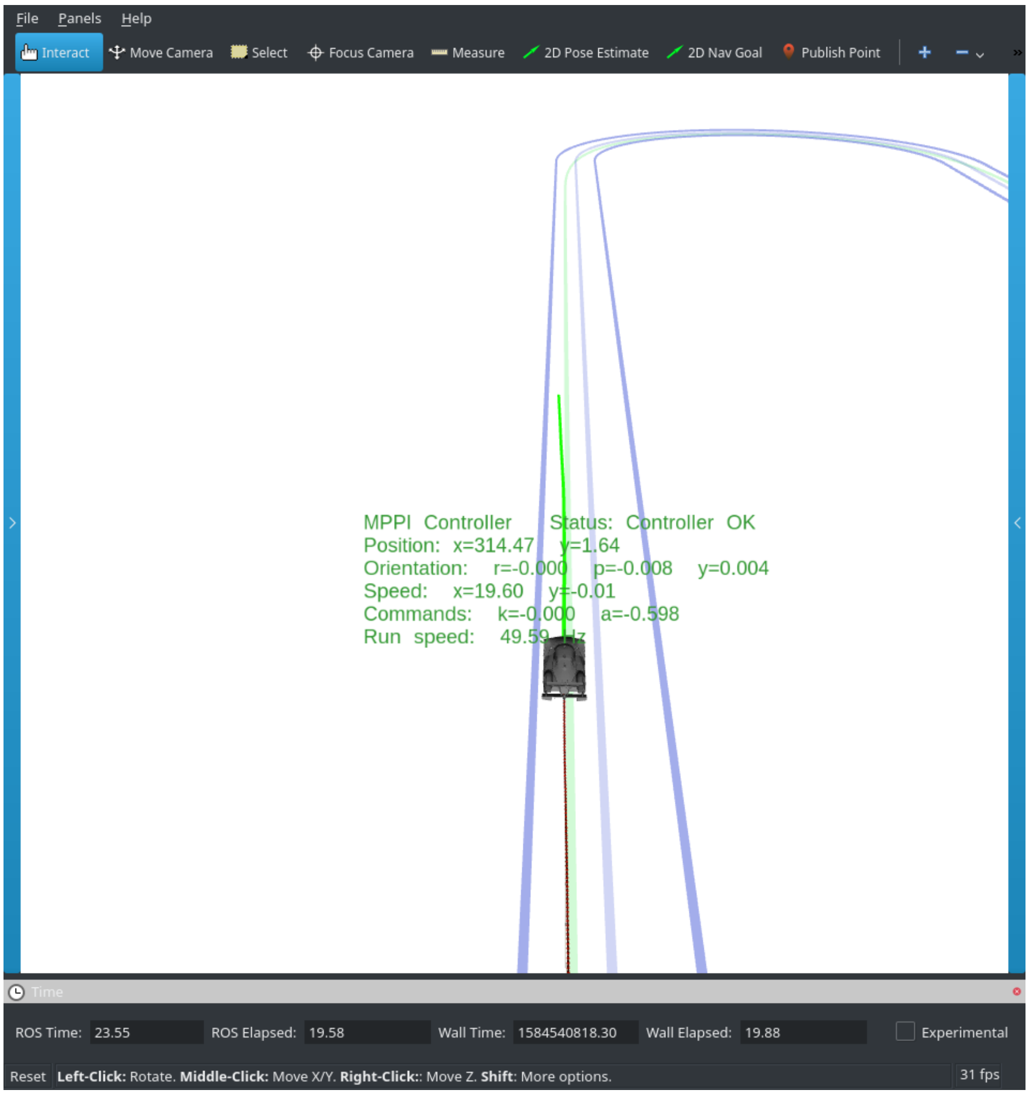
Figure 3.2: Overview of the stabilisation layer. It receives control inputs from some other system (in this diagram Trajectory matching) and converts them to actuator commands. FFW and FFB stand for feed-forward and feed-back. δ is the steering angle, M_{tot} is the total motor torque and p_F and p_R are the front and rear brake pressures. The stabilisation layer decides when to accelerate and brake the car based on the acceleration command and predefined maps. The torque-vectoring aims to stabilise the vehicle during cornering by altering the individual motor torques sent to both rear wheels. This results in a more stable and faster cornering performance.

3.2 Integration

As presented in Section 2.3, MPPI needs two major inputs: (1) a state estimate of the vehicle and (2) a map of the environment, as well as a means of controlling the vehicle. To achieve this, we chose to reuse as much as possible from the existing modules as they have been proven to work and minimise the integration work needed. The full MPPI integration is best shown via Figure 3.4. To provide more background on individual modules:

- The costmap is generated from an offline map which was already pre-existing. These offline maps consist of geographical coordinates of the inner and outer boundaries of the track, centerline and a racing line. The costmap generation process itself is based on our previous work in [Georgiev, 2019] but is also improved which is detailed in Section 6.1.
- Roborace already has an existing localisation system which provides all the necessary state inputs that would be needed for this project. This includes position, orientation, velocities, turning rates, accelerations as well as wheel speed data and battery state. These estimates are provided at a rate of 250 Hz which is more than sufficient.
- The *Path Planner* is an externally developed module which generates a trajectory consisting of speed, acceleration and curvature based on the offline generated racing line. This is used to feed MPPI with a reference velocity for the optimisation. This is further elaborated later in Section 6.3.

- The *Speedgoat block* encompasses everything that runs on the Speedgoat computer as explained in Section 2.2. In the particular case of MPPI, only the stabilisation layer as explained in the previous section is running. The Speedgoat module in this diagram also represents the abstraction layer assumed for this project where its interface is the abstracted interface to the car.
- The iterative learning module is provided here for completeness but can be ignored for now; it is later explained in Section 4.7.



```
MPPI Controller     Status: Controller OK
Position:  x=314.47   y=1.64
Orientation:   r=-0.000   p=-0.008   y=0.004
Speed:   x=19.60   y=-0.01
Commands:   k=-0.000   a=-0.598
Run speed:   49.59 Hz
```

```
ROS Time: 23.55   ROS Elapsed: 19.58   Wall Time: 1584540818.30   Wall Elapsed: 19.88   Experimental
Reset   Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Move Z. Shift: More options.   31 fps
```

Figure 3.3: Visualisation of the BLC Simulator in RViz with the MPPI algorithm driving. The car visualisation is given by the simulator. The racing circuit is also shown with blue lines for the boundaries and middle line and a vague green line for the offline computed racing line. The red trail left by the car is its interpolated position, and the solid green line is the currently planned trajectory. The green text above the car is the debugging data from the MPPI algorithm itself and not connected to the simulation.

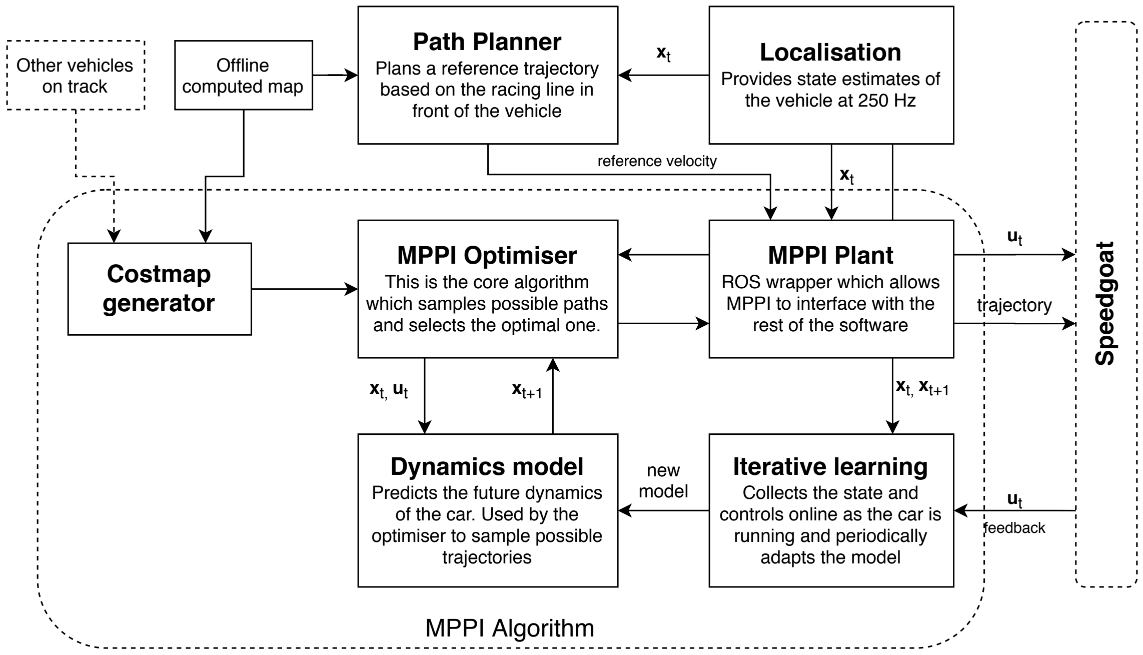

Other vehicles on track
Offline computed map
Path Planner
Plans a reference trajectory based on the racing line in front of the vehicle
x_t
Localisation
Provides state estimates of the vehicle at 250 Hz
reference velocity
x_t
Costmap generator
MPPI Optimiser
This is the core algorithm which samples possible paths and selects the optimal one.
MPPI Plant
ROS wrapper which allows MPPI to interface with the rest of the software
u_t
trajectory
Speedgoat
x_t, u_t
x_{t+1}
x_t, x_{t+1}
Dynamics model
Predicts the future dynamics of the car. Used by the optimiser to sample possible trajectories
new model
Iterative learning
Collects the state and controls online as the car is running and periodically adapts the model
u_t
feedback
MPPI Algorithm

Figure 3.4: Integration of MPPI into the Roborace software. All modules inside the MPPI Algorithm section are developed for the purposes of this project. Everything else in the diagram has been developed outside of this project.

Chapter 4

Dynamics modelling

Modelling robot dynamics has become a field of itself. Traditionally, this has been done by leveraging decades of research in physics, attempting to formulate them mathematically and use them computationally. However, this is limited by our understanding of dynamics and computational reasons. A more attractive alternative is data-driven modelling, where we attempt to find structure in data or fit some basic models to that data.

In this chapter, we present several data-driven methods of dynamic modelling for model-based control, conduct experiments with each in the context of the project and finally draw conclusions for the final model to be used for the remainder of the thesis. In Section 4.1, we present the procedures used to obtain data for the experiments. In Sections 4.2 - 4.5 we propose three different model types and explore them. Finally, in Section 4.7, we apply the best model to an online learning scenario.

Note that this chapter only explores the dynamic state of the vehicle $\mathbf{x}^d$. The kinematic state $\mathbf{x}^k$, as presented, remains unchanged from our previous work and is fully explained in Section 2.3.3.

4.1 Data preparation

All vehicle models developed in this chapter are data-driven and as such, depend highly on good quality state and control data. As such, before delving into the core of vehicle modelling, we would first like to present how data is collected, processed and prepared.

Data collection
In our previous work, we collected data by manually driving the car in simulation in different conditions. However, this was no longer feasible due to constant changes in the simulation. Instead, a more automated method of data collection was developed where the vehicle was scripted to drive and explore its state space [Christoforos Chatzikomis, personal communication, 2019]. This was done in two separate methods, one for longitudinal dynamics and one for lateral dynamics as presented in Algorithms 2 and 3. These approaches have been empirically developed based on practical experience. Good state exploration has been verified with a pair-wise scatter plot along all state dimensions found

in Appendix A.1. Using both of these approaches, we have collected one hour of driving data, which is used for the remainder of this report.

Algorithm 2: Longitudinal state exploration

```
$V$: Current longitudinal velocity
$V_{max}$: maximum velocity to explore
$N_v$: Number of sampled velocities
$N_p$: Number of sampled controller parameters
$PID(\cdot)$ : controller which accepts $K_P$, $K_I$ and $K_D$ parameters
$\sigma_P$, $\sigma_I$, $\sigma_D$: Controller sampling variances

for $N_p$ do
    Sample $K_{\{P,I,D\}} \sim |\mathcal{N}(0, \sigma_{\{P,I,D\}})|$
    Initialise new controller $PID(K_P, K_I, K_D)$
    for $N_v$ do
        Sample $V_{target} \sim Uniform(0, V_{max})$
        while $error > 0.1$ do
            $error = V_{target} - V$
            Compute new control $u_a = PID(error)$
            apply $u_a$
        end
    end
end
```

Algorithm 3: Lateral state exploration

```
$V$: Current longitudinal velocity
$V_{max}$: maximum velocity to explore
$N_v$: Number of sampled velocities
$N_k$: Number of sampled curvature commands
$a_y^{max}$: Maximum lateral acceleration
  $PID(K_P = 1)$ : Velocity controller
for $N_v$ do
    Sample $V_{target} \sim Uniform(0, V_{max})$
    for $N_k$ do
        while $error > 0.1$ do
            $error = V_{target} - V$
            Compute new control $u_a = PID(error)$
            apply $u_a$
        end
        $\kappa_{max} = \frac{a_y^{max}}{V^2}$
        Sample curvature command $u_k = Uniform(-\kappa_{max}, \kappa_{max})$
        Apply $u_k$
        Wait 1 second
    end
end
```

Data interpolation and re-sampling

Similar to our previous work, the data here is collected at a high frequency of 250 Hz and needs to be resampled at 50 Hz (the target MPPI run rate) for dynamics learning procedure. This is done by first linearly interpolating the data and then resampling it at the desired frequency.

Data filtering

Due to the limitations of some dynamic models developed later in this Chapter, we needed to introduce filtering of the data during loading. Thus we developed a modular framework which needs as input the dimension along which to detect outlier data and the numerical threshold for the outlier. Afterwards, the framework removes all data (along all dynamics dimension) 1 second around the outlier data. This procedure safely assumes that only outlier driving scenarios will lead to outlier data as such those scenarios are best to be fully removed. For the purposes of this project, we use this framework to consistently remove data with roll values $|\theta| > 0.35$ rads and slip angles[1] $\beta > 0.785$ rads.

[1]slip angle is given by $\beta = -\arctan(\dot{y}/|\dot{x}|)$

Data smoothing

This procedure remains unchanged from our previous work. There we noticed that the collected data was very noisy even in simulation. This makes the learning process more complicated, and as such, we chose to smooth the data using one-dimensional splines fit through time using a smoothing factor $s = \sigma/10$ where σ is the standard deviation of the respective data dimension. The results of the smoothing can be seen in Figure 4.1.

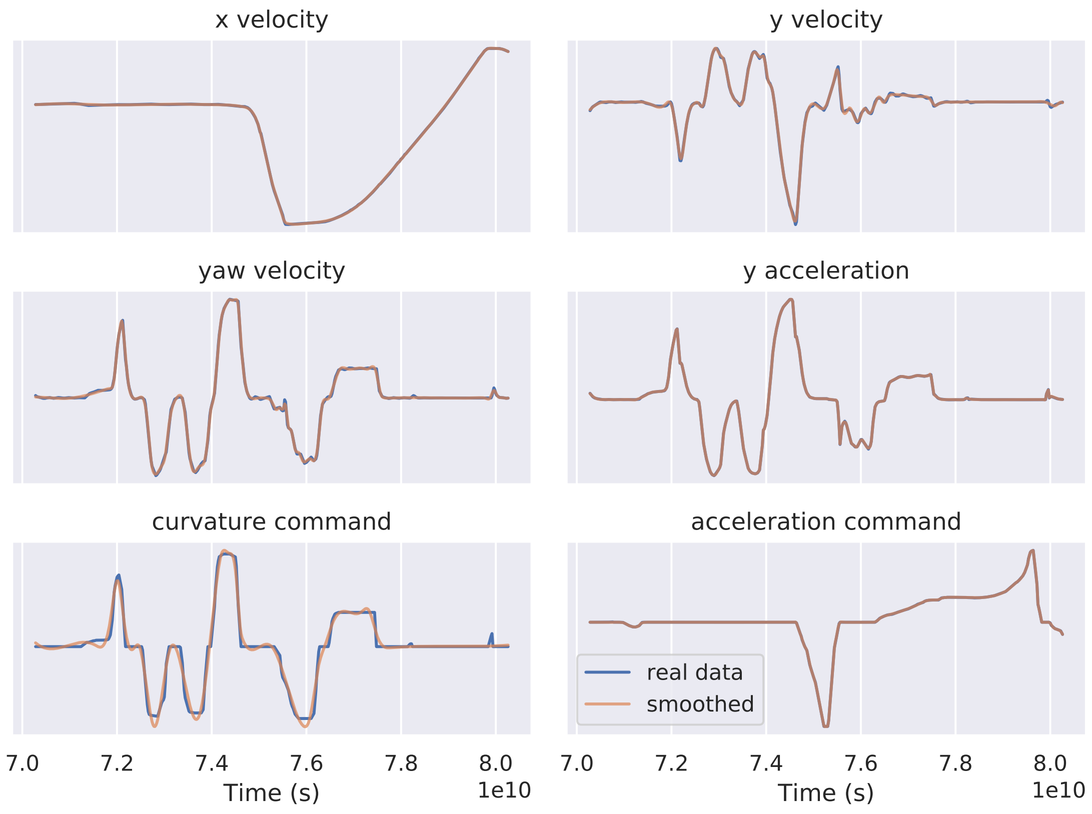


Figure 4.1: An example of dataset smoothing using splines. This is a random snapshot of 500 data points.

Data preparation for learning

Remains the same as in our previous work but is provided here for completeness.

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

$$\mathbf{X}_t = \left(\mathbf{x}_t^d, \mathbf{v}_t\right)$$

Then, the targets are defined as:

$$\mathbf{Y}_t = \left(\mathbf{X}_{t+1} - \mathbf{X}_t\right)\Delta t$$

Δt is needed in this case to scale up the outputs linearly to SI units.

4.2 Dynamics model types

As presented in Section 2.3.3, in our previous work, we have used solely neural network dynamic model without any justification. We now consider that outlook on modelling single-sided and would like to present a complete view on dynamics modelling for model-based control.

Dynamics models are essential for robot control as they allow us to start the modelling process from the lowest physical entity - forces and build up from there until the full robot state is explained. Traditionally, these models are built from first principles Newtonian dynamics harvesting generations of knowledge in physics. However, even then, these models are not very accurate due to the limited understanding of physics, manufacturing variance and computational limitations.

To overcome these issues, the notion of **parametric models** was introduced, which parameterised a few physical quantities and then fit them to data using standard parameter estimation techniques [Hollerbach et al., 2016], [Zorzi, 2014]. The advantages of this are that the model is explainable, predictable and generalises to larger state space. However, these models are subject to the user's understanding of dynamics and the capabilities of the parameter estimation method. Often to guarantee convergence of the parameters, these models are made linear, which further limits their capacity to capture the real underlying dynamics of the system.

On the other hand, we have the family of **non-parametric models** which can use any function approximator to express the dynamics of the system. These data-driven models have the capability of capturing dynamics much more accurately and without the requirement for extensive dynamics knowledge beforehand. However, similar to their parametric counterparts, the non-parametric models also have their drawbacks as they are not intuitively explainable, require larger amounts of data to be trained and often fail to generalise to unseen states [Hollerbach et al., 2016]. Still, there have been successful results models in the form of neural networks [Williams et al., 2017], basis functions [Williams et al., 2016], Gaussian regression [Zorzi and Chiuso, 2015] [Zorzi and Chiuso, 2017] and more.

Finally, we have the intersection of the two model types mentioned before called **semi-parametric models**. These models combine parametric and non-parametric models into one intending to harvest the benefits of both - maintain the generalisation and explainable properties of parametric models and the accuracy of non-parametric ones. In these semi-parametric models, the parametric components are designed to handle the rigid body dynamics, while the non-parametric ones handle the unmodeled dynamics (also known as residual learning) [Smith and Mistry, 2020]. Figure 4.2 summarises these three categories of models.

4.3 Parametric vehicle model

In the world of car dynamic,s it is common to split dynamics modelling into chassis and tyre modelling, with the latter being more challenging to model.

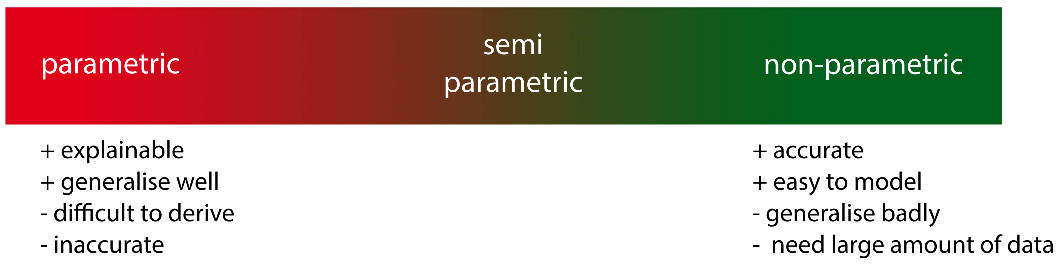


Figure 4.2: Visual explanation of the three types of vehicle models presented in this chapter.

4.3.1 Chassis modelling

Due to the inherent symmetry of cars, it is common to simplify 4-wheeled motors to a 2-wheeled one where the front and rear set of tyres are lumped into only one at the front and one at the real. This is commonly referred to as the *bicycle model* and can be seen in Figure 4.3. The tyres at each axle exhibit twice the cornering stiffness and force capability in comparison to a 4-wheeled model. The model can be summarised with the following state transition:

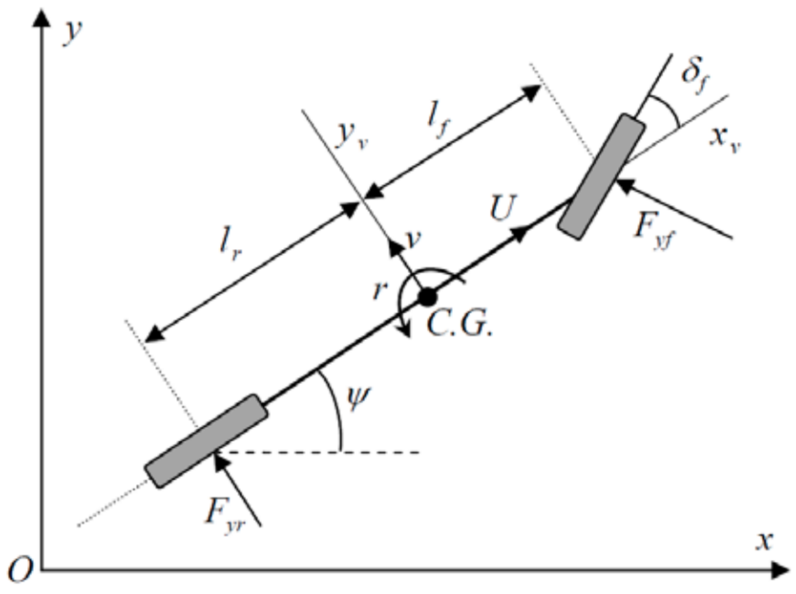


Figure 4.3: The bicycle model which lumps both of the front tyres into one front tyre and similar for the rear. [Hindiyeh, 2013]

$$\begin{aligned}\ddot{x} &= \dot{\psi}\dot{y} + u_a \\ \ddot{y} &= -\dot{\psi}\dot{x} + \frac{2}{m}(F_{yf}\cos u_\delta + F_{yr}) \\ \ddot{\psi} &= \frac{2}{I_z}(l_f F_{yf} - l_r F_{yr})\end{aligned} \tag{4.1}$$

where F_{yf} and F_{yr} are the lateral tyre forces at the front and rear respectively. The parameters of this chassis model are:

- m - mass
- I_z - yaw inertia
- l_f - distance from CoG to front axle
- l_r - distance from CoG to rear axle

This chassis model assumes planar motion of the vehicle and as such does not incorporate rate and pitch motion of the chassis. In principle, these help with identifying load transfer [Reza, 2008] and such a full 4-wheel chassis model would be more suitable. However, the complexity of that was deemed unfeasible for the time-frame of this project.

4.3.2 Tyre modelling

Except for aerodynamic drag and gravity, tyre ground forces are the only external force acting on the vehicle. The tyre forces have highly non-linear behaviour at the limits of friction which is a function of the slip angle defined as $\beta = \arctan(\dot{y}/|\dot{x}|)$. However, it is also on that limit that tyres exhibit maximum force which is required for aggressive driving. Thus, correct tyre modelling is essential.

When modelling tyres, it is normal to express the lateral tyre forces F_y as a function of the tyre slip angle [Reza, 2008]. Due to the effects of steering, the slip angles at the front and rear axles are different and can be modelled as:

$$\begin{aligned} \alpha_f &= u_\delta - \arctan\frac{\dot{y} + \dot{\psi} l_f}{\dot{x}} \\ \alpha_r &= -\arctan\frac{\dot{y} - \dot{\psi} l_r}{\dot{x}} \end{aligned} \tag{4.2}$$

The simplest possible tyre model is the **linear tyre model** which assumes that the vehicle operates only in the *linear operating region* [Hindiyeh, 2013] in which it is accurate. However, as soon as it leaves that, the model breaks down. The lateral tyre forces can be expressed simply as

$$F_y = C_s \alpha \tag{4.3}$$

where C_s is the cornering stiffness parameter, this combined with the dynamic bicycle model in Equation 4.1 gives us our first complete vehicle model which we will call the **linear bicycle model**.

Since we are operating in the non-linear part of the dynamics, the above-mentioned tyre model is inadequate. A better alternative would be the so-called *tyre brush model* which divides the tyre into three substructures: the ring, carcass and flexible ”brushes” representing thread elements that come into contact with the road as shown in Figure 4.4 [Hindiyeh, 2013]. There exist many different approaches to modelling these brushes and their interaction with the road surface [Reza, 2008]. For this thesis, we will consider the

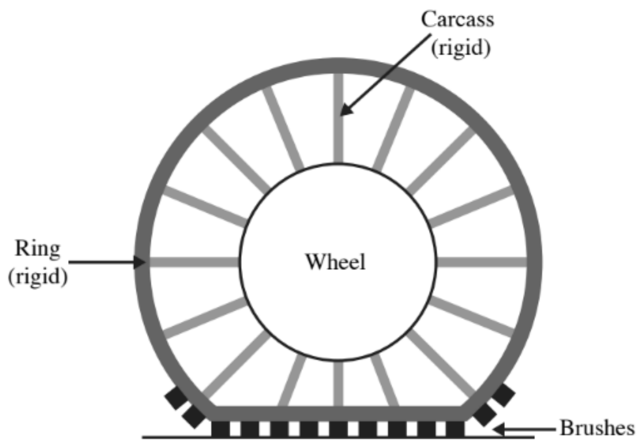


Figure 4.4: Components of the brush tire model and assumptions associated with the brush tire model in this dissertation. [Hindiyeh, 2013]

.

variant proposed by Fromm [Hadekel, 1952]. By that model, the lateral tyre forces are defined as:

$$F_y = \begin{cases} -C_s \tan\alpha + \frac{C_s^2}{3\mu F_z}|\tan\alpha|\tan\alpha - \frac{C_s^3}{27\mu^2 F_z^2}\tan^3\alpha & |\alpha| \leq \alpha_{crit} \\ -\mu F_z sgn(\alpha) & |\alpha| > \alpha_{crit} \end{cases} \tag{4.4}$$

where C_s is again the cornering stiffness, α is the slip angle of the tyre given by Equation 4.2, μ is the friction coefficient between the road surface and the tyre, F_z is the vertical force for the tyre which for our simplified model can be calculated for the front and rear axles with:

$$\begin{aligned} F_{z,f} &= \frac{l_r}{2(l_f + l_r)} mg \\ F_{z,r} &= \frac{l_f}{2(l_f + l_r)} mg \end{aligned} \tag{4.5}$$

where m is the mass of the vehicle and g is the gravitational force. It is worth noting that this vertical force can be improved with the addition of roll and pitch dynamics. Finally, α_{crit} from Equation 4.4 is the smallest magnitude slip angle at which the entire contact patch is at the friction limit. This can be derived analytically to be:

$$\alpha_{crit} = \arctan\frac{3\mu F_z}{C_s} \tag{4.6}$$

With this model, it is possible to capture two significant characteristics of the tyre behaviour: linear dependence of tyre upon slip angle at small slip angles and saturation due to friction limitation at large slip angles. Combining this with the dynamic bicycle model from Equation 4.1 we obtain a more sophisticated model which we will label the **dynamic bicycle with tyre brush model**.

A major criticism of the tyre brush model is that even though it is analytically derived, it is not accurate. A different approach was adopted by Pacejka, who pioneered data-driven tyre models [Pacejka, 2005]. These models are not derived analytically but are instead mathematical formulations fitted real data. This gave rise to the so-called ”Magic Tyre Formula” which is composed of 96 atomic equations with 18 unique parameters [Pacejka, 2005]. However, the complexity of this formulation is too high; instead, here we will consider the simplified model:

$$F_y = F_z D \sin\left(C \arctan\left(B\alpha - E\left(B\alpha - \arctan\left(B\alpha\right)\right)\right)\right) \tag{4.7}$$

where there are four parameters A, B, C and D . This model combined with the dynamic bicycle model from Equation 4.1, we will call the **dynamic bicycle with magic tyre model**.

Finally, we present in Figure 4.5 a graphical comparison between the three tyre models presented in this subsection. The peak of magic tyre model can be considered the point of maximum saturation for the tyres at which they exhibit maximum lateral force. Ideally, the vehicle would get peak performance if it stays in that region.

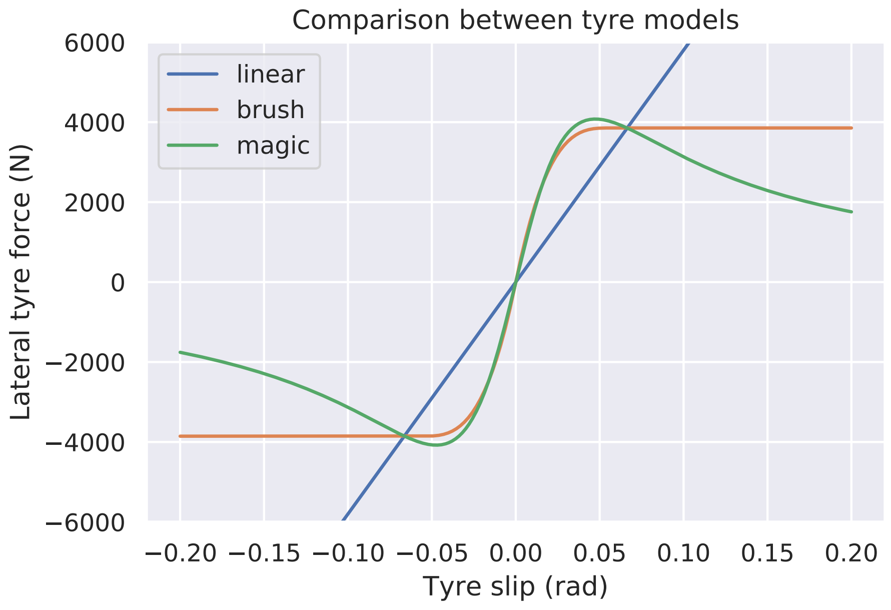


Figure 4.5: Comparison between the three tyre models used in this thesis. These models are fitted to data from the simulation.

4.3.3 Model comparisons

Using the above-defined models, it now possible to parameterise them and fit them to data from simulation gathered with the procedure explained in Section 4.1. The parameterisation procedure is shared between all models, and a few parameters are fixed as they are known physical quantities from Section 2.2:

Mass	1350 kg
Front axle to CoG	1.5 m
Rear axle to CoG	1.4 m
Gravity	9.81 m/s^2

Table 4.1: Fixed parameters for all parametric models

The other parameters are defined as free ones and fit using non-linear least-squares optimisation provided by the Python `lmfit` library. For all models, the tyre parameters were split to independent front and rear parameters, except for the μ friction coefficient in the dynamic tyre brush model. Full results can be seen in Table 4.2. The full models and identified parameters for these experiments can be found in Appendix B.

Model	MSE Error	$\dot{x}$ norm. error	$\dot{y}$ norm. error	$\dot{\psi}$ norm. error
Dynamic bicycle with linear tyres	4.64559	1.053	8.074	0.990
Dynamic bicycle with tyre brush model	1.69144	1.053	0.886	0.643
Dynamic bicycle with magic tyre formula	1.64934	1.053	0.780	0.641

Table 4.2: Comparison between fitted parametric models. All of the models are fit on the same data. The MSE error refers to the mean square error over all the dynamical states. The three other columns show error over each dynamic state normalised by its standard deviation.

Unsurprisingly, the dynamic bicycle with linear tyres performs the worst with a significantly larger error than the two other models. Both the tyre brush and magic tyre models provide improvements in terms of overall prediction. However, the overall mean square error does not tell us the full story. Since the three dynamic states have different scales, it is appropriate to compare their standardised results which can be seen on the right-hand side of Table 4.2. From those columns, we can see that the prediction of $\dot{x}$ does change between the models. This is because those dynamics are not affected by tyre dynamics according to the dynamic bicycle model from Equation 4.1. Furthermore, the reason for the relatively high error in $\dot{x}$ is due to a poorly tuned stabilisation layer. However, since those dynamics do not affect the model comparison here, they can be ignored.

Finally, we opted for the dynamic bicycle with tyre brush model as it doesn't degrade performance significantly, has analytical proof and its fitting process is more stable (see Appendix B). We coupled this model with MPPI, optimised it to sample 2560 possible trajectories at 50 Hz and ran experiments at a target baseline speed of 16 m/s. The achieved results are presented in Table 4.3, and we will use these as the baseline for future modelling experiments. The full list of the MPPI parameters used for these experiments can be found in Appendix C.2. Tests with the other parametric models were not run due to timing constraints.

Average lap time	122.647
Min. lap time	122.369
Lap time std.	0.412
Max lateral acceleration	11.716
Max slip angle	0.0746

Table 4.3: Lap time experiments using the dynamic bicycle with brush tyre model. Results are shown for five consecutive laps using the same control parameters. All values are in SI units.

4.3.4 Torque vectoring effects

Since DevBot V2 is an electric car with two motors at the rear, it can run them at different speeds to enhance the handling of the vehicle [Chatzikomis et al., 2017]. This system also affects the overall vehicle dynamics as presented in this chapter. Due to the complexity of torque vectoring, it is not feasible to model the full system in our vehicle model. Instead, we shall introduce a torque vectoring term to the yaw rate prediction of the chassis model in Equation 4.1 [Kabzan et al., 2019]. With that, the new chassis model becomes:

$$\begin{aligned} \ddot{x} &= \dot{\psi}\dot{y} + u_a \\ \ddot{y} &= -\dot{\psi}\dot{x} + \frac{2}{m}(F_{y,f}\cos u_\delta + F_{y,r}) \\ \ddot{\psi} &= \frac{2}{I_z}(l_f F_{y,f} - l_r F_{y,r} + \frac{u_\delta \dot{x}}{l_f + l_r} - \dot{\psi} P_{TV}) \end{aligned} \tag{4.8}$$

where P_{TV} is the proportional torque vectoring gain; a parameter that can be identified from data.

Using this new chassis model, we can now couple it with the tyre brush model from Section 4.3.2, fit it to data generated by manual driving with torque vectoring and compare the effects of the newly introduced torque vectoring. Results are shown in Table 4.4. Note that these results are on different data that the one used in the previous subsection. Of interest is only the column of the $\dot{\psi}$ normalised error and interestingly, the error seems to have increased slightly. This might be as a result of a conflict between the tyre and P_{TV} parameters. Another explanation for these results might be the insignificant torque vectoring effects of the stabilisation layer on the vehicle. Due to these results, torque vectoring gain is not considered further.

4.4 Non-parametric vehicle model

Building on the non-parametric models from our previous work (Section 2.3.3), we have trained non-parametric models in the form of *fully-connected neural networks*.

Model	MSE Error	$\dot{x}$ norm. error	$\dot{y}$ norm. error	ψ norm. error
Without torque vectoring	1.03502	0.395	0.0297	0.04096
With torque vectoring	1.03411	0.395	0.0295	0.04114

Table 4.4: Comparison between fitted parametric models with and without torque vectoring. The MSE error refers to the mean square error over all the dynamical states. The three other columns show error over each dynamic state normalised by its standard deviation.

4.4.1 Dynamic state identification

Similar to Section 4.3 of [Georgiev, 2019], we first have to select the inputs to the model from the dynamics states of the vehicle. However, unlike our previous work, instead of blindly experimenting with all combination of dynamic states, we can harness physics understanding from the previous subsection and more efficiently select state variables that can contribute to the vehicle model. Apart from the core dynamic variables ($\dot{x}, \dot{y}, \dot{\theta}$), we have selected the roll ϕ and lateral acceleration $\ddot{y}$ as additional dynamic states. Experiments were run using the network architecture from Section 2.3.3 but without normalisation for 1000 epochs using a batch size of 100.

Dynamic variables	Validation MSE
$\dot{x}, \dot{y}, \dot{\theta}$	0.344
$\dot{x}, \dot{y}, \dot{\theta}, \psi$	0.354
$\dot{x}, \dot{y}, \dot{\theta}, \ddot{y}$	0.436
$\dot{x}, \dot{y}, \dot{\theta}, \psi, \ddot{y}$	0.489

Table 4.5: Experiment results for neural network dynamics learning using different combination of dynamic state inputs. The error shown is the mean square error only for the core dynamic variables $\dot{x}, \dot{y}, \dot{\theta}$.

From the experiment results in Table 4.5, we obtain only confusion. Not only do the new explored variables not enhance the model but even the roll ψ identified to aid the model from our previous work does not help in this case. On further investigation, explanations for these results were found:

1. Roll is always in the global frame, and as such, it will result in different values depending on the position of the car in the track. The simplest example would be that the roll values would invert themselves if the car drives in the opposite direction of the track. This issue was also present in our previous work ([Georgiev, 2019]), however, due to the small oval-shaped track, the roll was always ”in the correct frame” as it was pointing outwards of the centre of the track. Additionally, this is also the case for the original authors of the MPPI algorithm ([Williams et al., 2017].

2. The reason why $\ddot{y}$ did not aid in the model was due to the drastically different scales of it and the other variables. For example, $\dot{y}$ and $\dot{\theta}$ have standard deviation of 0.65 and 0.33 respectively in this dataset. However, $\ddot{y}$ has a standard deviation of 4.23,

close to an order of magnitude larger. As is well explored in the machine learning field, neural networks do not perform well on non-standardised data. Note that all variables are in SI units.

To give lateral acceleration another chance, we reran the experiments, this time with normalisation of both inputs and outputs of the neural network. The results presented in Table 4.6 finally show the expected results - lateral acceleration improves model accuracy by 32%. Note that the errors are drastically different from Table 4.5 as the outputs are normalised. Due to these results, from now, we will only do further experiments with normalised data.

Dynamic variables	Validation MSE
$\dot{x}, \dot{y}, \dot{\theta}$	2.131
$\dot{x}, \dot{y}, \dot{\theta}, \ddot{y}$	1.618

Table 4.6: Experiment results for neural network dynamics learning using different combination of dynamic state inputs. The error shown is the mean square error only for the core dynamic variables $\dot{x}, \dot{y}, \dot{\theta}$.

4.4.2 Model experiments

Using the best dynamic state identified from the previous subsection, we now aim to identify the best model hyper-parameters. Several experiments were run with different hidden layer sizes, learning rates and optimisers. Results shown in Table 4.7. Full MPPI hyper-parameters are shown in Appendix C.2.

A couple of interesting insights can be gained:

- The Adam optimiser [Kingma and Ba, 2014] continuously outperforms the RMSProp optimiser.
- Lower prediction error does not always translate to better lap times. This might be an artefact from the limitations of a fixed validation set; however, that does not mean that the training hyper-parameters perform worse.
- The neural networks with hidden layers of 32,32 do not fair well in the experiments and often resulted in complete failures or large lap time standard deviations. Larger hidden layers have more stable results and capture the dynamics better; however hidden layer sizes of 64,64 has diminishing results over the 48,48 case.

With that in mind, we select the 48,48 hidden layer size model with Adam optimiser and a learning rate of 0.00075 (bolded in Table 4.7 as the non-parametric model of choice for later comparisons.

Hidden layer size	Learning rate	Optimiser	Validation MSE	Average lap time	Lap time std.
32,32	0.005	RMSProp	1.3327	122.00	1.82
32,32	0.005	Adam	1.3066	126.62	0.39
32,32	0.0025	RMSProp	1.3133	-	-
32,32	0.0025	Adam	1.1402	139.85	14.67
32,32	0.001	RMSProp	1.1558	-	-
32,32	0.001	Adam	1.3157	124.70	0.26
32,32	0.00075	RMSProp	1.1314	140.56	0.15
32,32	**0.00075**	**Adam**	**1.1066**	**124.03**	**0.57**
32,32	0.0005	RMSProp	1.2127	120.57	0.34
32,32	0.0005	Adam	1.1948	-	-
48,48	0.005	Adam	1.1786	119.5824	0.3203
48,48	0.0025	Adam	1.1541	120.8680	0.2435
48,48	0.001	Adam	1.1441	125.1408	0.2435
48,48	**0.00075**	**Adam**	**1.0251**	**127.3658**	**0.2345**
48,48	0.0005	Adam	1.1271	119.3318	0.2588
64,64	0.005	Adam	1.3273	123.0038	0.4020
64,64	0.0025	Adam	1.1127	122.8498	0.2301
64,64	**0.001**	**Adam**	**1.0126**	**124.2694**	**0.2474**
64,64	0.00075	Adam	1.0729	124.0948	0.2329
64,64	0.0005	Adam	1.0373	123.4382	0.2319

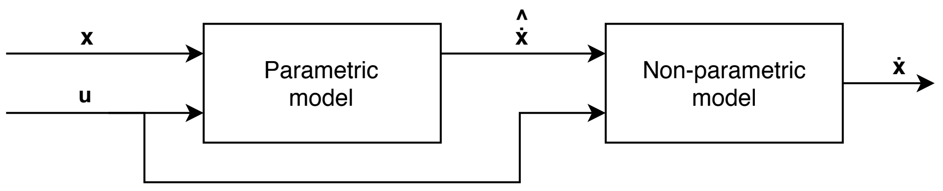
Table 4.7: Neural network training experiments for different hidden layer size, learning rates and optimisers. All experiments are run on inputs from $\dot{x}, \dot{y}, \dot{\theta}, \ddot{y}$ and normalised data. Average lap times are estimated over 5 consecutive laps and whenever any one was failed, that is labelled as a *failed model*. The best hyper-parameters for each hidden layer size have been bolded. Note that RPMSProp optimiser experiments with 48,48 and 64,64 hidden layer sizes are omitted for brevity as the results are similar to the 32,32 hidden layer size case.

4.5 Semi-parametric vehicle model

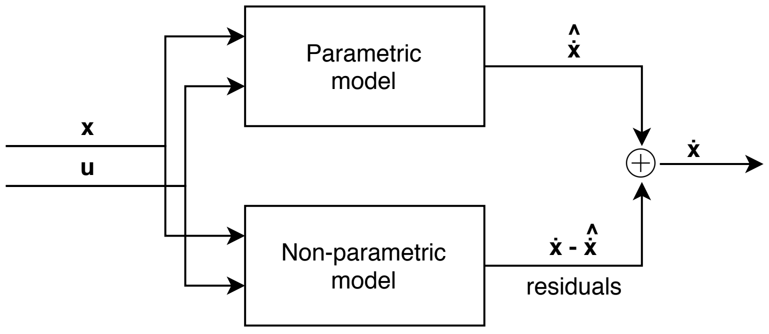
4.5.1 Types

Now we can combine the results from the parametric models from Section 4.3 and the non-parametric models from Section 4.4 into the so called semi-parametric models, harvesting the benefits of both.

The first architecture we propose is where the state is first fed into a parametric model, and then its predicted state derivatives are fed to a non-parametric model. We call this the **cascade semi-parametric model** and is shown in Figure 4.6a. In this case, the parametric part is acting as a filter to all possible states, reducing the effective state space of the non-parametric part which then learns the effects that were left unmodeled in the parametric part.



(a) Cascade semi-parametric model.



(b) Parallel semi-parametric model.

Figure 4.6: The two types of semi-parametric models. In both cases, the parametric model performs the core prediction, and the non-parametric part learns its residuals (i.e. errors). **x** and **u** are the state and control inputs. $\hat{\dot{\mathbf{x}}}$ is the standalone prediction of the parametric part and $\dot{\mathbf{x}}$ is the overall prediction of the full semi-parametric model.

Additionally, we propose an alternative architecture where both parts of the model act in parallel, as seen in Figure 4.6b. In this method, the non-parametric part aims to learn the residuals of the parametric part and then adds them to its prediction. This model allows computational parallelisation with CUDA, which the MPPI algorithm already utilises heavily. We call this architecture the **parallel semi-parametric model**.

4.5.2 Experiments

Now that we have established our new models, we can compare them to the standalone parametric and non-parametric models. For the following experiments, we used common data and the same hyper-parameters where applicable. For all parametric models, we used the dynamic bicycle with tyre brush model. For the non-parametric models, we used simple and stable hyper-parameters:

- batch and epoch size of 100
- 2 hidden layers of 32 neurons each and biases
- tanh activation function (other options are not explored due to the results in Section 4.4.3 of [Georgiev, 2019])
- Adam optimiser
- 10^{-3} learning rate
- 10^{-5} weight decay coefficient

- data normalisation

However, in order to test the generalisation and accuracy benefits of the semi-parametric model, we have to modify the validation process such that the models are evaluated on unseen state data. For this process, instead of randomly selecting train, validation and test, we now order the data in ascending values and select only a part of it. This is visually explained in Figure 4.7.

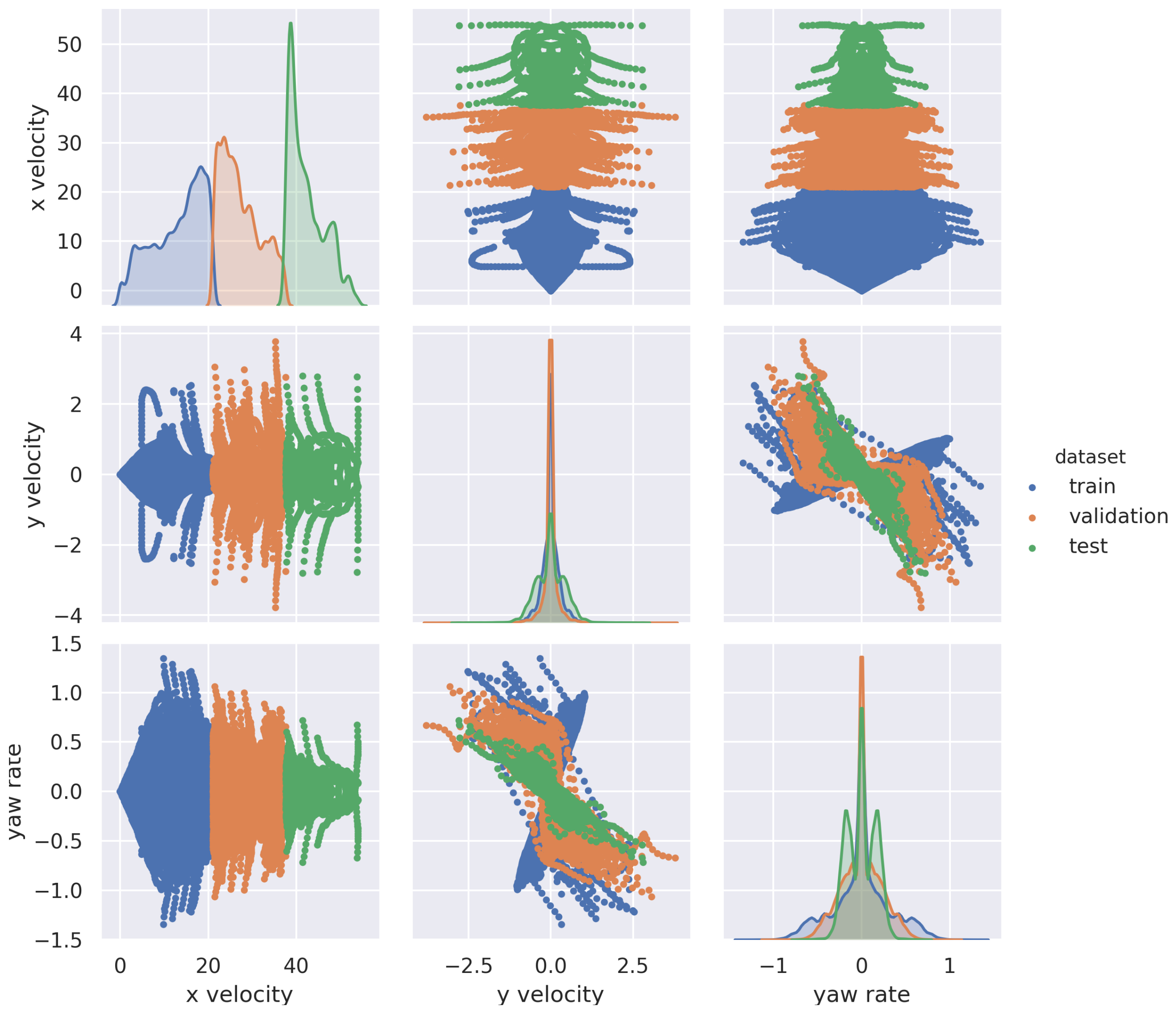


Figure 4.7: Example of how the dataset is split into train, validation and test sets such that they occupy a different part of the overall state space. In this particular example, the data is divided according to the x velocity field, which can be seen in the top left plot. In this plot the train, validation and test datasets have 60%, 35%, 5% proportions. It is worth noting that these proportions are not accurately represented in the diagonal KDE plots.

Now that we have means of validation, we can run the model training experiments for all four different models with only the core dynamic variables and a 60%, 35%, 5% train, validation and test dataset split. The results are shown in Figure 4.8. From them, we can deduce a couple of interesting insights:

- The non-parametric model initially manages to achieve good validation loss but then swiftly overfits the data to the extent where the simple parametric model outperforms it.
- Both semi-parametric models are slightly faster to train than the non-parametric model. This is a desirable effect that makes the model more responsive and applicable to online learning applications.
- The cascade semi-parametric model consistently outperforms all other models in generalisation, making it ideal for real-world applications where it is feasible to explore the full state space. This model also seems not to be prone to overfitting, something that we will further investigate. However, unsurprisingly the cascade semi-parametric model also performs slightly worse than the non-parametric model in train loss.
- The parallel semi-parametric model behaves strangely in terms of validation loss. The reason for this remains largely unexplored due to the adequate performance of the cascade semi-parametric model.

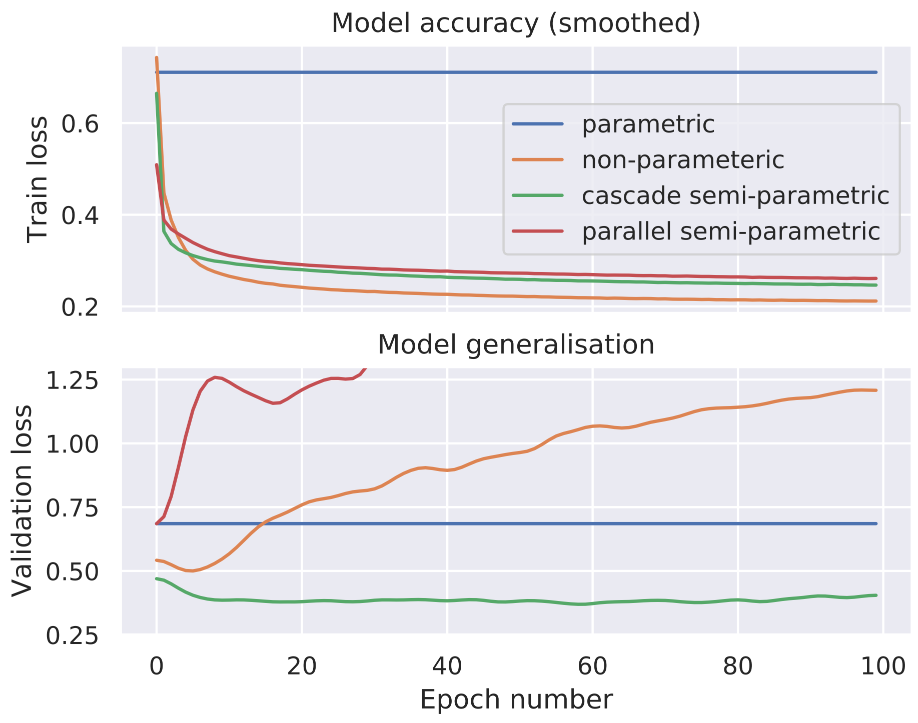


Figure 4.8: Accuracy and generalisation comparisons between all four different models presented in this report. The first plot shows model accuracy, which is evaluated on the training dataset. Lower values here are generally better but might also signify overfitting. Then the model generalisation capabilities are shown when applied to a validation dataset which contains state data that the model has not seen previously. Here we are most interested in consistently low error which shows excellent generalisation capabilities. Note that the parametric model does not change with epoch number as it is fitted to the full dataset only once.

The experiments above show one exciting quality of the cascade semi-parametric model - it refuses to overfit similar to the other models. As this is so desirable, we chose to explore this further and confirm the behaviour by running the same experiment for 1000 epochs. From the results shown in Figure 4.9, we can see that the cascade model fluctuates in validation loss but never overfits for the duration of the experiment. In contrast, the non-parametric model overfits very swiftly. This characteristic of the cascade semi-parametric model makes it a good candidate for iterative learning which will be further explored in Section 4.7.

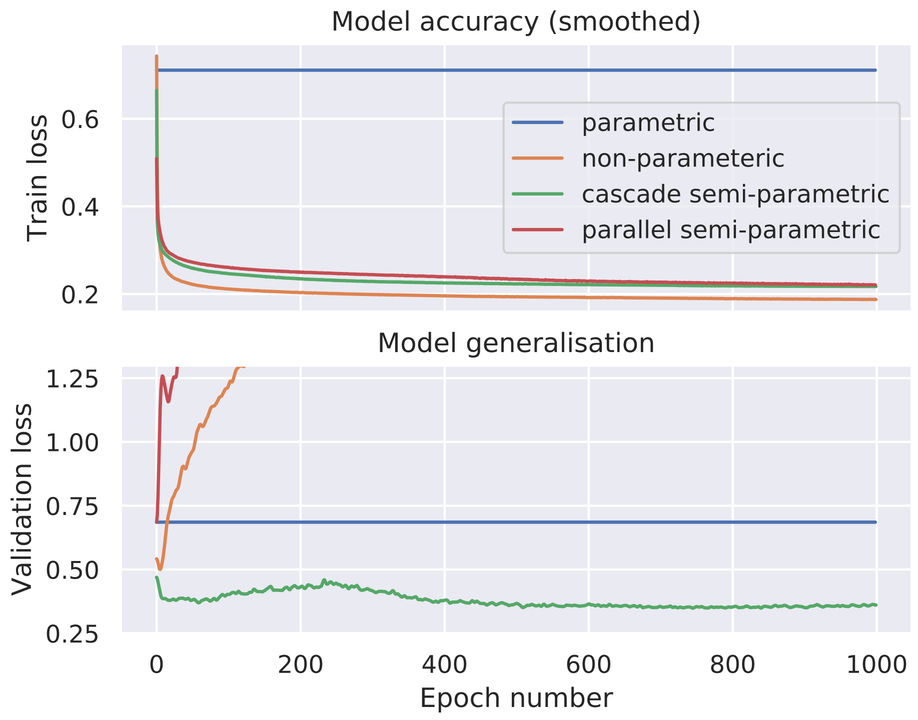


Figure 4.9: The same experiment as Figure 4.8 but run for 1000 epochs. This allows the models to overfit to the train dataset.

4.5.3 Hyper-parameter search

With the good results of the cascade semi-parametric model, in this section, we proceed to find good hyper-parameters for its non-parametric part. As the parametric model is now responsible for a good portion of the prediction of the dynamics, we can afford to have a smaller non-parametric part. Due to that and computational restrictions, we chose to explore hidden layer sizes of (16,16), (24,24) and (32,32) and learning rates of 0.0025, 0.001, 0.00075 and 0.0005. The other hyper-parameters are shown below, and the results are shown in Table 4.8.

- batch size of 100 and epoch size of 1000
- tanh activation function (other options are not explored due to the results in Section 4.4.3 of [Georgiev, 2019])

- Adam optimiser (other optimisers are not explored due to the results in Section 4.4.2)
- 10^{-5} weight decay coefficient
- data normalisation only for the non-parametric part
- 70%, 20%, 10% train, validation and test dataset split

Hidden layer size	Learning rate	Validation MSE	Average lap time	Lap time std.
16,16	0.0025	1.4217	-	-
16,16	**0.001**	**1.3875**	**124.8235**	**0.2437**
16,16	0.00075	1.4254	125.0624	0.1327
16,16	0.0005	1.4339	-	-
24,24	0.0025	1.1909	126.2583	0.8365
24,24	***0.001***	***1.1766***	***120.6480***	***0.3267***
24,24	0.00075	1.1848	121.9842	0.4841
24,24	0.0005	1.1787	123.8647	0.2187
32,32	0.0025	1.2137	-	-
32,32	0.001	1.1682	125.6762	0.6672
32,32	0.00075	1.1670	121.9273	0.2475
32,32	**0.0005**	**1.1628**	**-**	**-**

Table 4.8: Semi-parametric training experiments with different hidden layer sizes and learning rates. All experiments are run using the cascade semi-parametric model with normalised inputs $\dot{x}, \dot{y}, \dot{\theta}, \ddot{y}$. Average lap times are estimated over five consecutive laps. A failure is designated if the car span or went off track; in that case, lap times are omitted from the table. The best hyper-parameters for each model size is bolded. The model which has achieved the best lap times is italicised.

We can see that the italicised model with hidden layer sizes 24,24 achieves best results in both validation MSE and lap time performance. However, we are still not finished with identifying hyper-parameters. The semi-parametric model differs from the non-parametric model as in that it learns not the full model, but only the residuals leftover from the parametric part. As such, one would expect the weights of such a model to be smaller. Thus, here we consider one more important hyper-parameter - the weight decay coefficient which is responsible for the regularisation of the neural network. In the case of this project, we use L2 regularisation due to its prior availability and excellent results. It is incorporated into the loss function of the network and penalises it for having big weights with the idea of improved generalisation:

$$w_{t+1} = w_t - \gamma \nabla_w J - \lambda w_t \tag{4.9}$$

where γ is our learning rate and λ is the weight decay coefficient. The higher this coefficient is, the more regularisation will take effect, improving generalisation capabilities but also reducing accuracy. We put this to the test with the previously identified best semi-parametric model and show the results in Figure 4.10. From there we can identify two good values of the parameter: $\lambda = 10^{-4}$ produces the best result on the validation dataset but $\lambda = 10^{-3}$ produces more stable results as training error plateaus. For the remainder of the report, we will use the latter model for its superior stability.

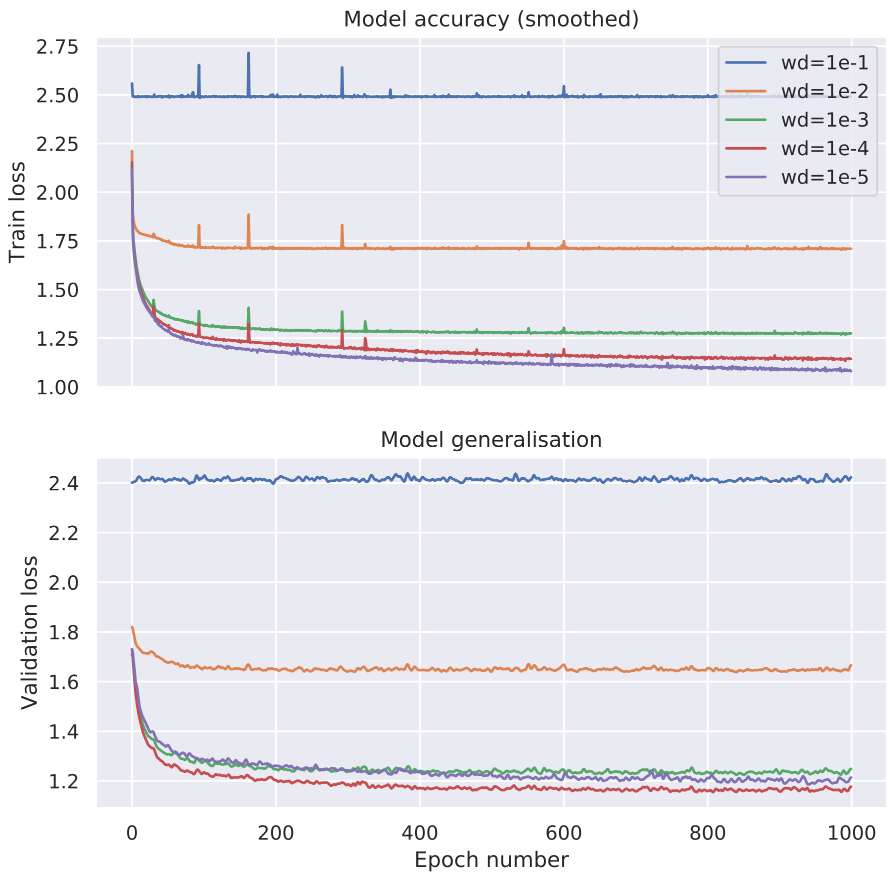


Figure 4.10: Comparison between different weight decay coefficients.

4.6 Model comparison

Here we take the best of all three models developed earlier and compare them using the same control parameters used so far in this chapter (Appendix C.2). However, instead of only testing them on the same racecourse, we also compare their performance on three different racetracks, effectively testing the model generalisation capabilities (for track details see Appendix D). The experiments were conducted using the same control parameters as detailed in Appendix C.2; the only changes between experiments are the track and dynamics model. The experiments results are shown in Table 4.9.

The results are not surprising and we can see three tendencies:

Track	Model	Average lap time	Lap time std.
Simulation	P	122.65	0.4120
	NP	119.33	0.2588
	SP	120.65	0.3267
ORN3	P	98.06	0.0408
	NP	93.98	0.0449
	SP	94.40	0.0897
Modena	P	111.78	0.0659
	NP	108.23	0.0705
	SP	109.02	0.0554

Table 4.9: Comparison of the performance of different models. Experiments were run using the same parameters but on different tracks and using different models. All results shown are over the span of five laps, and all units are in SI. P, NP and SP stand for parametric, non-parametric and semi-parametric models respectively.

1. The non-parametric model consistently produces the best results confirming its superiority given enough data.
2. The semi-parametric model on average performs slightly worse than the non-parametric model, offering a middle-ground between the non-parametric and parametric models.

4.7 Incremental learning

As is natural with all physical systems, dynamic properties change over time due to a plethora of reasons. Examples include long-term effects such as the wear out of joints and mechanical links, tyre wear, loss of magnetisation in electric motors but also more short-term effects such as a slippery surface, wind disturbances or even weight changes. As such, ideally, models used in model-based control should adapt to these changes online; however, this is a difficult problem and is usually referred to as *Incremental Learning* [van de Ven and Tolias, 2019].

In this project, both the non-parametric and semi-parametric models utilise neural networks which unfortunately are notoriously bad in incremental learning applications due to the so-called *catastrophic forgetting* issue [McCloskey and Cohen, 1989]. However, as we saw in the previous section, the semi-parametric model offered superior generalisation capabilities which are connected to the forgetful nature of neural networks. Thus, in this section, we will examine if the semi-parametric model is suitable for incremental learning applications.

In designing this system, we had two goals in mind (1) the model must adapt to changes swiftly but also (2) not wholly ignore prior dynamics. The justification for this is that the system must adapt to changes fast but also be resilient to short-term disturbances such as wind gusts. With these goals in mind, we researched into state of the art in the field of incremental learning, which has developed several methods for dealing with the forgetful nature of neural networks. The most popular techniques include training only a randomly selected part of the network on new data, regularising the update weights and replay strategies (also known as *pseudo-data*) [van de Ven and Tolias, 2019]. The latter uses a technique where as well as letting the model learn on new data, it is also trained on the old data in parallel (i.e. the old data is being replayed). The appeal of this is clear, but it has two limitations - the old training data is not necessarily correct anymore, and it is impossible to store all previous data in memory. Shin et al. developed a method to address both of these issues called *Deep Generative Learning (GDR)* [Shin et al., 2017] where they keep a large, complicated model and train it on new and old data; however the old data is not stored but generated online using generative methods[2]. This approach has seen tremendous success in practice, and we consider it a right direction for the work of this report. Furthermore, Grady Williams et al. released work where they adopted a similar approach to incremental learning with MPPI [Williams et al., 2019].

Although the systems mentioned above are best suited for this task, we did not have time to implement them in the context of this project. Instead, we chose to approximate them with similar architecture but different tools. We developed a system which trains the non-parametric part of the semi-parametric model from two sources: the local operating set and a rolling buffer of dynamics data over a more extended time. Additionally, to tackle short-term disturbances, we also introduce a constrained gradient optimisation [Williams et al., 2019], which ensures that the new update cannot move away from the previously seen data. To formalise this system mathematically, we introduce two data distributions:

$$\text{Local Operating Distribution: } X_L, Y_L \sim \mathcal{D}_L \tag{4.10}$$
$$\text{Rolling Buffer Distribution: } X_B, Y_B \sim \mathcal{D}_B \tag{4.11}$$

Online, data is sampled from both and used to compute their own independent gradients for the stochastic gradient descent procedure in the neural network optimisation:

$$G_L = \nabla_{\theta_i} ||Y_L - f(X_L, \theta_i)||^2 \tag{4.12}$$
$$G_B = \nabla_{\theta_i} ||Y_B - f(X_B, \theta_i)||^2 \tag{4.13}$$

where $f()$ is the prediction, θ are the parameters and ∇ is the derivative operator. Next, we compute constraint the gradient and use it to update the parameters:

$$\begin{aligned} \theta_{i+1} &= \theta_i - \gamma(\alpha G_L + G_B) \\ \alpha &= \underset{a \in [0,1]}{max} \text{ s.t. } \langle a G_L + G_B, G_B \rangle \geq 0 \end{aligned} \tag{4.14}$$

[2]Generative models are unsupervised learning methods which aim to learn the true data distribution with the aim of afterwards generating new data points with some variation.

where γ is the learning step and $\langle\cdot\rangle$ is an inner-product operation. This update law balances the objective of simultaneously optimising the model with data from both sets without allowing them to contradict each other. Finally, the data from the local operating dataset is fed into the local buffer $\mathcal{D}_B$ and used for subsequent optimisations. However, we found it beneficial not to include data which achieved $\alpha < 0.4$ as this data is drastically different from the previously seen data. An additional visual explanation of this approach is shown in Figure 4.11. It is important to note that the model is bootstrapped by using a pre-trained model and that the rolling buffer must pre-loaded with the data used to train the pre-trained model. Apart from the newly-introduced constrained gradient optimisation, the neural network learning procedure and parameters must remain the same as the ones used to train the model initially. Here, we used the same Python framework used throughout this chapter. Finally, there are a couple of parameters to set:

1. The size of the local operating set. As the optimisation is run when the local operating set is full, its size determines how swiftly the model adapts. However, if this is made too small, then the model becomes susceptible to outlier data.

2. The size of the rolling buffer set. This determines how much data the model ”remembers” which must be a function of the size of the local operating set and also significantly larger than it. In general, we found that a rolling buffer 100x the size of the local operating set works well in practice.

3. The number of epochs to train for each time the local operating set fills up. This determines how well the model trains but also how badly it can overfit. Luckily, due to the nature of the semi-parametric model used here, overfitting is not an issue, and we left the iterative learning system to train for 10 epochs without any noticeable problems.

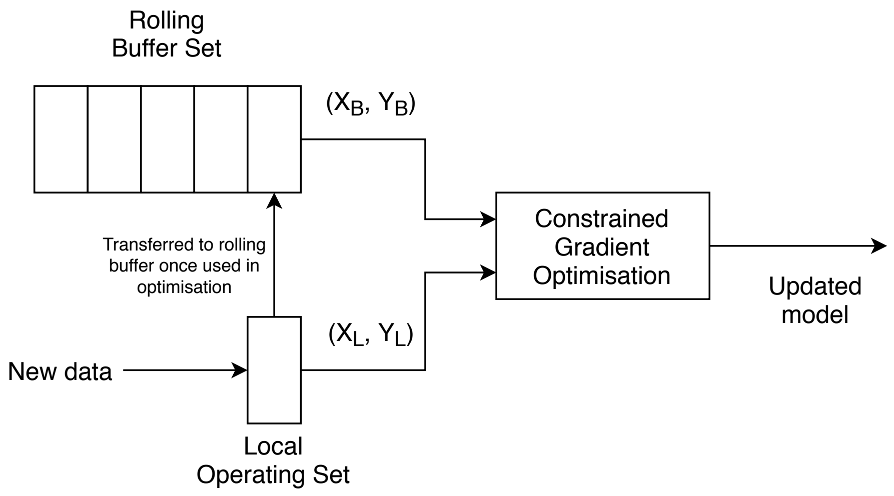

Rolling Buffer Set
(X_B, Y_B)
Transferred to rolling buffer once used in optimisation
Constrained Gradient Optimisation
Updated model
(X_L, Y_L)
New data
Local Operating Set

Figure 4.11: The online learning system visualised. New data comes from the operation of the vehicle and is stored in the local operating set. When the local set is filled up, data is sampled from both it and the rolling buffer set and used to update the model via the constrained gradient optimisation. Afterwards, if the local operating set data is not an outlier; it is transferred to the rolling buffer set.

To verify our approach to iterative learning, we consider two experimental cases. (1)

long-term effects where the environment has changed since the initial model learning and (2) short-term effects where the environment is dynamically changing at operation time for short time instances.

4.7.1 Long-term experiments

Here we use the trained semi-parametric model from the previous section and apply it to the ORN3 track; first with the same simulation as used to learn the dynamics; we call this the **normal** scenario. Then we run another experiment with a simulation which has the mass of the vehicle changed from 1350kg to 1600kg and the friction coefficient μ from 1,0 to 0,9; we call this the **modified** experiment. Control parameters for this were chosen to be safe ones to ensure that all models complete the course successfully (the full list can be found in Appendix C.3). These two experiments serve as the lower and upper bound of performance we should expect. Finally, we run our iterative learning system using the same model as the previous two runs as bootstrapping and the dataset used to train the model as the initial rolling buffer. Empirically we found that a local operating set of 1000 data entries (20 seconds of driving) worked well and resulted in a rolling buffer of 10^5 data entries. We are keeping training at ten epochs, as stated previously produced the results shown in Figure 4.12.

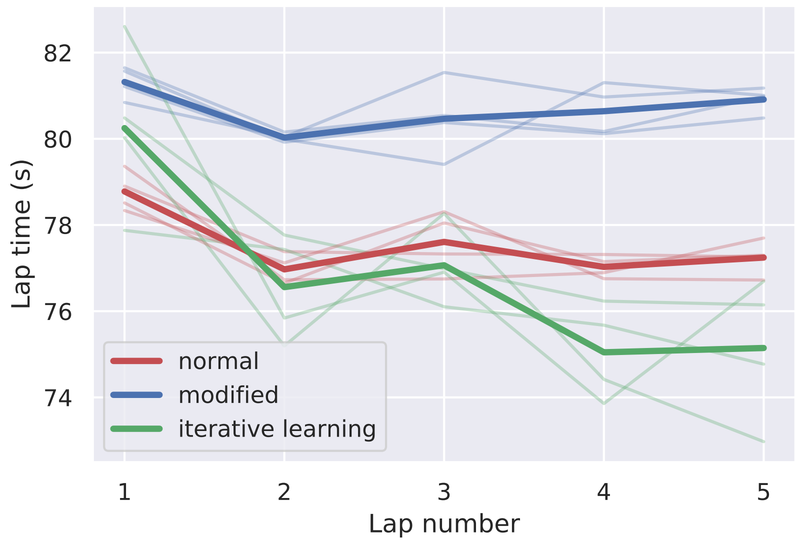


Figure 4.12: Consecutive times from three different experiments averaged over four trials. The independent tests are shown with transparent lines, and the solid lines are their averages. The normal experiments are from applying a trained model in the same environment it was trained in. Modified is when the model is applied to a changed environment and iterative learning is again within the modified environment, but this time with the model adapting to the changes online. It is worth noting that even in the cases where the models are fixed, the lap times are not identical due to the stochastic nature of the MPPI algorithm. Additionally, in all experiments, the performance in the first laps is slightly worse than average as the car starts from a standstill 20 meters before the start line.

From these results, we can see that iterative learning is working successfully and adapts to the changing dynamics. Furthermore, it consistently outperforms the *normal* experiments which were previously considered the upper bound of performance. This could be because the model fits more specifically to the current racetrack or because the modified dynamics allow for faster lap times or maybe a combination of these two hypotheses. However, there is also a negative effect - the iterative learning model becomes progressively more unstable. This is best shown via the variance between the different runs, as seen in Table 4.10. This makes the model not ideal for real-world use but still produces superior results.

Lap number	Normal	Modified	Iterative Learning
1	0.455	0.368	1.940
2	0.342	0.099	1.238
3	0.707	0.873	1.195
4	0.252	0.589	1.098
5	0.400	0.298	1.659

Table 4.10: Standard deviation between lap times over four trails

4.7.2 Short-term experiments

In this subsection, we want to short-term variable dynamics as typical in the real world. Two example cases were considered: one where a part of the road is slippery and one where a gust of wind increases drag (air resistance) in a part of the track. We created an artificial track to recreate this where a 300m segment of the track was with a reduced friction coefficient from 1.0 to 0.8 and a second segment of the track 300m long where the drag coefficient increased from 1.225 to 1.5. Similar to the long-term experiment above, we first set the two baselines for performance and then run the iterative learning model. Different to the previous experiments now is that the model must adapt quickly, and for that reason, we are forced to use a smaller local operating set. In this case, we experimented with sizes of 200, 400, 600 (4s, 8s, 12s of driving respectively) and kept the previously successful size of 1000. Sadly none of the experiments produced positive results. The experiments with smaller local operating sets fail to finish a single lap as their models deteriorate as soon as they enter the modified segments and the experiment with the size of 1000 fails to adapt fast enough to the changed regions. A plot of the successfully finished models is seen in Figure 4.13. Thus, we conclude that the iterative learning system developed here does not work for short-term disturbances. Sadly, due to time limitations alternatives were not explored.

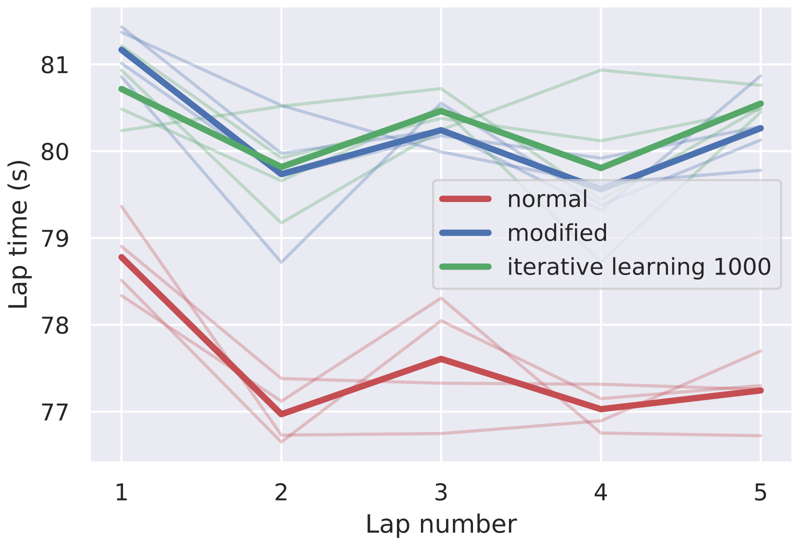


Figure 4.13: Consecutive times from three different experiments averaged over four trials. The independent trials are shown with transparent lines, and the solid lines are their averages. The normal experiments are from applying a trained model in the same environment it was trained in. Modified is when the model is applied to a changed environment and iterative learning is again within the modified environment, but this time with the model adapting to the changes online. It is worth noting that even in the cases where the models are fixed, the lap times are not identical due to the stochastic nature of the MPPI algorithm. Additionally, in all experiments, the performance in the first laps is slightly worse than average as the car starts from a standstill 20 meters before the start line.

Chapter 5

MPPI Enhancements

5.1 Trajectory tracking

As shown in Section 2.3.2, the MPPI algorithm does open-loop optimisation and as such, produces a trajectory consisting of sequences of states and controls. This is also known as *motor tape*. When an approach like this is applied to a continuous environment where the state estimates can be received at a higher rate than the optimisation rate, several problems start to surface.

The first issue is highlighted in Figure 5.1, where there is no trajectory tracking happening, commands are simply executed open-loop. Due to issues such as modelling error, external disturbances and optimisation issues, the trajectory commands are executed, but the optimised trajectory is not achieved. Instead, the actual trajectory is far from the optimal one.

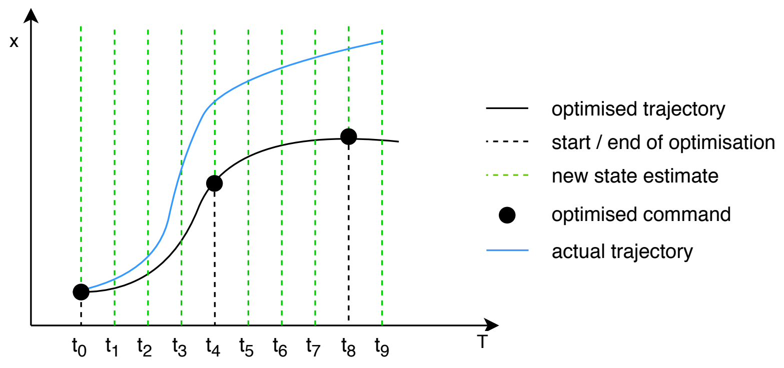

x
T
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
optimised trajectory
start / end of optimisation
new state estimate
optimised command
actual trajectory

Figure 5.1: A simplified 1-D trajectory tracking example. At each timestep, a state estimate is received, and a new control can be executed. However, the trajectory optimisation process has a timestep five times higher than the rate at which the state estimates are received. Thus, in this case, at each timestep, the last optimised command is executed.

This issue can be resolved by taking better actions in-between the optimisation timesteps. The simplest way of doing this is by linearly interpolating the state and controls between optimisation timesteps, as seen in Figure 5.2. With this, we now have an estimate of what commands can be given to the robot in-between optimisation timesteps. However, this does still not ensure trajectory tracking and is still an open-loop solution.

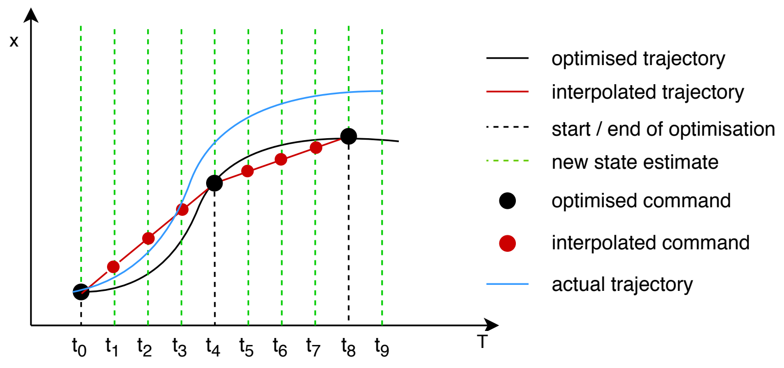


Figure 5.2: A simplified 1-D trajectory tracking example. States and controls are now linearly interpolated, which provides us better commands at each timestep at which we receive state feedback.

To solve the issue of trajectory tracking, we have to introduce feedback. This is done by effectively adding an intermediate lower-level controller. The two most popular feedback controllers are Proportional Integral Derivative (PID) and Linear Quadratic Regulator (LQR). Here we have chosen the latter for its superior dynamic stability and for the fact that the model from MPPI can also be used to obtain a linearised model for the LQR.

5.1.1 LQR Background

Linear Quadratic Regulator (LQR) is a state-feedback controller which provides the optimal feedback gains for closed-loop stabilisation of a dynamical system. It operates on linear system dynamics subject to

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u} \tag{5.1}$$

and a quadratic cost function with Q and R being the cost coefficients:

$$\mathcal{L} = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} + \frac{1}{2}\mathbf{u}^T R\mathbf{u} \tag{5.2}$$

The LQR produces a gain matrix K which works on the full state feedback to produce the best controls $\mathbf{u} = K(\mathbf{x}_{desired} - \mathbf{x})$. Abstracting ourselves from the maths, it is only necessary to provide the controller with the $A \in \mathbb{R}^{S\times S}, B \in \mathbb{R}^{C\times C}, Q \in \mathbb{R}^{S\times S}$ and $R \in \mathbb{R}^{C\times C}$ matrices where S and C are the state and control dimensions [Alexis, 2012].

5.1.2 LQR Design

The difficult part of designing an LQR controller for this project is handling the dynamics correctly. Since a car's dynamics are highly non-linear (after all that is the reason why MPPI exists), it is impossible to linearise the full dynamics. However, since the LQR will be working in a small time window (<20ms) it is possible to approximate the dynamics linearly. Traditionally this is done using 1st order Taylor series expansion about the reference point $(\mathbf{x}_0, \mathbf{u}_0)$:

$$f(\mathbf{x},\mathbf{u}) \approx f(\mathbf{x}_0,\mathbf{u}_0) + \underbrace{\frac{\delta f(\mathbf{x},\mathbf{u})}{\delta \mathbf{x}}}_{A}(\mathbf{x}-\mathbf{x}_0) + \underbrace{\frac{\delta f(\mathbf{x},\mathbf{u})}{\delta \mathbf{u}}}_{B}(\mathbf{u}-\mathbf{u}_0) \tag{5.3}$$

To apply it to this project, we need to compute the A and B matrices for the three different dynamics models introduction in Chapter 4. To maintain consistency all models, simplify the development needed and still keep correct comparisons between the three models, we chose to calculate the A and B matrices using *automatic differentiation* [Griewank et al., 1989]. This was implemented uniformly for all models using the Eigen library[1].

The process mentioned above worked in practice but needs to be executed on the CPU after every optimisation, which quickly became a bottleneck for the MPPI algorithm. Since the kinematic part of all of the models used throughout this project is the same for all models, we chose to also fix it in the process of obtaining the LQR matrices. Referring back to the kinematic state $\mathbf{x}^k$ from Equation 2.10, we can manually calculate the derivatives both with respect to $\mathbf{x}$ and $\mathbf{u}$. Then, we leave the dynamics part of the state transition to be calculated by the automatic differentiation process, and we finally get our A and B matrices below. Note that that lateral acceleration $\ddot{y}_v$ was excluded due to its noisy nature.

$$A = \begin{pmatrix} 0 & 0 & -\dot{x}\sin\psi - \dot{y}\cos\psi & \cos\psi & -\sin\psi & 0 \\ 0 & 0 & \dot{x}\cos\psi - \dot{y}\sin\psi & \sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ & & & \nabla_{\mathbf{x}^d} f & & \end{pmatrix} \tag{5.4}$$

$$B = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \multicolumn{2}{c}{\nabla_{\mathbf{u}^d} f} \end{pmatrix} \tag{5.5}$$

However, one thing is missing from the matrices above - the fixed point $(\mathbf{x}_0, \mathbf{u}_0)$. Since we are only interested in the linearised dynamics at the current timestep, we can choose our fixed point to be the last optimisation timestep. Then we can continue recomputing the LQR gain K at every optimisation step to ensure that its dynamics are accurate as possible.

Finally, we have to choose appropriate cost matrices. We found the following ones to work well from empirical results:

[1]Eigen is a C++ library for linear algebra

$$Q = \begin{pmatrix} 1.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.0 & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & 0.75 & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & 0.2 & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.1 \end{pmatrix} \tag{5.6}$$

$$R = \begin{pmatrix} 80 & 0 \\ 0 & 4 \end{pmatrix} \tag{5.7}$$

Results
Identical experiments were run with and without this tracking while maintaining all other configurations identical. The results are shown in Table 5.1. This trajectory tracking seems to improve performance and stability of the car (shown via the standard deviation). This topic is not explored further as it is not that relevant in the simulated environment. However, it will become more important in a real-world application.

Tracking	Lap time avg.	Lap time std.	Max slip angle	Max lateral acceleration
No	88.17	2.14	0.024	9.66
Yes	87.05	1.00	0.0577	10.58

Table 5.1: Effects of trajectory tracking on overall performance. Lap time is averaged over five consecutive laps. The same dynamics model and parameters are used across both experiments.

5.2 Variable sampling

In the original formulation of the MPPI Algorithm [Williams et al., 2016] control signals were standardised into $[-1,1]$ unit range and sampled using a normal distribution. If sampling controls for a sequence $\mathbf{u}_t$ then we would do that according to

$$\mathbf{u}_t \sim \mathcal{N}(\mathbf{u}_{t-1}, \sigma) \tag{5.8}$$

where σ is a predefined static sampling variance. This approach has one major limitation - it assumes that the full control range is available at all times. However, this is not always the case. This is best explained visually in Figure 5.3. The strategy used so far works well for the case where the car is driving at 10 m/s as the full control range is feasible. However, at a speed of 40 m/s only a subset of the control range is feasible. Here we define control feasibility as the desired control can be reached and results in dynamically stable behaviour.

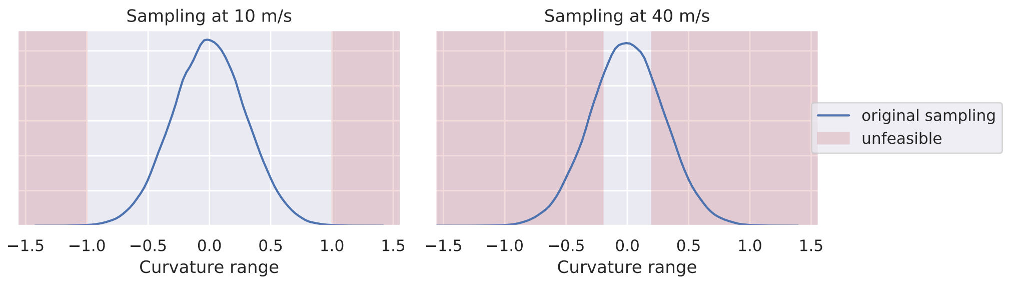


Figure 5.3: Curvature control sampling in MPPI. Note that the curvature is scaled up to uniform values $[-1, 1]$.

For example a curvature command of 0.5 at a speed of 40 m/s is technically possible but would result in the car sliding. As such that control option is considered unfeasible.

In the case of a fixed sampling distribution and the car driving at a speed of 40 m/s, a large portion of the samples would end up being unfeasible. This translates to less useful sampling and in general, a non-optimal end trajectory. This is not an issue in the case of acceleration commands as the full range is available at all times there.

To make trajectory sampling more efficient, we can limit sampled controls to be within a range we know is feasible. For curvature, the limits can be found with the formula [Christoforos Chatzikomis, personal communication, 2020]:

$$\kappa_{max} = \frac{\ddot{y}_{max}}{\dot{x}^2} \tag{5.9}$$

where $\ddot{y}_{max} \approx 13.734$ according to the vehicle parameters from Table 2.1. We can calculate this value dynamically at run time and clip curvature samples online. This is shown in the left plot of Figure 5.4. However, this clipping results in a biased sampling distribution towards the limits, which in term results in inefficient trajectory sampling. To improve this, we propose a dynamic sampling scheme where the sampling distribution is adjusted based on the current limits. In the case of a normal distribution, the standard deviation is taken as 30% of the control limit as seen in the right-hand plot of Figure 5.4. However, this is still not perfect as the mean of the normal distribution can be close to the limits, resulting again in bad sampling. A more robust alternative for curvature sampling would be a uniform distribution.

Experiments

Experiments were run using the same parameters, model and costs. Varied were the choice between sampling - uniform or Gaussian and the amount of variance in the latter. Note that for uniform sampling, possible controls are sampled across the whole range. Final results are shown in Table 5.2 and a full list of parameters can be found in Appendix C.4.

A couple of interesting insights can be gained from these experiments:

1. Uniform and high variance normal distributions do not work for acceleration sampling. In all of the failed experiments in these configurations, the issue was identical

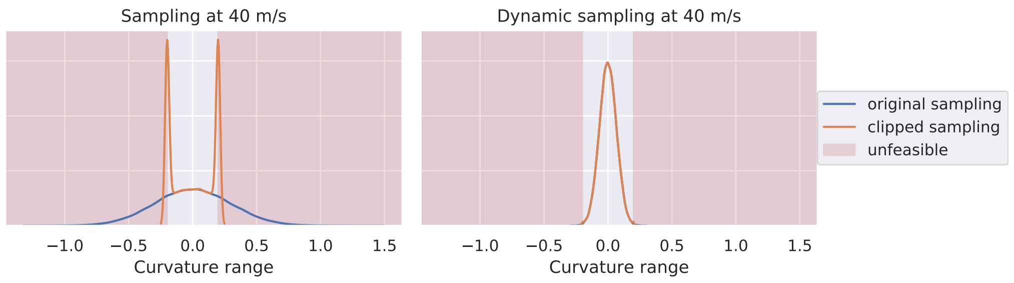


Figure 5.4: Curvature control sampling in MPPI. Here samples are clipped at limit values. Note that the curvature is scaled up to uniform values $[-1,1]$.

- the car failed to slow down before a corner. An intuitive explanation can be offered - due to the relatively sparse sampling, the algorithm does not sample enough around the optimal region and as such fails to find the optimal trajectory.

2. On the other extreme, low variance normal distributions for both acceleration and curvature result in poor results or complete failures. The explanation for this behaviour is opposite to the one above - the algorithm does not explore sufficiently and is unable to find the optimal trajectory. However, this results in fewer failures in comparison to the previous case, as the algorithm can still fall back to the previous solution.

3. Uniform sampling in curvature produces good results and lap times with the lowest standard deviation. This translates to stability which is a desirable characteristic. Unlike acceleration, the curvature range is far smaller and affects dynamics less (especially at high speeds) and as such the algorithm benefits from more exploration in this control dimension.

Based on the above, we can draw two conclusions (1) that the algorithm achieves the best results when there is sufficient but not extreme acceleration exploration and the full curvature range is explored and (2) lower exploration in general results in more stable but less optimal control. Thus, we propose two sampling options to the reader:

1. For stable control - normal sampling with a standard deviation of 0.6 for acceleration and uniform sampling for curvature

2. For best lap times - normal sampling with a standard deviation of 0.6 for both control inputs.

Acceleration std	Curvature std	Avg. lap time	Min lap time	Std. lap time
Uniform	Uniform	Fail		
1	Uniform	Fail		
0.6	Uniform	86.92	86.49	0.9523
0.5	Uniform	86.05	85.53	0.8587
0.4	Uniform	86.12	85.64	0.8453
0.3	Uniform	86.95	86.43	0.7430
0.2	Uniform	Fail		
Uniform	*	Fail		
1	*	Fail		
0.6	1	85.73	84.87	1.3114
0.6	0.7	85.78	85.71	1.3362
0.6	**0.6**	**85.54**	**84.83**	**1.3043**
0.6	0.5	86.51	85.17	1.9014
0.6	0.4, 0.3, 0.2	Fail		
0.5	1	85.88	85.18	1.3011
0.5	0.7	85.83	85.23	1.2335
0.5	0.6	85.82	85.05	1.2700
0.5	0.5	85.89	85.10	1.2264
0.5	0.4	85.95	85.37	1.1742
0.5	0.3, 0.2	Fail		
0.4	1	85.83	84.93	1.3685
0.4	0.7	85.68	84.84	1.3922
0.4	0.6	85.95	85.29	1.2104
0.4	0.5	85.95	85.30	1.2495
0.4	0.4	86.01	85.40	1.2432
0.4	0.3	86.48	85.94	1.1376
0.4	0.2	Fail		
0.3	1	87.29	86.30	1.2458
0.3	0.7	86.83	86.16	1.2715
0.3	0.6	86.65	85.86	1.3046
0.3	0.5	86.47	85.78	1.2351
0.3	0.4	86.52	85.87	1.2482
0.3	0.3	86.78	86.02	1.2654
0.3	0.2	87.49	86.62	1.5157

Table 5.2: Experiment results from varying control sampling. The first two columns show the standard deviation if Gaussian sampling is used. If uniform sampling is used, then it overrides the standard distribution number. Lap time results are shown on the three columns to the right averaged over a total of 5 laps. If the car went out of the track or slid during any lap, then the experiment is labelled as a failure. The standard deviation in lap time is an indication of the stability of the controller. The best average lap time is made bold.

Chapter 6

Optimising lap time

In this setting of optimal control, the optimality is defined by the cost function. As such, setting an adequate cost function for achieving the best possible lap time is of high importance. The overall objective in the case of this project is to achieve the best possible lap time while ensuring not to go outside of the boundaries of the course and not crash. Note that here we consider only the case in which the car is driving by itself on a race track.

As shown in Section 2.1, the overall cost of a trajectory is given by:

$$J(\mathbf{x}_0) = \mathbb{E}_{\mathbf{x}} \left[\phi(\mathbf{x}_T) + \int_{t_0}^{T} \mathcal{L}(\mathbf{x}_t, \pi(\mathbf{x}_t)) dt \right] \tag{6.1}$$

As this project tackled a receding horizon trajectory optimisation problem, there is no trajectory termination, and as such, we set the terminal cost $\phi(\mathbf{x}_t) = 0$. That leaves us only with the instantaneous cost (also known as cost-to-go) $\mathcal{L}$. Simplifying the notation with $\pi(\mathbf{x}_t) = \mathbf{u}_t$, we formulate our cost as

$$\mathcal{L}(\mathbf{x}, \mathbf{u}) = \alpha_1 \mathcal{L}_{track}(\mathbf{x}) + \alpha_2 \mathcal{L}_{control}(\mathbf{u}) + \alpha_3 \mathcal{L}_{speed}(\mathbf{x}) + \alpha_4 \mathcal{L}_{slip}(\mathbf{x}) \tag{6.2}$$

In the following sections we dissect each term of the cost function.

6.1 Track cost

The track cost is the component which provides spatial information to the optimisation procedure. For this project, that was chosen to be a **costmap** in the form of a 2D grayscale image as inspired by the original work on MPPI in [Williams et al., 2017]. Each pixel of this costmap represents the cost of the car is in that position. The optimal position would have a cost of 0, and the cost gradually increases to a value of 1 signifying the least optimal position. This particular strategy of feeding spatial information to the optimisation procedure was chosen as it is a nice fit for the parallel architecture of MPPI because of which, it is necessary to evaluate the track cost timesteps $\times$ rollouts $\times$ run rate many times each second. A typical example used in this project is $100 \times 2560 \times 50 = 12800000$ reads per second. The costmap approach is also chosen due to the Nvidia GPU hardware architecture, which supports fast access to image-like data.

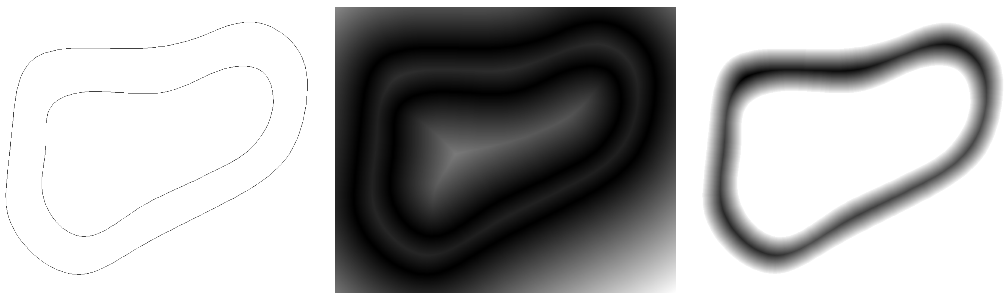
Costmap generation
As explained in Section 2.2.3, we have access to the track boundaries, middle line and racing line. These are read by a standalone node and fed into a Python ROS node used to generate a costmap online. Finally, the costmap is sent to the core MPPI algorithm, dynamically loaded into GPU memory and ready for starting the algorithm. In comparison to our previous work where costmaps were offline generated images that were manually loaded, this new approach is more portable and integrated with the existing structure (see Figure 3.4 for integration). Additionally, to reduce the number of optimisations needed to reach the overall optimal solution, we chose to assign the lowest cost to the racing line instead of the centre line. This reduces the exploration of the trajectory optimisation procedure but ensures that the algorithm converges to a solution faster.

The process of generating the costmap is similar to our previous work in [Georgiev, 2019] but is summarised here as well for completeness:

1. Track bounds 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.
2. Since images have only positive values in their coordinates, the track boundaries and middle line are shifted so that they are centred in the image coordinate frame.
3. A blank white image with the correct dimensions is created.
4. The pixels corresponding to the track boundaries are set to 255. Figure 6.1a.
5. A Huber distance transform[1] is then performed which results in smooth gradients extending outwards from both track boundaries. Figure 6.1b.
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. The costmap is now finished - Figure 6.1c.

Track cost calculation
Now that we have the costmap, we can calculate the track cost $\mathcal{L}_{track}$. This is done in the same fashion as in our previous work - by taking the values of the costmap from the four corners of the car and its centre and averaging them. This is best explained visually in Figure 6.2. Additional experiments were run with track cost calculations only at the centre of the car and average from the front and rear axles. All three methods produced similar performance results, but the method chosen in Figure 6.2 is the most stable one as it forces the car to stay within the track boundaries.

[1] `https://en.wikipedia.org/wiki/Huber_loss`



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

Figure 6.1: The process of creating a costmap. Source: [Georgiev, 2019]

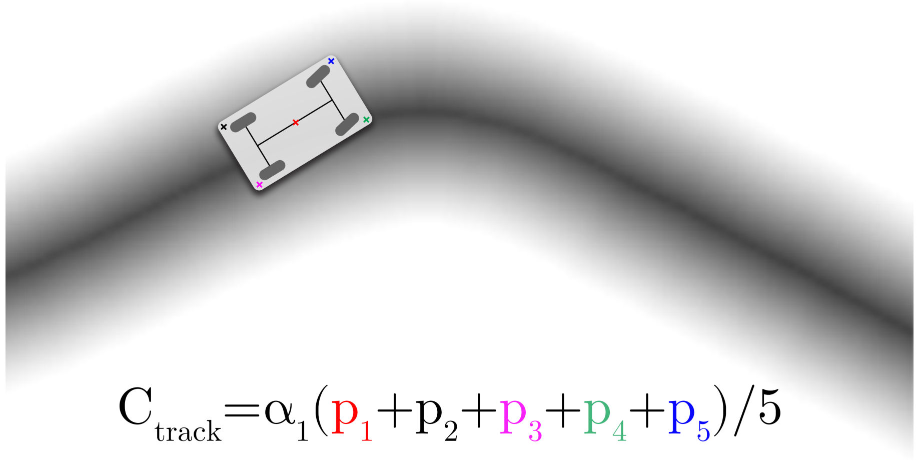


Figure 6.2: Dimensional calculation of track cost.

Costmap optimisation

A limitation of this costmap approach where the optimiser struggled with larger tracks. This is because the costmaps are stored in the GPU texture memory[2] which is limited. For this project, this issue was tackled by reducing the memory footprint of the costmap in two ways:

1. The costmap was reduced from a 4-channel image to a single-channel image.
2. The costmap which originally was stored as floating-point numbers is now dynamically converted to 8-bit unsigned integers if it is too big for the GPU texture memory.

These improvements reduced the memory footprint of the costmap by 16 times without

[2]Nvidia GPUs have a special texture memory which is made for fast 2D accessing. This is usually used for graphical applications.

compromising performance. This was a satisfactory solution for this project, but the overall costmap idea should be revised for a more widely applicable solution.

6.2 Control cost

The control cost remains as defined in [Williams et al., 2017]:

$$\mathcal{L}_{control}(\mathbf{u}) = \frac{\mathbf{u}_t - \mathbf{u}_{t-1}}{(\mathbf{v}_t - \mathbf{u}_t)^2} \tag{6.3}$$

where $\mathbf{u}$ is the control and $\mathbf{v}$ is the noisy control. Increasing this cost will force the controller to change the control sequence less rapidly and as a result, stabilise trajectories. A detail omitted for brevity here is that the control cost coefficients are split independently for the two control signals.

6.3 Speed cost

Both in the original work on MPPI in [Williams et al., 2016] and in our previous work in [Georgiev, 2019], the speed cost is the only method of encouraging a fast trajectory. In both works, this was done using a fixed desired forward speed $\dot{x}_{des}$:

$$\mathcal{L}_{speed}(\mathbf{x}) = (\dot{x}_v - \dot{x}_{des})^2 \tag{6.4}$$

In this sense, the optimisation objective is to optimise speed, whereas the real optimisation objective is minimising lap time. The authors in [Williams et al., 2016] argue that maximising speed is the same as minimising time via the relationship $t = S/V$. It is true that t and V are inversely proportional but minimising one does not necessarily mean maximising the other as S could change in-between. This is best explained visually in Figure 6.3.

With these findings, it is best to reformulate the cost of this section to minimise lap time; however, this is not trivial in a receding horizon problem as in this project. There is one successful implementation of such an objective in a classical numerical optimisation MPC [Lam et al., 2010]. However, the strategy adopted there is possibly too computationally intensive for MPPI. Due to that and time constraints, we chose to still optimise speed, which when constrained with the racing line costmap chosen in Section 6.1 should result in a close approximation to a minimising lap time objective.

However, this manifests itself in another issue - a single desired velocity produces an optimisation problem too complex for real-time optimisation. Consider a case where the desired speed is set to 70 m/s, but the car is entering a corner where the maximum achievable velocity without crashing is 15 m/s. The optimisation process would struggle to randomly sample enough trajectories to bring the car to an acceptable speed to enter the corner. In this sense, a fixed desired speed is not suitable for this problem. Instead, we chose to leverage the pre-existing dynamic path planner (see Section 2.2.3), which outputs a trajectory with desired velocity, acceleration and curvature. We use the velocity given by the path planner to set a desired velocity for the MPPI optimisation procedure while still

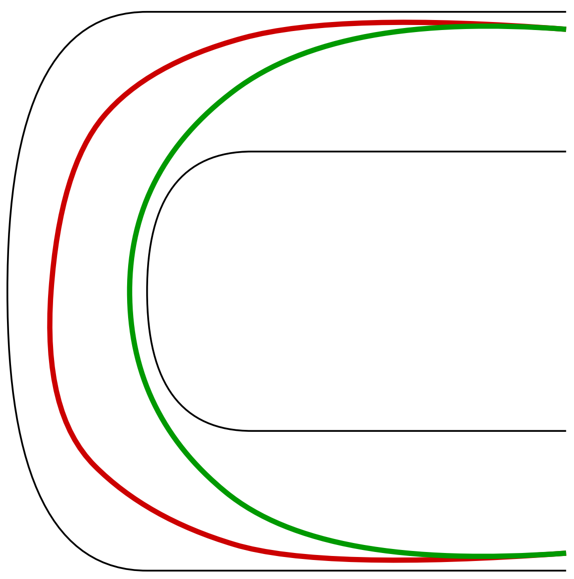


Figure 6.3: A comic illustrating the difference in cornering with different optimisation objectives. The green path shows an example where the minimisation objective is time, and the red path shows a case where the optimisation objective is maximising speed. Both paths are correct in their sense but produce different results. The green path achieves a lower speed and time. The red path achieves higher speed, but also higher distance and higher time.

utilising Equation 6.4. In particular we map the speeds given in the path planner path to the appropriate timesteps of the MPPI optimisation. The integration of this can be seen back in Figure 3.4 and brings us full circle back to the original motion control concept shown in Figure 2.1. Now we have the three components of it - the path planner, trajectory optimiser (MPPI) and trajectory tracker (Section 5.1).

6.4 Slip cost

Another form of increasing the stability of the optimised trajectory is by penalising the car for sliding. This is done via the slip angle first introduced in Section 4.3 and defined as

$$\mathcal{L}_{slip}(\mathbf{x}) = \beta = \arctan \frac{\dot{y}_v}{|\dot{x}_v|} \tag{6.5}$$

As the slip angles of the car are in radians, they have low numerical values, and as such a high slip cost parameter should be used.

6.5 Cost tuning

With the somewhat complex function laid-out above, we are left with a total of five cost parameters to tune. This can be done manually, harvesting expert knowledge but also proving inefficient and time-consuming. Due to the stochastic nature of MPPI several laps must be completed with a single set of parameters in order to confirm performance. For example, if we target five laps of an average of 120 seconds each, then each experiment will take 10 minutes. If we are to run 100 experiments with different cost parameters, that would result in more than 16 hours of continuous testing. Instead, we chose to look into automated parameter tuning. The more popular classical approaches are Grid Search and Random Search [Bergstra and Bengio, 2012]. The first automatically evaluates a set of predefined permutations of possible parameters and then returns the ones that achieved the best possible results. Random search operates in a similar fashion, but instead of user defined search-spaces, it randomly samples possible parameters. Both of these methods have been proven to work in practice, however, their major limitations are that they blindly search parameters, not taking into account previous experience, and that they take a long time to optimise the parameters. An alternative approach is to use a Bayesian framework which attempts to minimise an objective function by building a probabilistic model based on past evaluations. Here we chose to use the Tree of Parzen Estimator (TPE) [Bergstra et al., 2011] due to its available Python implementation in the Hyperopt library [Bergstra et al., 2013].

Using that approach, we created a Python wrapper around the simulation and the MPPI algorithm, defined the ranges of all cost parameters as seen in Table 6.1. We then ran the optimisation for 100 iterations on the Simulation track and got the optimal parameters as seen in the table.

Parameter	Name	Search range	Optimised value
α_1	Track cost	[1;500]	372
$\alpha_{2,k}$	Curvature control cost	[0;10]	2.91
α_{2_a}	Acceleration control cost	[0;10]	1.43
α_3	Speed cost	[0.1;4]	2.78
α_4	Slip cost	[0;1000]	18.57

Table 6.1: Cost parameters used. The search range is used for the Bayesian parameter optimisation.

6.6 Final results

Finally, here we compare the best lap times obtained from our work until now. The only comparable results are from the prior motion control system utilised by Roborace. It consists of the path planner and stabilisation layer as described throughout this report (Sections 2.2.3 and Chapter 3) where the path planner outputs a desired trajectory and the stabilisation layer has an additional path matching functionality which tracks the desired trajectory. Using this approach, results were obtained for three different configurations as seen in Table 6.2.

The final experiment results can be found in Table 6.3 where our approach outperforms the previous solution in configurations #1 and #2 but is beaten in configuration #3 which tests performance on the limit. Multiple possible explanations for this exist with the most likely one being the still not ideal generalisation capabilities of the semi-parametric

Configuration	Max. speed	Long. acceleration	Long. deceleration	Lat. acceleration
#1	41.67 (150 kph)	4.9 (0.5 g)	6.867 (0.7 g)	11.772 (1.2 g)
#2	55.56 (200 kph)	4.9 (0.5 g)	7.848 (0.8 g)	11.772 (1.2 g)
#3	69.44 (250 kph)	4.9 (0.5 g)	8.829 (0.9 g)	12.753 (1.3 g)

Table 6.2: Configurations for different experiments. These affect how aggressive the car is to drive. All units are in SI by default.

model combined with the lack of near limit driving data as in experiment configuration #3. However, due to time limitations and the difficult comparison between the two approaches, the reasons for this are not explored and we are satisfied with the performance obtained in these experiments. It is also worth noting that applying the iterative learning system in this case, did not yield superior results as the vehicle simply never reached the near-limits state space and thus failed to explore it. This can be seen on the GG plot from the experiments in Figure 6.4. The full list of parameters for this experiment can be found in Appendix C.5 and a supplementary video can be found online[3].

Configuration	Stabilisation layer	MPPI
#1	89.73	80.23
#2	77.55	76.94
#3	74.79	76.42

Table 6.3: Comparison between the lap time performance between RoboraceDriver control solution and MPPI with the semi-parametric approach developed in this project. The trials were carried out on the simulator track. MPPI experiments are averaged over 10 different trails due to its stochastic nature.

[3]`https://youtu.be/aLWXa95AEKE`

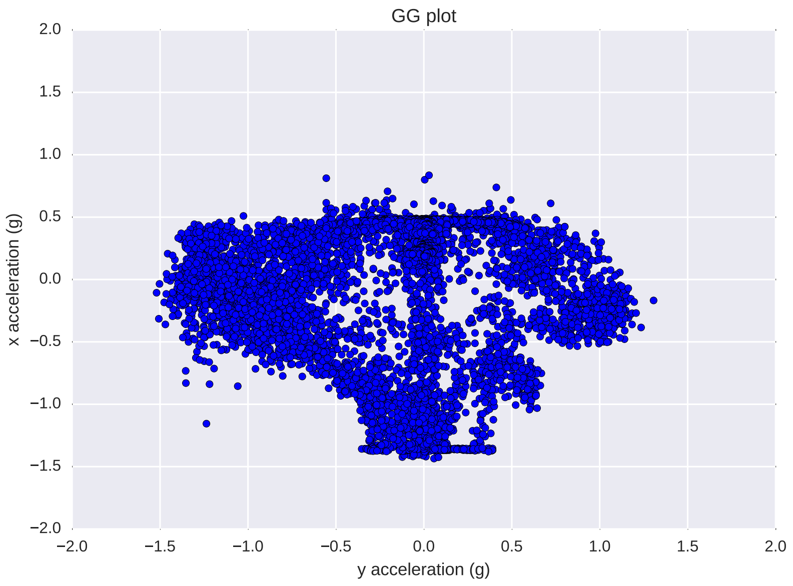


Figure 6.4: A GG plot of the best experiment results. This kind of plot shows lateral (y) vs longitudinal (x) acceleration in g $(9.81m/s^2)$. This plot shows how well the control algorithm manages to reach the limits of performance of the vehicle which can be imagined by fitting a symmetric oval along the plot. The more of that oval that is filled up, the more aggressive the performance is. In this case, the top half of the plots shows very high performance, reaching more than 1.3g lateral acceleration, however, the bottom half indicates that the algorithm does not enter corners at the high enough speeds to result in peak performance.

Chapter 7

Conclusions

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.

7.1 Summary

Recent years have seen the rise of popularity of self-driving cars which are slowly becoming the first instance of massive adoption of mobile robot technology. A subfield of that is aggressive driving which aims to control the vehicle to the limits of its capabilities in challenging environments. In turn, this has the potential to contribute to the safety of read-legal driverless cars.

This project focuses on autonomous car motion control in a racing scenario, a direct continuation of our previous work in [Georgiev, 2019] where we utilised the novel Model Predictive Path Integral (MPPI) controller to drive a small racecar in simulation. This is an unconventional approach which instead of numerically optimising a trajectory around a given reference, it samples multiple possible control sequences and then cost-wise averages, resulting in the optimal sequence. 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 is described using the path integral framework [Williams et al., 2017]. Robot control with this method opens a new frontier of possibilities as path integrals are derivative-free and can theoretically use more complex models. We exploited this in our previous work by successfully using a neural network as the vehicle dynamics model but also acknowledge its limitations.

In this report, we expand on the modelling capabilities by aiming to build an iterative learning system for the vehicle dynamics capable of adapting to a changing environment or modified dynamics. However, on the road to this, we also develop further contributions and create an integrated and deployable motion control system for the Roborace DevBot V2.

The first step towards this goal was to develop an adequate simulator. Harnessing prior simulators, we developed a standalone ROS-based motion control simulation which strikes a balance between vehicle model accuracy, ease of use and automation. We model this after the real vehicle and use it to integrate all of the work carried out in this project into the existing systems.

After we set the foundations for development, we proceed with the central part of this project - vehicle modelling where we lay out the concepts of parametric, non-parametric and semi-parametric models. We develop each individually and compare them, showing that the non-parametric model is more accurate than the parametric one but generalises badly to unseen states. On the other hand, the semi-parametric model which combines both of the previous approaches is shown to offer the best of both accuracy and generalisation capabilities. Furthermore, that model also indicates that it is resistant to overfitting, making it ideal for real-world applications. We next used the semi-parametric model to develop the iterative learning system which attempts to adapt the model at run-time utilising a buffer for old dynamics and a constrained gradient optimisation to ensure compatibility between the old and new data. We develop specialised tests in simulation and show that this approach is successful in adapting to long-term dynamical changes such as tyre wear but fails in doing so with short-term dynamical changes such as a slippery part of the road.

We additionally improve the MPPI algorithm by introducing trajectory tracking where instead of only sending control in an open-loop fashion, it now also interpolates the nominal trajectory and maintains it using an LQR controller. This is a unique approach as it leverages the same vehicle model as used by the core algorithm by automatically differentiating it to obtain its linearised form. This makes the trajectory tracking applicable to any model. Furthermore, to make control sampling more efficient, we also introduce variable sampling to one of the commands, making the sampling process aware of the dynamical limitations in the current state and exploring only the feasible states.

Next, we acknowledge the limitations of our previous work in the cost function of the optimisation process. We extend it for better vehicle stability and also reformulate the main optimisation objective to be approximately lap time instead of raw speed by harnessing the racing line of each track and a path outputted from an external path-planner. Then we use a Bayesian parameter optimiser to find the cost coefficients resulting in the best lap time and compare the final performance of our approach to an existing system.

Finally, we evaluate the peak performance of the algorithm and compare it to the previous approach used, showing that its performance is similar but not better. However, our method can optimise trajectories online and adapt to varying dynamics, theoretically making it more suitable for real-world applications. However, due to time constraints, those are not performed.

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

- Understanding of the fundamentals of motion control and the requirements for a real-world application.
- Appreciation for classical approaches for their simplicity, understandably and ease of use.
- A deep understanding of dynamics modelling and the trade-offs between different models and approaches.

- The importance of taking a critical view on ones own work.

7.2 Discussion

The most important point to touch on is the application of the MPPI algorithm. Its high degree of modularity and scalability offers evident advantages over more classical MPC solutions. Contradicting the stated in our previous work [Georgiev, 2019], this approach is not a complete motion control solution but can only solve part of the problem. MPPI can be a drop-in replacement for trajectory optimisation; it can be used as a path planner or even a trajectory tracker, making it a flexible tool. However, with this flexibility, it also suffers from an *identity crisis*. The algorithm can be seen as a solution to many problems, yet we do not know which problems it can solve best as none of the previous work in the area has compared it to existing or classical solutions. In this report, we used as a trajectory optimiser and tracker, only to find out that is did not outperform previous, more conventional, solutions.

The work carried out in this project with vehicle modelling brings the learned dynamics approach to motion control one step closer to real-world applications. The semi-parametric model gives the user understanding of the dynamics model used within the MPPI algorithm and tackles the instability issues we found in non-parametric models in our previous work [Georgiev, 2019]. However, it does not entirely remove them, and there is always a probability that the learned dynamics are incorrect from some unexplored state space and can result in unwanted behaviour. As such, intentionally pushing this system (or anyone with learned dynamics) to an unexplored state-space involves a degree of risk, however small that can be.

Similar to the previous paragraph, the iterative learning system brings this approach into a more realistic light, allowing for simulation-to-real-world dynamics model transfer. A semi-parametric model can be pre-trained in simulation and then applied to a real vehicle and let to train online iteratively. This suffers from the same issues outlined in the previous paragraph but also makes the system less predictable and susceptible to exploding weights which neural networks are prone to. Thus, one should consider alternative non-parametric models as well.

As outlined in Section 5.2, the sampling scheme of MPPI is naturally flawed as it assumes Gaussian distribution and clips commands outside of a certain threshold. This manifests itself in inefficient exploration and thus optimisation, which can produce failures at run-time. In this report, we solved a significant part of this issue - the assumption of a constant distribution. However, the problem with clipping remains and makes the algorithm perform sub-optimally. Instead of just clipping the samples, a more informed sampling process should be developed, which is aware of the limits and samples controls more efficiently.

The final point of discussion we would like to raise is from a pragmatic viewpoint - the development process of MPPI. Due to its highly parallel nature, if needed to be applied to the real world, this approach must be implemented in the CUDA (or some other parallel programming language) which is has a high barrier of expertise, requires specialised hardware and is difficult to develop. Thus, the development process of MPPI can be complicated and slow. Alternatively, one can consider using a different environment for development purposes, but that has other limitations - what can be implemented sequentially might not run concurrently. In our experience, this issue forces the development in the native CUDA environment.

7.3 Future work

One of the significant limitations of our approach throughout this thesis has been the CUDA implementation of MPPI. For the author, parallel programming within the Nvidia ecosystem of this scale is still largely a mystery. At times we have managed to obtain a 4x boost in optimisation performance; at others, we have managed to break the algorithm altogether. However, one thing is clear - this approach can benefit from a more professional implementation which will in term improve the capabilities of the algorithm. Furthermore, a more generic implementation suitable for any robot will improve its popularity.

The iterative learning system developed in Chapter 4 was shown to work and successfully adapt to long term changes. However, due to the novelty of the idea and field, multiple possible extensions can be done to this system to improve its performance, usability and scalability. First, the buffering system we developed is not the best method of scaling this approach; instead, more robust tools should be used to maintain the ”remembering” capabilities of the system. Many excellent examples of this exist in iterative learning literature such as the GDR by [Shin et al., 2017] or the LWPR[2] system which has already been coupled with MPPI in [Williams et al., 2019]. Additionally, another significant limitation of the system we have developed here is that the parametric part of the semi-parametric model is fixed and not updated iteratively. In effect, the parametric and non-parametric part of the models can end up pinned against each other after a prolonged operation and severe changes to the dynamics. This issue should be alleviated by also updating the parametric part, which is not trivial but possible [Smith and Mistry, 2020]. Furthermore, neural networks are not the best candidates semi-parametric models due to their bad generalisation capabilities and catastrophic forgetting. We only used them in this project due to the existing CUDA implementation. However, further research into semi-parametric models for MPPI should look into more stable models such as Bayesian Neural Networks [Blundell et al., 2015] or Gaussian Process Regression [Nguyen-Tuong et al., 2009].

Finally, we believe that the optimisation objective for this project should be fundamentally reformulated. Instead of approximating the fundamental objective of minimising lap time with speed and track costs (Chapter 6), one should look to directly formulating the main objective into the cost function. Furthermore, we believe that even though the costmap idea lends itself nicely to optimising the MPPI algorithm, it is not easily scalable to a real-world application in a typical road vehicle. Namely, it is impossible to describe the environment in a single image and fit it into a graphics card memory while maintaining fast memory access. Instead, further work on this should look into more natural ways of providing the algorithm with spatial information.

Bibliography

[Alexis, 2012] Alexis, K. (2012). Lqr control. `http://www.kostasalexis.com/lqr-control.html`.

[Bergstra and Bengio, 2012] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305.

[Bergstra et al., 2013] Bergstra, J., Yamins, D., and Cox, D. D. (2013). Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference*, pages 13–20. Citeseer.

[Bergstra et al., 2011] Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554.

[Blundell et al., 2015] Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*.

[Bojarski et al., 2016] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., et al. (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.

[Chatzikomis et al., 2017] Chatzikomis, C., Sorniotti, A., Gruber, P., Bastin, M., Shah, R. M., and Orlov, Y. (2017). Torque-vectoring control for an autonomous and driverless electric racing vehicle with multiple motors. *SAE International Journal of Vehicle Dynamics, Stability, and NVH*, 1(2017-01-1597):338–351.

[Chen and Huang, 2017] Chen, Z. and Huang, X. (2017). End-to-end learning for lane keeping of self-driving cars. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1856–1860. IEEE.

[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.

[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.

[Folkers et al., 2019] Folkers, A., Rick, M., and Büskens, C. (2019). Controlling an autonomous vehicle with deep reinforcement learning. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2025–2031. IEEE.

[Gao, 2014] Gao, Y. (2014). *Model predictive control for autonomous and semiautonomous vehicles*. PhD thesis, UC Berkeley.

[Georgiev, 2019] Georgiev, I. (2019). Path planning and control for an autonomous race car. `http://www.imgeorgiev.com/files/Ignat_MInf1_project.pdf`. The report from the first part of this project.

[Griewank et al., 1989] Griewank, A. et al. (1989). On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107.

[Hadekel, 1952] Hadekel, R. (1952). The mechanical characteristics of pneumatic tires. *Clearinghouse Fed Sci & Tech Info*.

[Hindiyeh, 2013] Hindiyeh, R. Y. (2013). Dynamics and control of drifting in automobiles. *PhD thesis, PhD Thesis*.

[Hollerbach et al., 2016] Hollerbach, J., Khalil, W., and Gautier, M. (2016). Model identification. In *Springer handbook of robotics*, pages 113–138. Springer.

[Kabzan et al., 2019] Kabzan, J., Hewing, L., Liniger, A., and Zeilinger, M. N. (2019). Learning-based model predictive control for autonomous racing. *IEEE Robotics and Automation Letters*, 4(4):3363–3370.

[Kendall et al., 2019] Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., and Shah, A. (2019). Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8248–8254. IEEE.

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

[Kong et al., 2015] Kong, J., Pfeiffer, M., Schildbach, G., and Borrelli, F. (2015). Kinematic and dynamic vehicle models for autonomous driving control design. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 1094–1099. IEEE.

[Lam et al., 2010] Lam, D., Manzie, C., and Good, M. (2010). Model predictive contouring control. In *49th IEEE Conference on Decision and Control (CDC)*, pages 6137–6142. IEEE.

[McCloskey and Cohen, 1989] McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

[Nguyen-Tuong et al., 2009] Nguyen-Tuong, D., Seeger, M., and Peters, J. (2009). Model learning with local gaussian process regression. *Advanced Robotics*, 23(15):2015–2034.

[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.

[Reza, 2008] Reza, N. (2008). Jazar. vehicle dynamics: Theory and application.-springer science.

[Shin et al., 2017] Shin, H., Lee, J. K., Kim, J., and Kim, J. (2017). Continual learning with deep generative replay. In *Advances in Neural Information Processing Systems*, pages 2990–2999.

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

[Smith and Mistry, 2020] Smith, J. and Mistry, M. (2020). Online simultaneous semi-parametric dynamics model learning. *IEEE Robotics and Automation Letters*.

[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.

[van de Ven and Tolias, 2019] van de Ven, G. M. and Tolias, A. S. (2019). Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*.

[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., 2019] Williams, G., Goldfain, B., Rehg, J. M., and Theodorou, E. A. (2019). Locally weighted regression pseudo-rehearsal for online learning of vehicle dynamics. *arXiv preprint arXiv:1905.05162*.

[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)*.

[Zorzi, 2014] Zorzi, M. (2014). Rational approximations of spectral densities based on the alpha divergence. *Mathematics of Control, Signals, and Systems*, 26(2):259–278.

[Zorzi and Chiuso, 2015] Zorzi, M. and Chiuso, A. (2015). A bayesian approach to sparse plus low rank network identification. In *2015 54th IEEE Conference on Decision and Control (CDC)*, pages 7386–7391. IEEE.

[Zorzi and Chiuso, 2017] Zorzi, M. and Chiuso, A. (2017). Sparse plus low rank network identification: A nonparametric approach. *Automatica*, 76:355–366.

Appendices

Appendix A

Supplementary figures

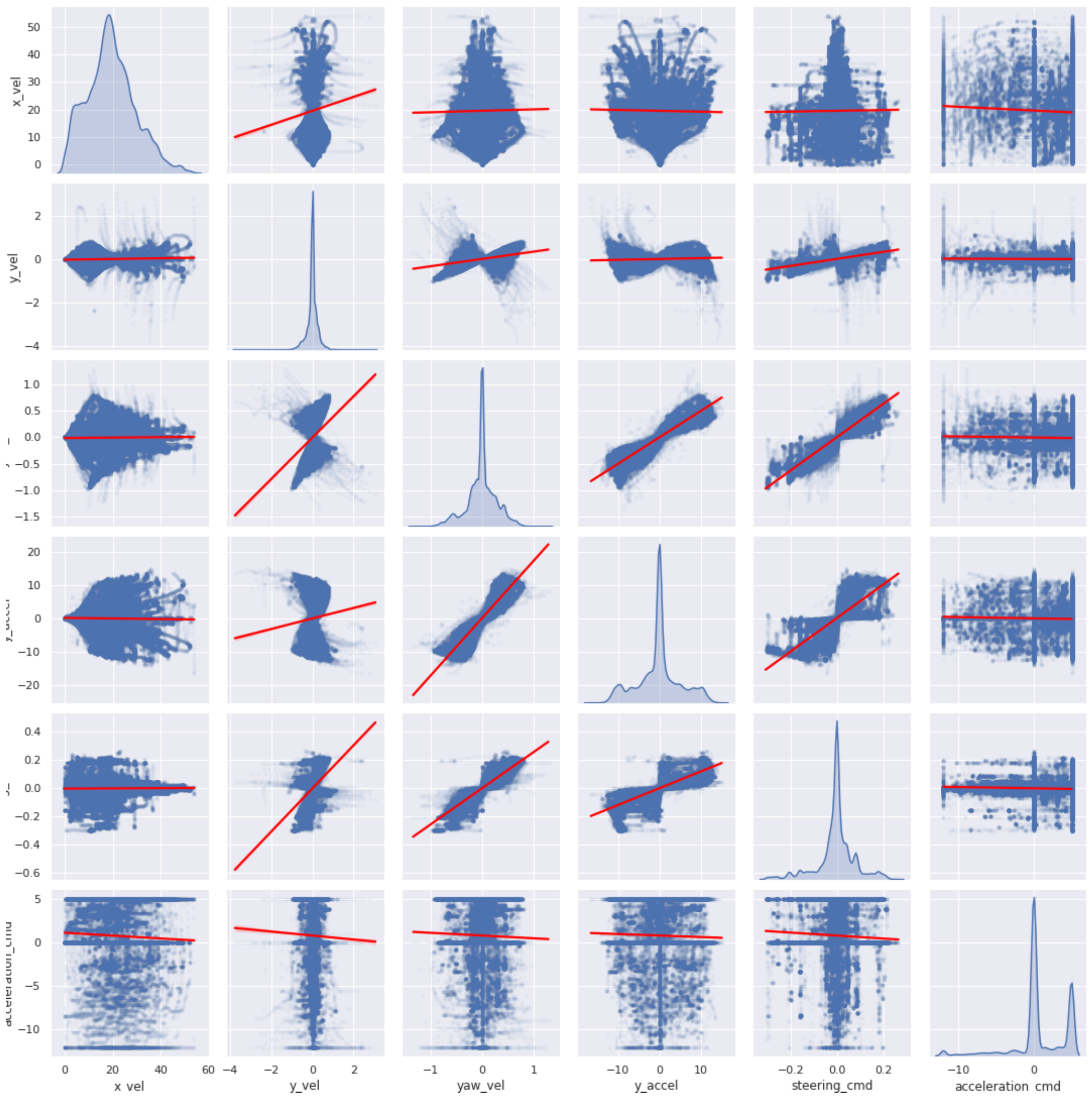


Figure A.1: Pairplot along all dynamics and control dimensions for an example dataset collected with the procedure in Section 4.1. The diagonal plots show estimated kernel density functions of each dimension. The off-diagonal plots are scatter plots with linear regressions fitted through them. This plot shows sufficient state exploration for the data collection procedure.

Appendix B

Fitted parametric models

The confidence is the estimated variance in the parameter by the fitting algorithm.

Parameter	Symbol	Fitted value	Confidence
Yaw inertia	I_{zz}	12966.22	1.14%
Front cornering stiffness	$C_{s,f}$	37405.74	0.18%
Rear cornering stiffness	$C_{s,r}$	74958.74	0.20%

Table B.1: Identified parameters for the dynamic bicycle with linear tyres model

Parameter	Symbol	Fitted value	Confidence
Yaw inertia	I_{zz}	4501.33	0.40%
Friction coefficient	μ	1.1526	0.10%
Front cornering stiffness	$C_{s,f}$	96420.96	0.23%
Rear cornering stiffness	$C_{s,r}$	208610.69	0.22%

Table B.2: Identified parameters for the dynamic bicycle with tyre brush model

Parameter	Symbol	Fitted value	Confidence
Yaw inertia	I_{zz}	4491.71	0.36%
	B_f	12.719	1.32%
	C_f	1.730	1.36%
	D_f	1.110	0.18%
	E_f	$\sim$0	10974.03%
	B_r	20.729	0.64%
	C_r	2.021	0.60%
	D_r	1.187	0.18%
	E_r	$\sim$0	1495011.92%

Table B.3: Identified parameters for the dynamic bicycle with magic tyre model

Appendix C

Experiment control parameters

This is a collection of different MPPI parameters used for running experiments throughout the thesis. The reader should first familiarise with the effects of the parameters.

C.1 Parameter guide

- Curvature variance - the variance of the normal distribution used for sampling the possible curvature commands. This parameter is ignored if uniform sampling is used. More details can be found in Section 5.2.
- Acceleration variance - the variance of the normal distribution used for sampling the possible acceleration commands. This parameter is ignored if uniform sampling is used. More details can be found in Section 5.2.
- Operating rate - the desired running rate in Hz of the MPPI algorithm which aims to achieve this as best as possible. This also designates the timestep Δt which is used for sampling the vehicle dynamics model and computing the next kinematic state of the vehicle. This scales linearly with computation.
- Number of timesteps - the number of Δt timesteps for the receding horizon trajectory optimisation. This scales linearly with computation.
- Map - a string pointing to an NPZ file containing the costmap. If this parameter is not given, then the algorithm waits to receive a dynamic costmap from an external source before starting. More details can be found in Section 6.1.
- Model - a string pointing to an NPZ file containing the model parameters. This is used throughout all three vehicle models introduced in Chapter 4 for their respective parameters.
- γ - softmax trajectory smoothing coefficient. Used to smooth out trajectories during the cost-wise averaging stage. More details can be found in 2.3.2.
- Trajectory tracking - boolean flag whether to enable the trajectory tracking as developed in Section 5.1.

- Reference speed - boolean flag weather to use the path planner path to determine the desired velocity.
- Desired speed - provides a single optimisation objective. Used only if reference speed is turned off.
- Track coefficient (α_1) - cost coefficient used to determine the importance of staying on the raceline. More information can be found in Section 6.1.
- Speed coefficient (α_3) - cost coefficient used to determine the importance of achieving the desired velocity. More information can be found in Section 6.3.
- Slip penalty (α_4) - cost coefficient for vehicle slip which penalises the car for drifting behaviour. More information can be found in Section 6.4.
- Maximum slip angle - maximum possible slip angle in radians. If the achieved slip angle is more than this, then the crash coefficient is imposed.
- Crash coefficient - cost imposed for the vehicle leaving the track boundaries or achieving a high slip angle.
- Curvature coefficient ($\alpha_{2,k}$) - control cost used to penalise the vehicle for executing highly-varying controls. More information can be found in Section 6.2.
- Acceleration coefficient ($\alpha_{2,a}$) - control cost used to penalise the vehicle for executing highly-varying controls. More information can be found in Section 6.2.
- Discount factor - used to discount costs of timesteps further away. This is set to 0 for all experiments within this project but is provided here for completeness and compatibility prior work.

C.2 Model comparison parameters

These are the MPPI controller parameters used for modelling experiments throughout Chapter 4.

- Curvature sampling var: 0.6
- Acceleration sampling var: 10.0
- Operating rate: 50 Hz
- Number of timesteps: 100
- Map: `simulator_test1_20ppm.npz`
- γ: 0.15
- Trajectory tracking: Off
- Reference speed: Off
- Desired speed: 16.0
- Speed coefficient: 1.0
- Track coefficient: 400
- Max slip angle: 0.75
- Slip penalty: 1000.0
- Crash coefficient: 20000.0
- Curvature coefficient: 0.0
- Acceleration coefficient: 0.0
- Discount factor: 0

C.3 Iterative learning parameters

These are the MPPI controller parameters used for the iterative learning experiments performed in Section 4.

- Curvature sampling var: 0.6
- Acceleration sampling var: 10.0
- Operating rate: 50 Hz
- Number of timesteps: 100
- γ: 0.15
- Trajectory tracking: Off
- Reference speed: On
- Speed coefficient: 0.5
- Track coefficient: 500
- Max slip angle: 0.75
- Slip penalty: 1000.0
- Crash coefficient: 20000.0
- Curvature coefficient: 1.0
- Acceleration coefficient: 0.0
- Discount factor: 0
- Path planner maximum velocity: 200 kph
- Path planner acceleration: 0.5g
- Path planner deceleration: 0.9g
- Path planner lateral acceleration: 1.2g

C.4 Sampling experiment parameters

This is the full list of parameters used for generating the results in Table 5.2.

- Model name: `system_id_net_2020-03-11T21:05:24.753646.npz`
- Map name: `simulator_test1_20ppm.npz`

- Operating rate: 50 Hz
- Number of timesteps: 100
- γ: 0.15
- Trajectory tracking: On
- Reference speed: On
- Speed coefficient: 1.0
- Track coefficient: 500
- Max slip angle: 1.0
- Slip penalty: 10.0
- Crash coefficient: 20000.0
- Curvature coefficient: 100.0
- Acceleration coefficient: 10.0
- Discount factor: 0
- Path planner max velocity: 250 kph
- Path planner longitudinal accel.: 0.5 g
- Path planner longitudinal decel.: 0.9 g
- Path planner lateral accel.: 1.3 g

C.5 Sampling experiment parameters

This is the full list of parameters used for generating the final results in Table 6.3.

- Model name: `system_id_net_2020-04-10T15:22:19.750725.npz`

- Map name: `simulator_test1_20ppm.npz`

- Operating rate: 50 Hz
- Number of timesteps: 100
- γ: 0.15
- Trajectory tracking: On
- Reference speed: On
- Speed coefficient: 2.8
- Track coefficient: 370
- Max slip angle: 0.1
- Slip penalty: 18.57
- Crash coefficient: 20000.0
- Curvature coefficient: 2.9
- Acceleration coefficient: 1.4
- Discount factor: 0

Appendix D

Race tracks

These are the racing courses used for the development of this project. All of these came with information on track boundaries, middle line and racing line. Note that this predates this project

Simulation track This is artificially created racing circuit which features continuous corners of various radii, tight chicanes and long straights aimed to test vehicles under different racing circumstances. This is the circuit used for all development aspects of this project.

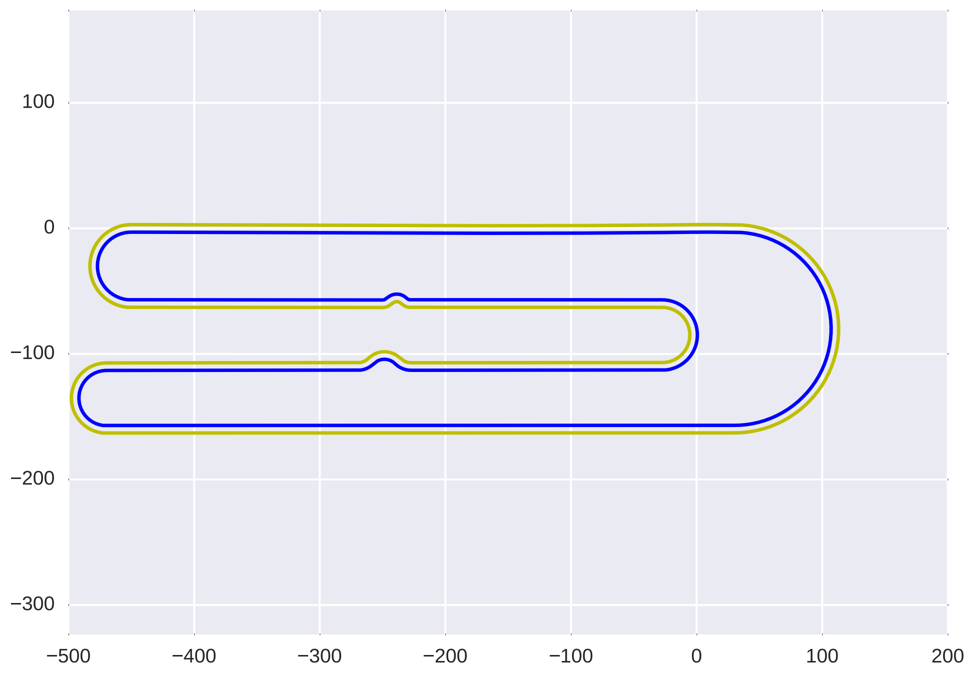


Figure D.1: Simulation track layout. Axes are in meters.

ORN3 circuit

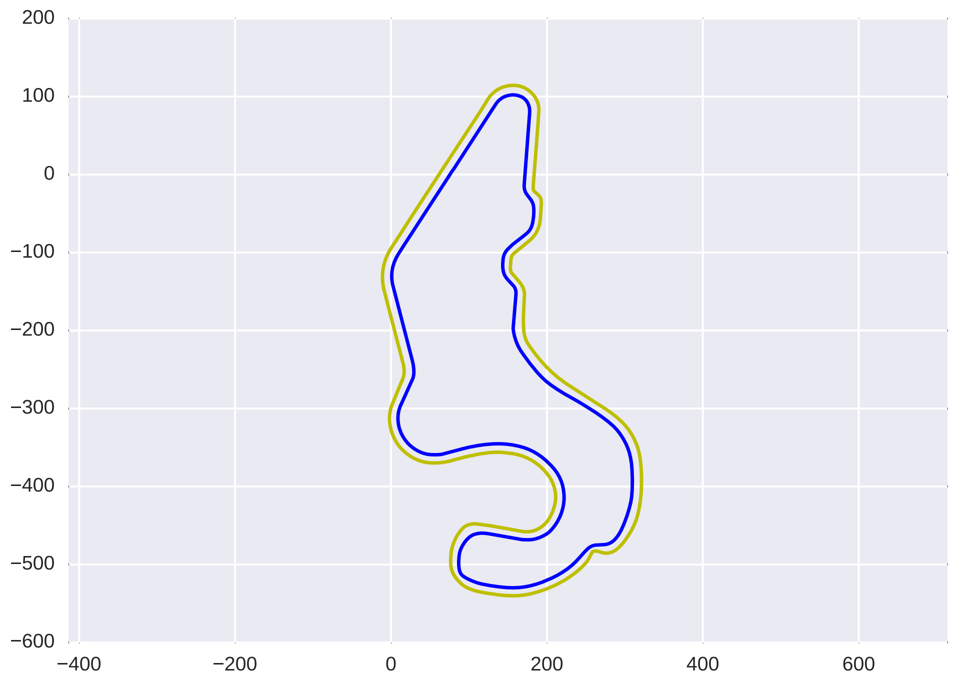


Figure D.2: ORN3 circuit layout. Axes are in meters.

Modena circuit A real circuit located near the city of Modena, Italy. The Modena circuit is of 3800m.

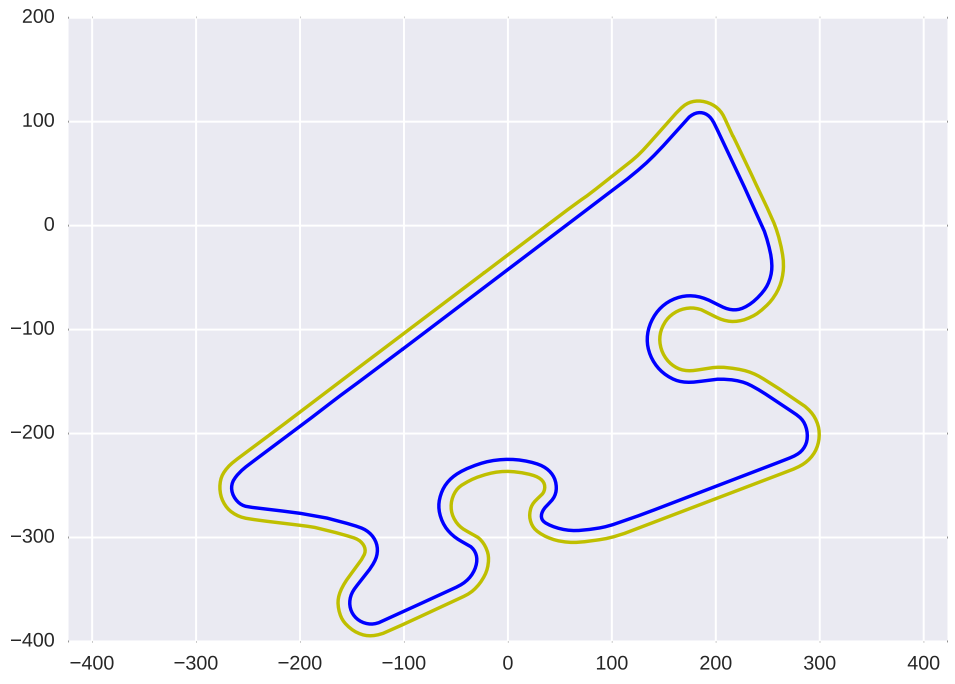


Figure D.3: Modena circuit layout. Axes are in meters.