Optimal Control and Reinforcement Learning for Formula One Lap Simulation

Christoph M. Höppke

Balliol College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2022

This thesis is dedicated to my parents
Sabine and Hans Dieter.

Acknowledgements

First and foremost, I would like to thank my team of advisors Prof. Yuji Nakatsukasa and Prof. Christoph Reisinger for many interesting mathematical conversations and their kind support. Secondly I would like to thank my team of industrial advisors and the InFoMM CDT directors Prof. Colin Please and Prof. Chris Breward for enabling my DPhil project. Let me also thank the people who shaped my development as a mathematician: A big thanks to Prof. Stefan Turek for introducing me to numerical analysis and Prof. Arieh Iserles for deepening my enthusiasm. I would also like to thank my collaborator Dr. Benno Simmons for providing invaluable postgraduate research opportunities. Let me also thank my guiding teachers, Alexander Kolodinski, Marin Neuhaus, and Peter Steinig for their support and for kindling my mathematical interest.
My deepest gratitude also goes to my friends and collogues in Oxford who have provided me with four truly unforgettable and who were always there to support me; let me extend special thanks to Nicolas Boullé, Maria del Rio-Chanona, Reneé Pereyra-Elías, Abinand Gopal, Katrin Harter, Fabian Laakman, Andrés Herrera Poyatos, Alex Puiu, Florian Rothenberg, Tommaso Senci, Timo Sprekeler, Arkady Wey, InFoMM CDT Cohorts I–VI, my parents Sabine and Hans Dieter, and my entire family for their endless support.

My DPhil studies at Oxford have been funded by the EPSRC Clarendon Fund and the Balliol College John Henry Jones scholarship, for which I am very grateful. My work was also supported by the EPSRC Centre For Doctoral Training in Industrially Focused Mathematical Modelling (EP/L015803/1).

Abstract

Lap simulation in a Formula One context is a subclass of optimal control problems and describes the computation of optimal trajectories around racing circuits. The results of lap simulation are primarily used for vehicle setup and strategic racing decisions. The optimal lap problem is solved using two classes of algorithms. The first algorithm uses direct collocation to compute optimal trajectories and the second algorithm uses specially constructed reinforcement learning environments and generalised function approximation to compute desirable system inputs. Historically direct collocation methods were considered impractical for minimum lap time simulations, due to their high computational costs. The exponential increase in computational performance has enabled the practical application of these algorithms.

These lap time simulations require a vehicle model, as well as a track discretisation. As an example for this, the classical bicycle model along with a curvilinear track model are introduced. To solve the resulting direct collocation problems, algorithms for non-linear optimisation problems are presented and performance critical aspects are discussed. The optimisation algorithm is accelerated by utilising highly parallel computer architectures, such as graphics processing units (GPUs). An analytical gradient approximation is presented to achieve approximations of projection systems which constitute one most performance critical components of the solution process. Mesh refinement algorithms are discussed and a novel mesh refinement heuristic based on optimal polynomial approximation in an L^1 sense is discussed. The L^1 approximation is improved by detecting singularities and using Clenshaw–Curtis quadrature on intermediary intervals.

In Chapter 4 of this work, the lap time optimisation problem is reformulated as a reinforcement learning environments. For this, the relevant

background literature on reinforcement learning is discussed and a translation of a training optimisation environment is constructed. Details of this environment are discussed in the form of reward signals, terminal conditions, and observation features. A series of learning models is discussed with increasing feature fidelity leading to an algorithm that can generalise well across representations of circuits from the 2022 Formula One calendar. This work expands on the current literature by providing novel, physically motivated, reinforcement learning environments for lap time optimisation tasks. The results of both approaches are combined by using strategy extraction to initialise the collocation optimisation algorithm and optimise the underlying mesh.

Contents

List of Figures

List of Tables

List of Algorithms

Symbols

$\mathcal{A}$ Action space 96

γ Discount factor 97

$\mathbb{E}$ Expected value operator 105

$\mathcal{X}$ Observation space 96

$\mathbf{P}_n$ Polynomial space of degree n 87

$\mathbb{P}$ Probability measure 98

t time 20

v_x Longitudinal Vehicle velocity in the vehicle reference frame 21

v_y Lateral Vehicle velocity in the vehicle reference frame 21

x Vehicle centre of gravity position, x component 21

y Vehicle centre of gravity position, y component 21

ψ Vehicle yaw orientation 21

ω Vehicle yaw rate 21

Chapter 1

Introduction

The industrial revolution and the development of digital computers have exponentially increased the number of systems we can interact with. Each time we interact with a system, that one can influence, we seek optimal inputs to achieve a goal state with the lowest cost, or in the shortest possible time. The rigorous formulation of controllable systems and the development of theory to describe, as well as algorithms to compute, these optimal inputs is known as *Optimal Control* Theory. In this Thesis we are interested optimal control for road vehicles, including the natural extension to autonomous racing. Figure 1.1 shows a broad categorisation of prior work in the domains of vehicle control and autonomous racing.

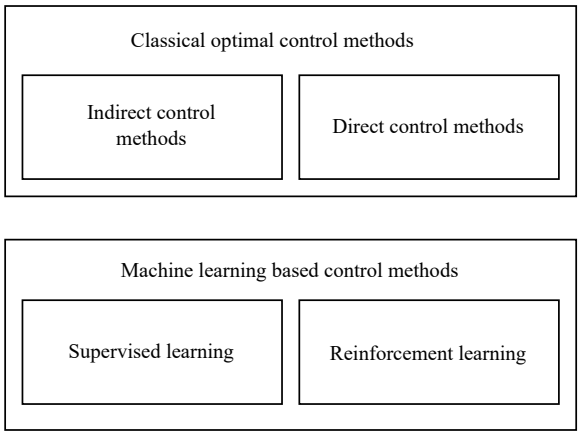


Figure 1.1: Optimal control methods categorisation

1.1 Optimal control for road vehicles

Optimal control for road vehicles dates back as far as the 1950s. In [82] simplified heuristic arguments were employed to estimate maximum cornering speeds based on vehicle parameters, such as body shape. From these cornering speeds an optimised trajectory, including regions for acceleration or breaking were estimated. By the early 1960s, it was common to use multiple shooting and gradient descent methods to solve trajectory optimisation problems in aerospace problems [17]. In motor-racing the objective is to complete a given set of manoeuvres, or racing laps, in the minimum possible time. This is commonly referred to as lap time optimisation. From a calculus of variations standpoint, it is natural to solve such a lap time optimisation problem by considering the necessary first order optimality conditions. We then solve these optimality conditions in the from of a two-point-boundary-value problem (TPBVP). This is the approach taken in [28]. In [8] an indirect method for solving optimal control problems using large multi-body systems is used. Later in the line of research on indirect optimal control methods for lap time simulation [92] uses an indirect method to compute minimum cornering times for single track vehicles performing a U-turn manoeuvre. Here a series of non-linear Pacejka-type tyre models [69, 68] are used for varying road surfaces. In addition to varying the tyre model, the authors

of [92] considered the effects of using different transmission layouts on the minimum cornering time.

The complexity of vehicle models, and tyre models has increased over the years. In 1969 the paper [32] introduced a velocity independent, physical based tyre model, which described the tyre forces using two constants for longitudinal and lateral stiffness. In 1987, Pacejka [68] introduced the semi-empirical magic formula model motivated by the models ability to fit experimental data. Later in 1995 [20] introduced the more complex LuGre model, able to capture the non-linear dependence of friction on sliding speed, known as the Stribeck effect [3]. This increase in model complexity made the adjoint equations, required for indirect optimal control methods more and more difficult to obtain. Consequently the use of direct optimal control methods for lap time optimisation increased.

We find literature using direct optimal control methods for time optimal vehicle control dating as far back as 1996 [42]. In this paper, the authors considered a lane change manoeuvre, subject to the Pacejka magic formula tyre model and used a direct multiple shooting-type optimal control method combined with a gradient descent optimiser. In the thesis [22] the author also used a direct multiple shooting method to compute optimal vehicle trajectories. The author uses the direct multiple shooting method to strike a balance between the instabilities and complexities associated with indirect methods and the increased non-linear optimisation problem size associated with direct optimal control methods. As noted in [15] the work presented in [22] was the first instance of full lap time optimisation performed using direct optimal control methods.

A key disadvantage of direct optimal control methods is the required computation time. Solving a full lap optimal trajectory for a simplified four wheeled vehicle model on the Barcelona track required up to 60 hours of computation time on a SunSpark workstation using the method presented in [22]. Later in [54] the authors were able to

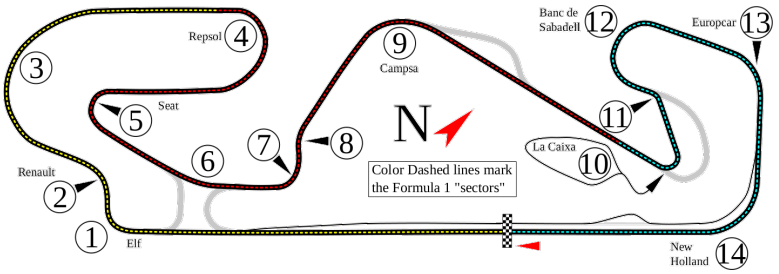


Figure 1.2: Barcelona Grand Prix race circuit 2004-2006 [77].

compute an optimal trajectory for the Jerez circuit using 10m meter control spacing in 8 hours of computation time on a 2.4GHz processor.

These long computation times required by direct optimal control methods inspired the development of faster alternatives. One approach to reduce the required computation time was to supply a racing line. This allowed the optimiser to only solve for the driver inputs which follow the given racing line in the shortest amount of time. This method has recently enabled real time control of model racing cars in at-limit handling situations [66, 80, 51, 67, 94].

In [15], the authors also introduced a quasi-steady-state method for computing optimal trajectories. The foundational assumption of quasi-steady-state methods is that transient behaviours of vehicles that accelerate, break, and corner smoothly can be well approximated by joining together a series of equilibrium, or steady-state, manoeuvres. The multiple shooting methods presented in [22] computed a 80.714s trajectory for the Barcelona race track. Compared to these results the quasi-steady-state method presented in [15] computed a 82.904s long trajectory, resulting in a significant difference of 2.19s.

In later works, direct collocation methods were successfully used for optimal lap time computations. The authors of [75] computed a trajectory which finishes the Barcelona race track 1.2 in 82.43 seconds. The algorithm required ≈ 1588 seconds to compute this trajectory at 1m node spacing[1].

[1]The authors of [75] present the formula $t_{\text{NLP}} \approx 0.005\,(4655/\Delta s)^{3/2}$ for the time required to

1.2 Machine learning-based methods

In recent years, we have seen the wide ranging success in the application of neural networks to challenging problems. This includes vehicle control problems such as autonomous driving and racing. The most popular machine learning approaches for vehicle control are imitation learning and reinforcement learning.

Imitation learning methods, instead of computing the optimal trajectory as part of the solution, aim to follow a given trajectory. This trajectory is often given by recorded data from an expert human or algorithmic demonstrator. One of the earliest examples for vehicle control using neural networks is the so called Autonomous Land Vehicle in a Neural Network (ALVINN) [78]. The network used in [78] is relatively small at one hidden layer of 29 nodes and 45 road curvature output nodes. The layers in this network are fully connected. The authors report that ALVINN was able to predict the correct road curvature in about 90% of the cases. In [14] the authors trained a convolutional neural network for road following. Recently we have also seen successes in the application of imitation learning to off road driving [72]. A natural upper bound on the quality of imitation learning approaches is given by the expert trajectories. In addition to this quality bound, the finite amount of training data can also be a limiting.

An alternative to imitation learning, which does not suffer from the described limitations is reinforcement learning. In reinforcement learning the expert trajectory is replaced by an environment and a reward function. These let an agent perform actions and measure their immediate desirability. Training data for reinforcement learning algorithms is generated during the training process. Recent advances in reinforcement learning for vehicle control include an autonomous drifting system [19], and the creation of a super human driving algorithm for the video game Gran Turismo

optimise non-linear programs using *IPOPT*. We evaluate this to $\approx$ 1588 seconds of compute time for $\Delta s = 1\text{m}$

Sport [36]. What sets reinforcement learning apart from imitation learning methods is the high performance achieved in extreme driving scenarios, such as [19] or beating records of expert human drivers [36].

We will expand on this by considering the interaction of reinforcement learning environments with the underlying optimal control problem. In particular, we will use the trajectory generated by reinforcement learning agents to increase the performance of a classical collocation based optimal control solver. In almost all reinforcement learning applications, only a single environment is considered for each problem. As we will demonstrate, lap time simulation offers the opportunity to learn from different state representations. We will introduce a simple base line environment, which only uses vehicle states present in the corresponding optimal control problem. The training results on the base line environment are later compared to performance achieved on state representations similar to the distance measurements from [36] or the image based observations from [78].

Our overarching goal will be to develop fast and reliable direct collocation based algorithms for lap time minimisation problems. For this, we introduce the classical bicycle model, following the description in [61], in Chapter 2. For a full model we incorporate the track model from [75]. Following the model construction we discuss direct collocation procedures in Chapter 3 and present initial numerical results for the lap time optimisation problem. We also present a mesh refinement heuristic, based on L^1 error estimates, in Section 3.7. Chapter 2 and 3 present the first part of this work. In addition to the classical direct collocation methods of Chapter 2 and 3, we also present reinforcement learning based methods in Chapter 4. We conclude with a discussion of the current results achieved by using reinforcement learning based initialisations for direct collocation problems in Section 4.5.

1.3 Introduction to optimal control methods

The objective of optimal control is to minimise a given cost functional, for a system, that accepts inputs, from a controller. Increasing automation, and digitisation of interactive systems has enabled new opportunities to increase process efficiency using optimal control methods. In recent history we have seen an acceleration of this trend by the application of machine leaning methods, and reinforcement learning methods in particular. In this Section, we introduce the foundational theory necessary to solve optimal control problems.

A non-trivial part of any control problem is to mathematically model the process at hand. Many processes can be expressed in the form of controlled ordinary differential equations. In controlled differential equations, we consider two types of variables, the states

$$\mathbf{x}(t) = (x_1(t), x_2(t), \dots, x_n(t)), \tag{1.3.1}$$

and the controls

$$\mathbf{u}(t) = (u_1(t), u_2(t), \dots, u_m(t)). \tag{1.3.2}$$

The state variables, as the name suggests, describe our model of the current system state, and the control variables act as inputs the dynamics that can influence the development of the states. We write these dynamics in the form

$$\begin{aligned} x_1'(t) &= f_1(\mathbf{x}(t), \mathbf{u}(t), t), \\ x_2'(t) &= f_2(\mathbf{x}(t), \mathbf{u}(t), t), \\ &\vdots \\ x_n'(t) &= f_n(\mathbf{x}(t), \mathbf{u}(t), t) \end{aligned} \tag{1.3.3}$$

for which we will use the vectorised notation

$$\mathbf{x}'(t) = \mathbf{f}\left(\mathbf{x}(t), \mathbf{u}(t)\right). \tag{1.3.4}$$

In most applications, there are natural bounds on our states and controls. This can include physical limits, which constrain the range of feasible inputs or constraints on the states by which we define a set of desirable behaviours for our system. In the context of optimising vehicle trajectories we find that it is natural to have bounded controls in the form of an acceleration or breaking force, as well as a steering input. We know that bounds on the on acceleration and breaking force depend on specific engine, differential and transmission properties, as well as tyre-road friction coefficient. The steering setup determines bounds on the steering angle input. We are thus particularly interested in problem formulations with bounded controls

$$u_i(t) \in [u_{i,\text{lower}}, u_{i,\text{upper}}] \qquad \forall i = 1, \ldots, m. \tag{1.3.5}$$

We are now interested in functions that we may integrate against and that satisfy the bound condition (1.3.5) at every point. This leads us to the definition of an *admissible control history.*

Definition 1.3.1 (Admissible control)**.** *Let* $\mathbf{u} : [t_0, t_f] \to \mathbb{R}^m$ *be measurable with respect to the Lebesgue sigma algebra [91, p.17, p.51-p.57], such that each component of* $\mathbf{u}$ *satisfies the bound condition* (1.3.5)*. We then call* $\mathbf{u}$ *an admissible control history. We denote the set of admissible control histories by*

$$\mathcal{U} = \{\mathbf{u} | \mathbf{u} \textit{ is measurable and } u_i(t) \in [u_{i,lower}, u_{i,upper}] \quad \forall i, t\}. \tag{1.3.6}$$

Similarly to bounded controls, we constrain the set of desirable states, and call a

state $\mathbf{x} \in \mathbb{R}^n$ with $c(\mathbf{x}) \geq 0$ feasible. The set of all feasible states is thus

$$\mathcal{X} = \{\mathbf{X} \in \mathbb{R}^n | c(\mathbf{x}) \geq 0\}\,.$$

The overall task is to find an admissible control $\mathbf{u}^*$, which minimises a certain cost functional

$$L(\mathbf{u}) = \int_{t_0}^{t_f} \underbrace{r\left(\mathbf{x}(t), \mathbf{u}(t), t\right)}_{\text{Lagrange term}}\, dt + \underbrace{g(\mathbf{x}(t_0), \mathbf{x}(t_f))}_{\text{Mayer term}}\,. \tag{1.3.7}$$

From a mathematical perspective in optimal control, the fundamental questions we seek to answer are:

- Does an optimal control exist?
- How can we characterise an optimal control?
- How can we compute an optimal control?

In the following, we will discuss the general concept of controllability and the emergence of so called *bang-bang* controls for linear time-optimal controls.

1.3.1 Controllability

When we discuss the controllability of an ODE we try to answer whether we can find an admissible control $\mathbf{u}(\cdot)$ which steers the state trajectory $\mathbf{x}(\cdot)$ from an initial state $\mathbf{x}_0$ towards a desired final state $\mathbf{x}_f$. The state trajectory follows the controlled dynamics function $\mathbf{f}$, such that we seek,

$$\mathbf{x}'(t) = \mathbf{f}\left(\mathbf{x}(t), \mathbf{u}(t), t\right), \tag{1.3.8}$$

$$\mathbf{x}(t_0) = \mathbf{x}_0, \quad \mathbf{x}(t_f) = \mathbf{x}_f \tag{1.3.9}$$

$$\mathbf{x}(t) \in \mathcal{X}, \mathbf{u} \in \mathcal{U} \quad \forall t \in (t_0, t_f). \tag{1.3.10}$$

We define the *reachable set* at time t $\mathcal{C}(t)$ to be the set of initial points from which the origin is reachable at time t, i.e.

$$\mathcal{C}(t) := \{\mathbf{x}_0 \in \mathbf{X} |\ \exists\ \mathbf{u} \in \mathcal{U} \text{ such that } \mathbf{x}(t) = 0\}\,. \tag{1.3.11}$$

Consequently, we define the general *reachable set* $\mathcal{C}$ as the union of all reachable sets at times $t \geq 0$,

$$C = \bigcup_{t \geq 0} C(t). \tag{1.3.12}$$

To introduce time-optimal control problems we consider a simplified linear example dynamic

$$\mathbf{x}'(t) = M\mathbf{x}(t) + N\mathbf{u}(t), \qquad \mathbf{x}(0) = \mathbf{x}_{\text{init}}, N \in \mathbb{R}^{n\times m}, M \in \mathbb{R}^{m\times m}. \tag{1.3.13}$$

Given a starting point $\mathbf{x}_{\text{init}}$ we want to find the control function $\mathbf{u}(\cdot)$ which steers the state trajectory towards the origin in the minimal time. For this, we define the time of the first arrival at the origin

$$T(\mathbf{u}) := \inf\{t \geq 0 | \mathbf{x}(t) = 0,\ \mathbf{x}(0) = \mathbf{x}_{\text{init}}\}\,.$$

For the minimal time problem, our objective function is

$$L(\mathbf{u}) = \int_0^{T(\mathbf{u})} 1\, dt. \tag{1.3.14}$$

Our goal is now to establish the existence of an optimal control function for the time minimisation problem. For this, we follow [49, ch. 2] and discuss the structure of the reachable set for linear control problems. With this structure we will then establish the existence of time-optimal controls for the linear problem (1.3.13).

Theorem 1.3.2 (Structure of the controllable set C)**.** *Let C be the reachable set of the controlled system* (1.3.13) *and let $A \subset \mathbb{R}^m$ be a convex set which contains the origin in its interior $0 \in A^\circ$. Further let the set of admissible controls be*

$$\mathcal{U} = \{\mathbf{u} : [0,t] \to A | \mathbf{u} \text{ is measurable, } t > 0\},$$

then the set of reachable states, C, is convex. We define the controllability matrix

$$G = \left[N, MN, M^2N, \ldots, M^{n-1}N\right] \in \mathbb{R}^{n\times(mn)}.$$

It holds that

$$0 \in C^\circ \Leftrightarrow \operatorname{rank}(G) = n. \tag{1.3.15}$$

Proof. We first establish the convexity of C. Let $\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \in C$ and let $\mathbf{u}^{(1)}, \mathbf{u}^{(2)} \in \mathcal{U}$ be the corresponding controls. We use the fundamental solution matrix function

$$\mathbf{X}(t) = e^{tM} = \sum_{i=0}^{\infty} \frac{t^i}{i!} M^i.$$

Using the method of varying parameters we find the explicit solution of Equation (1.3.13)

$$\mathbf{x}^{(i)}(t) = X(t)\mathbf{x}^{(i)} + X(t)\int_0^t X(-s)N\mathbf{u}^{(i)}(s)\,ds. \tag{1.3.16}$$

As $\mathbf{x}^i \in C(t_i)$ for some $t_i > 0$ we find that

$$\begin{aligned} 0 &= X(t_i)\mathbf{x}^{(i)} + X(t_i)\int_0^{t_i} X(-s)N\mathbf{u}^{(i)}(s)\,ds \\ \Leftrightarrow \mathbf{x}^{(i)} &= -\int_0^{t_i} X(-s)N\mathbf{u}^{(i)}(s)\,ds. \end{aligned}$$

Assume that $t_2 \geq t_1$. Should $t_1 > t_2$, then swap $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$. To show convexity we

choose $\lambda \in [0,1]$ and show that

$$\tilde{\mathbf{x}} = \lambda \mathbf{x}^{(1)} + (1-\lambda)\mathbf{x}^{(2)} \in C$$

by considering the convex combination of controls

$$\tilde{\mathbf{u}}(t) = \begin{cases} \lambda \mathbf{u}^{(1)}(t) + (1-\lambda)\mathbf{u}^{(2)}(t) & \text{for } t \leq \min t_1, t+2, \\ (1-\lambda)\mathbf{u}^{(2)}(t) & \text{for } t \in (t_1, t_2], \\ 0 & \text{for } t > t_2. \end{cases}$$

As both $\mathbf{u}^{(1)}, \mathbf{u}^{(2)}$ are measurable and take values in A, we follow that $\tilde{\mathbf{u}} \in \mathcal{U}$ is admissible. By Equation (1.3.16), we find that

$$\mathbf{x}^{(1)} = -\int_0^{t_1} X(-s) N u^{(1)}\, ds, \tag{1.3.17}$$

$$\mathbf{x}^{(2)} = -\int_0^{t_2} X(-s) N u^{(2)}\, ds. \tag{1.3.18}$$

so that

$$\begin{aligned} \lambda \mathbf{x}^{(1)} + (1-\lambda)\mathbf{x}^{(2)} &= -\lambda \int_0^{t_1} X(-s) N \mathbf{u}^{(1)}(s)\, ds \\ &\quad - (1-\lambda) \int_0^{t_2} X(-s) N \mathbf{u}^{(2)}(s)\, ds \\ &= -\int_0^{t_2} X(-s) N \tilde{\mathbf{u}}(s)\, ds \end{aligned}$$

and we conclude that $\tilde{\mathbf{x}} \in C$. As $\lambda, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}$ were arbitrary it follows that C is convex. We now use the convexity to establish that $\operatorname{rank}(G) = n \Leftrightarrow 0 \in C^{\circ}(0)$.

Clearly $\operatorname{rank}(G) \leq n$, assume $\operatorname{rank}(G) < n$, then there exists a $b \in \mathbb{R}^n \setminus \{0\}$ such

that $b^T G = 0$ and thus

$$\begin{aligned} b^T G &= b^T \left[N, MN, M^2 N, \ldots, M^{n-1} N \right] \\ &= \left[b^T N, b^T MN, \ldots, b^T M^{n-1} N \right] \\ &= 0. \end{aligned}$$

In fact, we claim that $b^T M^k N = 0 \quad \forall k \in \mathbb{N}_{\geq 0}$. To show $b^T M^n N = 0$ we appeal to the Cayley–Hamilton theorem [35] and find that the characteristic polynomial

$$p(\lambda) = \det(\lambda I - M) = \lambda^n + \sum_{k=0}^{n-1} \beta_k \lambda^k,$$

satisfies

$$p(M) = M^n + \sum_{k=0}^{n-1} \beta_k M^k = 0.$$

From this, we conclude

$$b^T M^n = -\sum_{k=0}^{n-1} \beta_k \underbrace{b^T M^k N}_{=0} = 0.$$

We show $b^T M^k N = 0$ for $k > n$ by induction. Assume that $b^T M^l N = 0$ for some $k \geq n$ and all $l \leq k$. It follows that

$$b^T M^{k+1} = -\sum_{i=0}^{n-1} \beta_i \underbrace{b^T M^{k-n+i+1} N}_{=0} = 0.$$

Now let $\mathbf{x}_{\text{init}} \in C$ be an arbitrary reachable point and let $\mathbf{u} : [0, t] \to A$ be the

corresponding control function. We find that

$$\begin{aligned} b^T \mathbf{x}_{\text{init}} &= -\int_0^t b^T X(-s) N \mathbf{u}(s)\, ds \\ &= -\int_0^t \sum_{k=0}^{\infty} \frac{(-s)^k}{k!} \underbrace{b^T M^k N}_{=0} \mathbf{u}(s)\, ds. \end{aligned}$$

As $\mathbf{x}_{\text{init}} \in C$ was arbitrary we find span(b) is a supporting hyperplane to C and thus $C \perp \text{span}(b)$ which implies $C^\circ = \emptyset$.

For the contrary argument, we assume that $0 \notin C^\circ$. As C is convex, we can find a supporting hyperplane H to C which goes through 0. Thus $\exists\ b \in \mathbb{R} \setminus \{0\}$ with $b^T x \geq 0 \quad \forall x \in C$. Let $\mathbf{x} \in C$ be arbitrary and $\mathbf{u} \in \mathcal{U}$ be the corresponding control function. Let $\mathbf{u} : [0,t] \to \mathbb{R}$ be arbitrary and let x be an arbitrary point of the

$$0 \leq b^T x = -\int_0^t b^T X(-s) N \mathbf{u}(s)\, ds.$$

Assume that the set $Q = \left\{ s \in [0,t] |\ b^T X(-s) N \mathbf{u}(s) \right\}$ has non-zero measure. As $0 \in A^\circ$ we can choose $\epsilon > 0$ sufficiently small, such that

$$\tilde{\mathbf{u}} = \begin{cases} -\epsilon \mathbf{u}(s) & , s \in Q \\ 0 & , \text{otherwise} \end{cases}, \qquad \text{and} \quad \tilde{\mathbf{u}} \in \mathcal{U}.$$

With the control $\tilde{\mathbf{u}}$ we find that the point

$$\tilde{\mathbf{x}} = \int_0^t X(-s) N \tilde{\mathbf{u}}(s)\, ds \in C, \tag{1.3.19}$$

is reachable. However $b^T \tilde{\mathbf{x}} > 0$, which contradicts that $|Q| > 0$ and hence

$$b^T X(-s) N \mathbf{u}(s) = 0 \ \text{ for a.e. } s \in [0,t].$$

With $\epsilon_2 > 0$ sufficiently small we find that

$$\mathbf{x}_0 = \int_0^t X(-s)N\epsilon_2 e_i \, ds$$

is reachable for all $i = 1, \ldots, m$ and thus by the previous argument $b^T X(-s)Ne_i = 0 \ \ \forall i = 1, \ldots, m$. Hence

$$0 = b^T X(-s)N = b^T e^{-sM} N \ \ \forall s \geq 0.$$

By differentiating, we find

$$\begin{aligned} 0 &= \frac{d^k}{ds^k} \left(b^T e^{-sM} N \right) \Big|_{s=0} = (-1)^k b^T M^k N, \\ &\Rightarrow b^T G = 0. \end{aligned}$$

As $b \neq 0$ it follows that G does not have full rank. □

We now use Theorem 1.3.2 to show the existence of a time-optimal control function.

Theorem 1.3.3 (Existence of time-optimal control)**.** *Let $\mathbf{x}_{init} \in C$ be an initial point for the controlled system* (1.3.13)*. Further let $A \subset \mathbb{R}^m$ be a compact and convex set with $0 \in A^\circ$. If*

$$\mathcal{U} = \{u : [0, t] \to A |\ t > 0,\ u \ \ \textit{measurable}\},$$

then there exists a time-optimal control function $\mathbf{u}^*$ in the sense that

$$T(\mathbf{u}^*) = \inf \{T(\mathbf{u}) | u \in \mathcal{U}\}.$$

Proof. By Theorem 1.3.2 we establish $0 \in C^\circ$. The optimal achievable time is

$$\tau^* = \inf \{t |\ \mathbf{x}_{\text{init}} \in C(t)\}.$$

As $\mathbf{x}_{\text{init}} \in C$ we can find a $t > 0$ and a $\mathbf{u} : [0, t] \to A$, such that

$$\mathbf{x}_{\text{init}} = -\int_0^t X(-s) N \mathbf{u}(s)\, ds.$$

We first establish that $\mathbf{x}_{\text{init}} \in C(\tilde{t})$ for all $\tilde{t} \geq t$. Given $\tilde{t} > t$, we extend $\mathbf{u}$ by letting

$$\tilde{\mathbf{u}}(s) = \begin{cases} \mathbf{u}(s) & \text{for } s \leq t \\ 0 & \text{for } s > t \end{cases},$$

and thus

$$\mathbf{x}_{\text{init}} = -\int_0^{\tilde{t}} X(-s) N \tilde{\mathbf{u}}(s)\, ds \in C(\tilde{t}).$$

This implies $C(t) \subset C(\tilde{t}) \quad \forall \tilde{t} \geq t$. We now choose a sequence $t_1 > t_2 > \dots$ with $t_n \searrow \tau^*$. For each t_n we find a control $\mathbf{u}_n : [0, t_n] \to A$ and, after extending by 0 to $[0, t_1]$, we use the weak-$*$ compactness of L^∞, established by Alaoglu's Theorem [60], to select a subsequence $t_{n_k} \searrow \tau^*$ such that

$$\mathbf{u}_{n_k} \overset{*}{\rightharpoonup} \mathbf{u}^* \in \mathcal{U}.$$

We find that $\mathbf{u}^*$ is optimal and that further

$$\int_{\tau^*}^{t_1} \mathbf{u}^*(s) \cdot v(s)\, ds \leftarrow \int_{\tau^*}^{t_1} \mathbf{u}_{k_n}(s) \cdot v(s)\, ds = 0,$$

for all $v \in L^1(0, t_1)$ and $n \in \mathbb{N}_{\geq} 1$ sufficiently large. Thus $\mathbf{u}^*(s) = 0 \quad \forall s > \tau^*$.

Therefore

$$\mathbf{x}_{\text{init}} = -\int_0^{t_1} X(-s)N\mathbf{u}^*(s)\,ds = -\int_0^{\tau^*} X(-s)N\mathbf{u}^*(s)\,ds,$$

so that $\mathbf{x}_{\text{init}} \in C(\tau^*)$ is attained using the control $\mathbf{u}^*$. □

Chapter 2

Vehicle and Track model

Our overarching goal is to find optimal driver inputs to navigate a vehicle around a racetrack in the shortest possible time. Two fundamental components in this procedure, regardless of using direct or indirect optimisation methods, are the vehicle and track model. In this section, we follow [61] and introduce a classical vehicle model, known as the Bicycle model. In addition to this classical vehicle model, we introduce a track model from [75]. Historically, the development of vehicle modelled occurred relatively late, compared to aircraft models. The first full force model for vehicles was developed by Maurice Olley, who was then an engineer at Rolls Royce.

2.1 Bicycle model

As the name bicycle model suggests, a four wheeled vehicle is approximated by a two wheeled one. For this approximation, forces occurring at the front and rear axle wheels are lumped together. Lumping the axes together implies that we are no longer able to model lateral roll transfers and roll motions. We ignore both of these effects and make a series of further simplifying assumptions.
The simplifying assumptions, as outlined by [61], are:

(1) No lateral load transfer. By lumping the front and rear axes of four wheeled two track vehicles together, we can no longer model lateral load transfers. Such a load transfer occurs naturally during cornering or driving on an angled surface. For example, in cornering, more of the vehicles weight will rest on the outside set of wheels as a result of tyre and centripetal forces, thus generating a load transfer along the lateral axis.

(2) No longitudinal load transfer. Similar to lateral load transfer, the bicycle model also neglects longitudinal load transfer effects, such as during acceleration or breaking.

(3) No pitch or roll motions. We normally describe an objects location and rotation in space with six degrees of freedom, consisting of location and rotation for each of respective to the three special dimensions. Roll and pitch describe a vehicle's rotation relative to the longitudinal and lateral vehicle axis. Similar to how lateral and longitudinal load transfer effects are neglected, we further neglect rotation around these axles.

(4) Lateral tyre forces depend linearly on slip angles. Tyres generate lateral, i.e. cornering forces due to the deformation of their contact patch with the road surface that occurs during cornering. We introduce the concept of slip angles later in this section and expand on this assumption to account for larger slip angles. The slip angle is the angle between the direction a tyre is pointing in and travelling in. A visualisation can be seen in Figure 2.1.

(5) No aerodynamic effects. Aerodynamic effect, while highly relevant in motor racing. In fact, a Formula One car can generate up to five times its own gravitational force in aerodynamic down force[53].

(6) Position control. Position control is known as that mode of vehicle operation,

where the steering input is known as a displacement input rather than an applied force. In non-electronic steering systems the driver will sense both displacement and force input in varying proportions. The invention of electronic and power assisted steering systems make this assumption more realistic. These two modes are called position control (or fixed) and force control (or free)[61].

(7) No chassis or suspension compliance effects. Lastly, we neglect the influence of chassis deformation, or compliance effects.

We assume further that the vehicle drives on an even and smooth road surface. Figure 2.1 shows a bicycle, rotated at an angle ψ relative to the global reference frame, additionally the front wheel is rotated at a steering angle δ relative to the vehicle chassis. During cornering, the contact patch between wheel and road is deformed. This deformation exerts a lateral force proportional to the tyre stiffness [69, 21]. The magnitude of this deformation depends on the angular difference between the direction in which the tyre is pointing and the direction in which is travelling. This angular difference is called slip angle. The slip angles α_F and α_R for the front and real axis respectively can also be seen in Figure 2.1. The state of the vehicle can thus

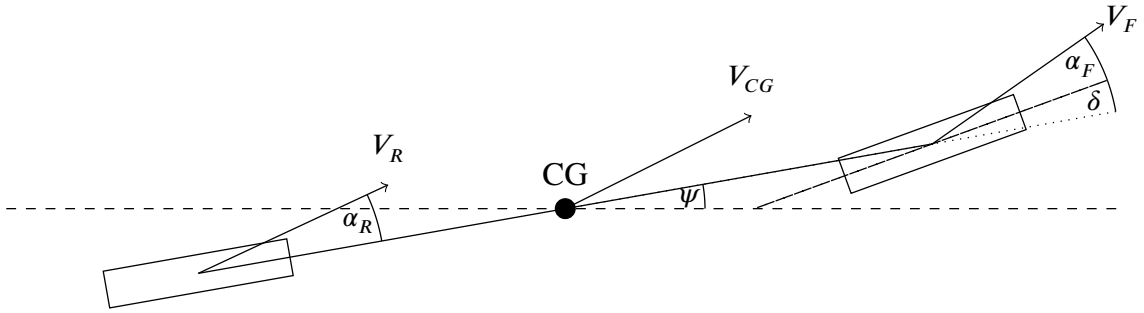


Figure 2.1: Planar view of a vehicle as described in the bicycle model

be described using the three state variables

- $x(t)$ the horizontal position of the vehicles centre of gravity $C(t)$.
- $y(t)$ the vertical position of the vehicles centre of gravity.

- $\psi(t)$ the angular position of the vehicle as it rotates around the z axis.
- $v_x(t)$ the longitudinal velocity in a vehicle aligned coordinate system.
- $v_y(t)$ the lateral velocity in a vehicle aligned coordinate system.
- $\omega(t)$ the angular velocity or yaw rate.

In order to create a first order model, we also introduce the first derivatives of the above state variables into our model, resulting in the state vector

$$\mathbf{y}(t) = \Big(x(t), y(t), \psi, v_x(t), v_y(t), \omega(t)\Big)^T. \tag{2.1.1}$$

2.2 Bicycle model equations of motion

In the previous section we have introduced the bicycle vehicle model and discussed the physical assumptions and simplifications which this model includes. In this section, we expand on this by deriving the equations of motions for a vehicle within the bicycle model setting. During this process we put particular consideration onto the rotating frame of reference. The rotation of the vehicle implies that velocities at the front wheel, rear wheel, and the centre of gravity will differ from each other. We denote the distance from the front wheel to centre point to the vehicle centre of gravity by a, and the distance from the rear wheel to centre of gravity by b. With this, we can write the front wheel location $F(t)$ and rear wheel location $R(t)$ as

$$F(t) = C(t) + a \begin{pmatrix} \cos(\psi(t)) \\ \sin(\psi(t)) \end{pmatrix} \tag{2.2.1}$$

$$R(t) = C(t) - b \begin{pmatrix} \cos(\psi(t)) \\ \sin(\psi(t)) \end{pmatrix}, \tag{2.2.2}$$

where $C(t)$ is the centre of gravity location. We let $C'(t) = \left(v_x^{\mathrm{g}}, v_y^{\mathrm{g}}\right)$ be the velocities at the centre of mass in a global reference frame. Taking first time derivatives in equations (2.2.1) and (2.2.2) gives

$$F'(t) = C'(t) + a\psi'(t) \begin{pmatrix} -\sin(\psi(t)) \\ \cos(\psi(t)) \end{pmatrix} \tag{2.2.3}$$

$$R'(t) = C'(t) - b\psi'(t) \begin{pmatrix} -\sin(\psi(t)) \\ \cos(\psi(t)) \end{pmatrix}, \tag{2.2.4}$$

Tyre forces are generated primarily due to a contact patch deformation which occurs as the wheel orientation differs from a wheels travel direction. The authors of [61, p.18-19] provide an insightful interpretation of slip angles. The presence of slip angles does not mean that the wheel is sliding, apart from the trailing edge of a wheels contact path, where sliding occurs. At high slip angles, the tyre will be deformed further, resulting in a larger sliding portion of the road-tyre contact patch. The ability for a tyre to travel outside its plane, without significant slippage, results from the lateral contact patch deformation and the motion of the contact patch due to the wheels rotation. In [61, p.18-19], we find the interpretation of walking at an angle to ones heading direction by successively offsetting one foot after the other in a lateral direction. Figure 2.2 shows a visualisation of this process. We may consider a tyres

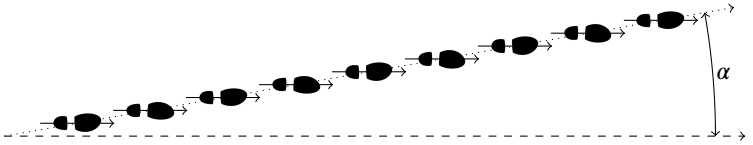


Figure 2.2: Walking analogy to tyre slip angles from [61, p.19].

behaviour to be the continuous case of this lateral walking as the number of feet approaches infinity.

We now consider the slip angles at the rear wheel, for which we let H_{rw} be the

heading direction of the rear wheel and P_{rw} be the direction it is pointing in. From Equation (2.2.4), we obtain

$$H_{\text{rw}}(t) = \begin{pmatrix} v_x^{\text{g}} + b\psi'(t)\sin(\psi(t)) \\ v_y^{\text{g}} - b\psi'(t)\cos(\psi(t)) \end{pmatrix}. \tag{2.2.5}$$

We assume that the rear wheel is aligned with the vehicle orientation, so that

$$P_{\text{rw}}(t) = \begin{pmatrix} \cos(\psi(t)) \\ \sin(\psi(t)) \end{pmatrix}. \tag{2.2.6}$$

With Equations (2.2.5) and (2.2.6) we can compute the rear wheel slip angle as

$$\cos(\alpha_{\text{rw}}) = \frac{\langle H_{\text{rw}}, P_{\text{rw}} \rangle}{\|H_{\text{rw}}\| \|P_{\text{rw}}\|}. \tag{2.2.7}$$

We observe that both H_{rw} and P_{rw} are naturally aligned with the vehicle reference frame, so that we may simplify (2.2.7) by considering both H_{rw} and P_{rw} in a vehicle aligned reference frame. For this we use the rotation matrix

$$R(\psi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{pmatrix},$$

so that we find

$$\begin{aligned} \cos(\alpha_{\text{rw}}) &= \frac{\langle R(-\psi)H_{\text{rw}}, R(-\psi)P_{\text{rw}} \rangle}{\|R(-\psi)H_{\text{rw}}\|} \\ &= \frac{v_x}{\sqrt{v_x^2 + (v_y - b\psi')^2}}. \end{aligned} \tag{2.2.8}$$

From Equation (2.2.8) we see that we may compute the rear wheel slip angle by using

$$\alpha_{\text{rw}} = \arctan\left(\frac{v_y - b\psi'}{v_x}\right). \tag{2.2.9}$$

For the rear wheel slip angle we proceed analogous to the rear wheel slip angle. The

front axis of our vehicle is steered, we thus denote the steering angle by δ. We can see the front wheel slip angle in Figure 2.3.

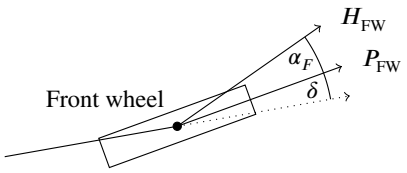


Figure 2.3: Front wheel slip angle

We may use the fact that the front wheel slip angle is the angular difference between a slip angle in the case of no steering $\alpha_{\text{fr},0}$ and the steering angle δ. In the case of no steering, the front and rear wheels are both aligned with the vehicle frame, and by Equation (2.2.3), the direction in which the front wheel travels is

$$H_{\text{fw}}(t) = F'(t) = \begin{pmatrix} v_x^{\text{g}}(t) - a\psi'(t)\sin(\psi(t)) \\ v_y^{\text{g}}(t) + a\psi'(t)\cos(\psi(t)) \end{pmatrix}. \tag{2.2.10}$$

We conclude that

$$\cos(\alpha_{\text{rw},0}) = \frac{\langle R(\psi)H_{\text{fw}}, R(\psi)P_{\text{rw}}\rangle}{\|R(\psi)H_{\text{fw}}\|} = \frac{v_x}{\sqrt{v_x^2 + (v_y + a\psi')^2}}. \tag{2.2.11}$$

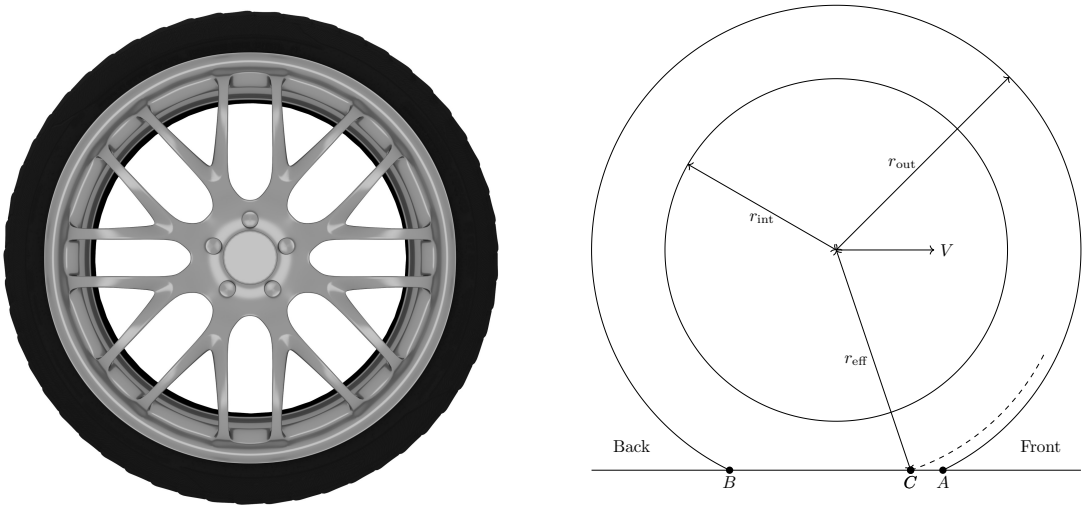
Similarly to the rear wheel case, we simplify Equation (2.2.11) to obtain

$$\begin{aligned} \alpha_{fw,0} &= \arctan\left(\frac{v_y + a\psi'}{v_x}\right), \\ \alpha_{fw} &= \alpha_{fw,0} - \delta = \arctan\left(\frac{v_y + a\psi'}{v_x}\right) - \delta. \end{aligned} \tag{2.2.12}$$

Understanding the interaction between road and tyre is crucial to understanding and modelling the motion of any road vehicle. In the following section, we expand on this by introducing tyre models in both low and high slip-angle paradigms.

2.2.1 Tyre modelling

Forces and moments acting on tyres greatly impact the vehicle dynamics. In this section, we discuss mathematical models to describe these tyre forces. As the vehicle tyres as not perfectly rigid, they deform due to vertical load forces. The deformation results in a footprint of non-zero area, which we call the *contact patch*. This deformation results in a reduction of the tyres radius. In the following we will denote the resulting effective tyre radius by r_{eff}. Figure 2.4b shows, the difference in geometric and effective tyre radius, as well as the resulting contact patch[1]. Following [79], we



(a) Wheel render with tyre and rim (b) Wheel geometry and effective tyre radius

Figure 2.4: Wheel and tyre geometries

assume that tyre forces and moments act at the centre of the tyre. We first consider the generation of longitudinal tyre forces at small slip angles. For this, we consider the slip ratio of a tyre, which describes the difference between the longitudinal tyre velocity and the velocity of a particle located at the effective radius of the tyre. These two velocities differ during acceleration or breaking, which causes the contact patch to stretch or compress. We denote the wheels rotation speed by ω_w and the longitudinal

[1]Render created using the Blender software [11].

velocity component of the wheel by V_x, such that we can compute the slip ratio σ_x as

$$\sigma_x = \begin{cases} \frac{r_{\text{eff}}\omega_w - V_x}{V_x}, & \text{in case of breaking,} \\ \frac{r_{\text{eff}}\omega_w - V_x}{r_{\text{eff}}\omega_w}, & \text{in case of acceleration} \end{cases} . \tag{2.2.13}$$

We know, from experimental results [79, 12] that, in the regime of low slip ratios, longitudinal forces are proportional to the slip ratio σ_x. Figure 2.5, which we reproduced from [79, Figure 13-5], shows longitudinal tyre forces during acceleration ($\sigma_x > 0$) and breaking ($\sigma_x < 0$). As we can see from Figure 2.5 the linear tyre force

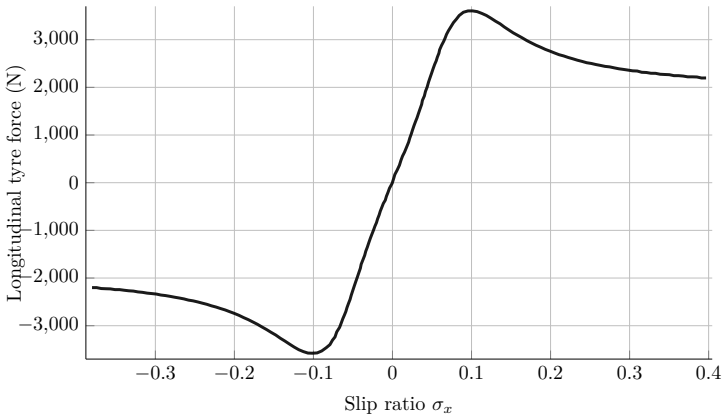


Figure 2.5: Longitudinal tyre forces

model

$$F_{x,\text{front}} = C_{\sigma,\text{front}}\sigma_{x,\text{front}}, \tag{2.2.14}$$

$$F_{x,\text{rear}} = C_{\sigma,\text{rear}}\sigma_{x,\text{rear}} \tag{2.2.15}$$

$$\tag{2.2.16}$$

serves as a good approximation for low slip ratios $\sigma_x < 0.1$ We call $C_{\sigma,\text{front/rear}}$ the longitudinal tyre-stiffness parameter. A rough explanation of why the longitudinal force is proportional to the slip ratio originates in so called "brush" tyre models [71]. Brush models describe the tyre tread as a series of independent springs. These springs

undergo longitudinal deformation and resist with a constant longitudinal stiffness. Let the longitudinal velocity of a tyre be V_x and its rotational velocity ω_w. The resulting net velocity of a tyre tread particle is therefore $r_{\text{eff}}\omega_w - V_x$. We consider the exemplary case of a driven wheel during acceleration, where $r_{\text{eff}}\omega_w - V_x > 0$ is small. When a new tread element enters the contact patch its tip, which is in contact with the road surface, has zero velocity. Therefore, no sliding motion occurs at the front of the contact patch. We therefore call this early part of the contact path, the static region. The upper part of the tread element moves with a velocity $r_{\text{eff}}\omega_w - V_x$ in the direction opposite to the longitudinal motion V_x. Hence the tread element bends forward. The maximal bending deflection is proportional to the relative velocity $r_{\text{eff}}\omega_w - V_x$ and the duration over which a tread element remains bend is inversely proportional to the rotational speed $r_{\text{eff}}\omega_w$, resulting in a force proportional to the slip ratio

$$\frac{r_{\text{eff}}\omega_w - V_x}{r_{\text{eff}}\omega_w}.$$

To obtain lateral tyre deflection, forces we present a simple analytical model which uses the same "brush" tyre model which we used for lateral force generation. Fig-

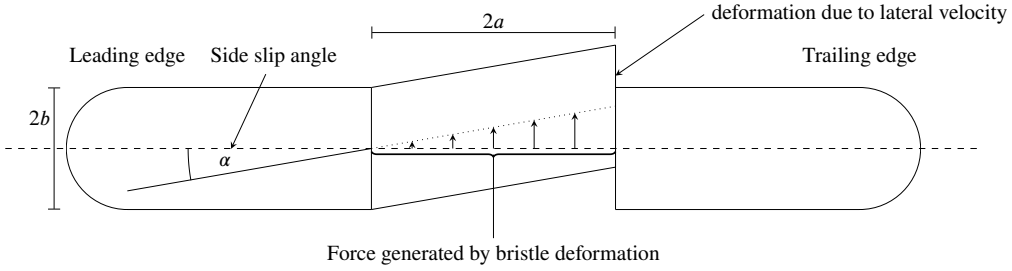


Figure 2.6: Lateral force generation by tyre deflection

ure 2.6 shows a top view of a tyre with a contact patch length of $2a$ and a tyre width of $2b$. Let c be the lateral stiffness per unit length of the tyre and let $\gamma(x) = x\tan(\alpha)$ be the lateral deformation of the contact patch at position $x \in [0, 2a]$. The total

generated force is hence

$$F_y = \int_0^{2a} c\gamma(x)\,dx = ca^2\tan(\alpha) = ca^2\alpha + \mathcal{O}(\alpha^2). \tag{2.2.17}$$

We now have a lateral force generation model for small slip angles $\alpha \ll 1$. To model tyre forces in the case of larger slip angles, we consider that the total lateral force can not exceed the friction force at the tyre. Let F_z be the total downward force at the tyre, then μF_z is the maximal friction force. Hence we know that

$$2ac\gamma_{\max} \leq \mu F_z.$$

We can see the relationship between tyre deformation and lateral position in Figure 2.7. so that we can compute

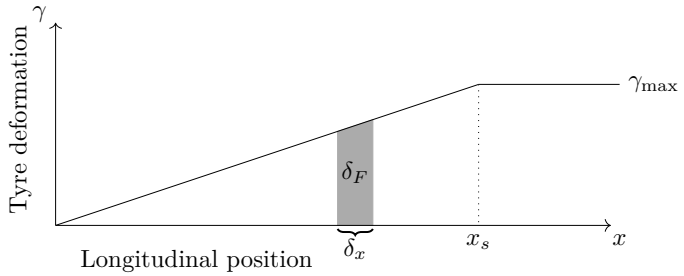


Figure 2.7: Tyre tread deformation at the contact patch

$$\begin{aligned}
F_y &= \int_0^{2a} c\gamma(x)\,dx \\
&= \int_0^{2a} c\min(\tan(\alpha)x, \gamma_{\max})\,dx \\
&= \int_0^{2a} c\min\left(\tan(\alpha)x, \frac{\mu F_z}{2ac}\right)\,dx \\
&\approx \int_0^{x_s} c\tan(\alpha)x\,dx + (2a - x_s)\frac{\mu F_z}{2a\tan(\alpha)},
\end{aligned}$$

where $x_s = \frac{\mu F_z}{2ac\tan\alpha}$. With this we obtain the quadratic formula

$$F_y = \mu G_z - \frac{\mu^2 F_z^2}{8ca^2\tan(\alpha)}. \tag{2.2.18}$$

2.2.2 Magic formula tyre model

While the analytical models we discussed so far are physically intuitive, they fail to accurately replicate tyre behaviour at large slip angles. We can see an illustration of the desired behaviour in Figure 2.5. The most significant factors which lead to differences between the model predictions and measured tyre performance are

- unequal stiffness in x and y directions
- non-symmetric and non-constant pressure distribution.
- non-constant friction coefficient, including a difference between static and kinetic friction coefficients.

One option we have for dealing with these factors is to include them directly in a more complex physical model. In the following we will chose an alternative approach, which is to use a carefully chosen approximate function which is able to replicate the desired measurements well. This choice leads to a so called semi-empirical model. The magic formula model, developed by Pacejka and Bakker [70] provides such a model for computing the lateral and longitudinal tyre forces F_y, F_x, as well as the aligning moment M_z for a wide range of operating conditions. We first consider the simpler case where only either a lateral or longitudinal force is generated. The force generated is denoted by Y and is the output variable of our model, subject to the input variable X. In this simplified case of the magic formula tyre model, we proceed

to compute Y by

$$Y(X) = y(x) + S_v, \tag{2.2.19}$$

$$y(x) = D \sin\left(C \arctan\left(Bx - E\left(Bx - \arctan(Bx)\right)\right)\right), \tag{2.2.20}$$

$$x = X - S_h. \tag{2.2.21}$$

In Equation (2.2.19), Y represents the model output, such as longitudinal or lateral tyre force F_x, F_y, or the aligning moment M_z. In (2.2.19, 2.2.20, 2.2.21) the parameters B, C, D, E, S_v, S_h appear and have impact on the tyre model, we find the definitions of these parameters in Table 2.1. The various parameters in the Pacejka Magic

Table 2.1: Pacejak magic formula parameters [70]

Parameter	Description
B	stiffness factor
C	shape factor
D	peak value
E	curvature factor
S_h	horizontal shift
S_v	vertical shift

Formula ([70]) model are functions of the road friction coefficient μ, the normal load F_z, the wheel camber angle, as well as other parameters which need to be determined for each tyre. From Equation (2.2.20) we see that D describes the peak tyre force or alignment moment, while the product BCD corresponds to the slope at the origin $x = y = 0$. For $\alpha \to \infty$ we see that y_s is the asymptotic limit of the model output and the shape factor C determines the range of the sine function

$$C = \frac{2}{\pi} \arcsin\left(\frac{y_s}{D}\right).$$

For the derivation of tyre dependent parameters we refer to the original work by Pacejka and Bakker [70] and only present the resulting formulas. To complete the

model we have

$$D = a_1 F_z^2 + a_2 F_z \tag{2.2.22}$$

$$BCD = \begin{cases} a_3 \sin\left(a_4 \arctan(a_5 F_z)\right), & \text{for lateral force} \\ \exp(-a_5 F_z)\left(a_3 F_z^2 + a_4 F_z\right), & \text{for longitudinal force} \end{cases} \tag{2.2.23}$$

$$E = a_6 F_z^2 + a_7 F_z + a_8, \tag{2.2.24}$$

where $a_1, a_2, \ldots, a_8$ are constants that we need to determine on a tyre by tyre basis.

2.2.3 Dugoff's tyre model

Dugoff's tyre model (see [32]) is an alternative to the analytical tyre model we discussed at the beginning of Section 2.2.1. We introduce Dugoff's tyre model for both its historical importance and the ability to model combined lateral and longitudinal force generation. Compared to the Magic Formula model, we discussed earlier Dugoff's model has the advantage that it is derived from force and moment balances. Additionally, the lateral and longitudinal friction forces are directly related to road friction coefficients in a more transparent manner.

For this, we let σ_x be the longitudinal slip ratio and α be the lateral slip angle. We will only describe the final form of Dugoff's tyre model and refer the reader to [32] for the derivation. The lateral and longitudinal tyre forces generated in this models

are

$$\lambda = \frac{\mu F_z (1 + \sigma_x)}{2\sqrt{C_\sigma^2 \sigma_x^2 + C_\alpha^2 \tan(\alpha)^2}}, \tag{2.2.25}$$

$$f(\lambda) = \begin{cases} \lambda(2 - \lambda) & \text{if } \lambda \leq 1, \\ 1 & \text{otherwise,} \end{cases} \tag{2.2.26}$$

$$F_x = C_\sigma \frac{\sigma_x}{1 + \sigma_x} f(\lambda), \tag{2.2.27}$$

$$F_y = C_\alpha \frac{\tan(\alpha)}{1 + \sigma_x} f(\lambda). \tag{2.2.28}$$

As demonstrated in [40] the Dugoff model can be interpreted as a friction circle model. In friction circle models, the total generated force, in lateral and longitudinal direction is bounded by the friction force in the sense that

$$\sqrt{F_x^2 + F_y^2} \leq \mu F_z.$$

To establish the connection between Dugoff's models and a friction circle we first compute the maximal force values, assuming an unlimited friction coefficient μ, such that

$$F_x^{\max} = C_\sigma \frac{\sigma_x}{1 + \sigma_x},$$
$$F_y^{\max} = C_\alpha \frac{\tan \alpha}{1 + \sigma_x}.$$

With this we extract the theoretical friction coefficient $\mu^{\max}$ required to achieve $F_x^{\max}, F_y^{\max}$

$$\mu^{\max} = \frac{1}{F_z} \sqrt{(F_x^{\max})^2 + \left(F_y^{\max}\right)^2}.$$

We see that the case $\lambda > 1$ in Equation (2.2.25), is equivalent to the resulting forces using less than half of the available friction force $\sqrt{F_x^2 + F_y^2} < \frac{1}{2}\mu F_z$, and the resulting forces are not reduced

$$F_x = F_x^{\max}, \qquad F_y = F_y^{\max}.$$

The case $\lambda < 1$ is equivalent to the point (F_x, F_y) being outside the friction circle. Following Equations (2.2.27), (2.2.28) the tyre forces are

$$F_x = F_x^{\max}\frac{\mu}{\mu^{\max}}\left(1 - \frac{\mu}{4\mu^{\max}}\right),$$
$$F_y = F_y^{\max}\frac{\mu}{\mu^{\max}}\left(1 - \frac{\mu}{4\mu^{\max}}\right).$$

All three tyre models we presented rely on the available friction force μF_z. The downward force F_z consists both of a vehicle weight component and an aerodynamic component. We use a simple model to describe the aerodynamic component of F_z as

$$F_z^{\text{aerodynamic}} = \frac{1}{2}c_{z,\text{aero}}\rho A v_x^2, \tag{2.2.29}$$

where ρ is the air density and A is the frontal area.

2.3 Full vehicle model

With the models presented in Section 2.2.1, we can describe the forces acting on the front and rear wheels. Unless otherwise stated we will use the analytically derived linear tyre model (2.2.17) for our computations. The last assumption which we require for a full bicycle model is that the front axis is not driven and that we can directly control the rear wheel driving force $F_{r,\text{throttle}}$. We will denote the total longitudinal force by $F_{r,x}$. $F_{r,x}$ includes $F_{r,\text{throttle}}$ as well as rolling and aerodynamic resistance

forces, which we denote by $F_{r,\text{roll}}$ and $F_{r,\text{aero}}$. We determine $F_{r,\text{roll}}$ and $F_{r,\text{aero}}$ by

$$\begin{aligned} F_{r,\text{roll}} &= -c_r v_x, \\ F_{r,\text{aero}} &= -c_a \rho A_{\text{front}} v_x^2, \\ F_{r,x} &= F_{r,\text{throttle}} + F_{r,\text{roll}} + F_{r,\text{roll}}, \end{aligned} \tag{2.3.1}$$

where c_r, c_a are rolling and aerodynamic drag constants, which we explain in Table 2.2. With these forces we apply Newton's second law and derive the equations of motion

Table 2.2: Resistance force constants

Variable	Symbol	Value
Rolling resistance constant	c_r	0.01
Aerodynamic drag constant	c_d	1.7

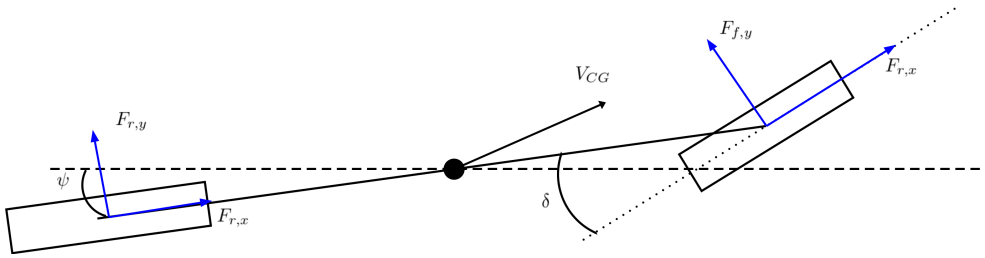


Figure 2.8: Bicycle model with tyre forces

in our vehicle model. A force balance in the vehicle aligned reference frame yields that

$$m \cdot \mathbf{a}_V = \mathbf{F}_{\text{total}}, \tag{2.3.2}$$

$$= \begin{pmatrix} \cos(\delta) & \sin(\delta) \\ -\sin(\delta) & \cos(\delta) \end{pmatrix} \begin{pmatrix} F_{f,x} \\ F_{f,y} \end{pmatrix} + \begin{pmatrix} F_{r,x} \\ F_{f,y} \end{pmatrix}, \tag{2.3.3}$$

where $\mathbf{a}_V$ is the acceleration vector in the vehicle aligned reference frame. We subsequently apply the appropriate rotation matrix

$$R(\tau) := \begin{pmatrix} \cos(\tau) & \sin(\tau) \\ -\sin(\tau) & \cos(\tau) \end{pmatrix}, \tag{2.3.4}$$

to derive the acceleration vector in the global frame of reference as

$$\begin{pmatrix} a_x^{\mathrm{g}} \\ a_y^{\mathrm{g}} \end{pmatrix} = R(\psi) \begin{pmatrix} a_x \\ a_y \end{pmatrix} = \frac{R(\psi)}{m} \left[R(\delta) \begin{pmatrix} F_{f,x} \\ F_{f,y} \end{pmatrix} + \begin{pmatrix} F_{r,x} \\ F_{f,y} \end{pmatrix} \right]. \tag{2.3.5}$$

With the acceleration from Equation (2.3.5), we can write the equations motion for the vehicle's centre of mass position. As before, the global reference frame velocities are denoted by $(v_x^{\mathrm{g}}, v_y^{\mathrm{g}})$. This far we derived the time derivatives of position and velocity components x', y', v_x', v_y' of our vehicle model as

$$\begin{aligned} \partial_t \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} &= \begin{pmatrix} v_x^{\mathrm{g}}(t) \\ v_y^{\mathrm{g}}(t) \end{pmatrix} = R(\psi(t)) \begin{pmatrix} v_x \\ v_y \end{pmatrix}, \\ \partial_t \begin{pmatrix} v_x(t) \\ v_y(t) \end{pmatrix} &= \frac{d}{dt} \left(R(-\psi) \begin{pmatrix} v_x^{\mathrm{g}} \\ v_y^{\mathrm{g}} \end{pmatrix} \right) = \begin{pmatrix} a_x \\ a_y \end{pmatrix} + \begin{pmatrix} \omega(t) v_y \\ -\omega(t) v_x \end{pmatrix}. \end{aligned}$$

To complete the model, we need to add the corresponding equations for rotational motion. For this, we need to introduce the concept of a *moment of inertia*. According to [4], the moment of inertia, or *angular mass*, of a physical object determines the *torque* required to achieve one unit of *angular acceleration*. Torque or *rotational force* is the rotational equivalent to linear force and is, in three dimensions, defined as the cross product between the force vector and the lever arm vector

$$\boldsymbol{\tau} = \mathbf{r} \times \mathbf{F}, \tag{2.3.6}$$

and hence the magnitude of torque vector for a level arm and force vector with a relative angle of θ is given by

$$\tau = |\mathbf{r}||\mathbf{F}|\sin(\theta). \tag{2.3.7}$$

This lets us derive the equation for rotational acceleration

$$\psi''(t) = \omega'(t) = \frac{1}{I_{zz}}\left[a\cos(\delta)F_{f,y} - bF_{r,y}\right], \tag{2.3.8}$$

where I_{zz} is the yaw moment of inertia. With this we complete the first order dynamical system

$$\frac{d}{dt}\begin{pmatrix}\psi(t)\\ \omega(t)\end{pmatrix} = \begin{pmatrix}\omega(t)\\ \frac{1}{I_{\mathrm{zz}}}\left[a\cos(\delta)F_{\mathrm{f,\,y}}(t) - bF_{\mathrm{r,\,y}}\right]\end{pmatrix}. \tag{2.3.9}$$

This lets us write a full model for the vehicle position and motion as

$$\frac{d}{dt}\begin{pmatrix}x(t)\\ y(t)\\ \psi(t)\\ v_x(t)\\ v_y(t)\\ \omega(t)\end{pmatrix} = \begin{pmatrix}\cos(\psi)v_x(t) - \sin(\psi)v_y(t)\\ \cos(\psi)v_y(t) + \sin(\psi)v_x(t)\\ \omega\\ \frac{1}{m}\left(F_{r,x} - \sin(\delta)F_{f,y}\right) + \omega(t)v_y\\ \frac{1}{m}\left(\cos(\delta)F_{f,y} + F_{r,y}\right) - \omega(t)v_x\\ \frac{1}{I_{zz}}\left[a\cos(\delta)F_{f,y}(t) - bF_{r,y}\right]\end{pmatrix} \tag{2.3.10}$$

Dynamics Equation (2.3.10) lets us see the connection between the steering angle δ and velocity states. The rear wheel force $F_{r,x}$ depends on the rear wheel driving force

$F_{r,\text{throttle}}$ by the relationship given in Equation (2.3.1)

$$F_{r,x} = F_{r,\text{throttle}} + F_{r,\text{roll}} + F_{r,\text{aero}},$$
$$F_{r,x} = F_{r,\text{throttle}} - c_r v_x - c_a \rho A_{\text{front}} v_x^2.$$

In addition, to the state description in Equation (2.3.10) it will be useful to have further information about the vehicles position relative to the race track. This position description will allow us to enforce the track constraint in a linear manner.

2.3.1 Track model

In Section 2.1 we introduced the bicycle model and derived the equations of motion, which a vehicle under this model follows. In addition to these equations of motion, we need to model the vehicle position relative to the racetrack. A natural restriction in motor sports is to remain within the track boundaries. In the following, we will denote the tracks length by L and describe the position of the centre line as a two-dimensional curve

$$c : [0, L] \mapsto \mathbb{R}^2. \tag{2.3.11}$$

The vehicles position along this centre line curve at time t will then be given by $s(t) \in [0, T]$. The change of s will depend on the vehicles distance to the centre line $\nu(t)$ and the vehicles rotation relative to the track centre line $\xi(t) \in \mathbb{R}$. In this description, $\nu(t) > 0$ will then denote that the vehicle is located left of the centre line, when the observer is aligned with the centre line. Conversely, $\nu < 0$ describes a vehicle left of the centre line. To ensure that the vehicle does not leave the racetrack

we enforce the constraint

$$\nu(t) \in \left[-\frac{W}{2}, \frac{W}{2}\right]. \tag{2.3.12}$$

We now derive the differential equations for $s(\cdot)$ and $\nu(\cdot)$. We let $\alpha(x)$ denote the track orientation at position $x \in [0, L]$, such that

$$c(s) = c_0 + \int_0^s \left(\cos(\alpha(x)), \sin(\alpha(x))\right)^T, dx \tag{2.3.13}$$

The change in track orientation is called the track curvature $K(x) = \alpha'(x)$, where $x \in (0, L)$. The vehicle position can then be described both in absolute and in curvilinear coordinates. For this description, we see that the orthogonal vector to the track is given by

$$\delta(x) = \begin{pmatrix} \cos(\alpha(x) + \frac{\pi}{2}) \\ \sin(\alpha(x) + \frac{\pi}{2}) \end{pmatrix} = \begin{pmatrix} -\sin(\alpha(x)) \\ \cos(\alpha(x)) \end{pmatrix}. \tag{2.3.14}$$

The track position then satisfies

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = c(s(t)) + \nu(t)\delta(s(t)) \tag{2.3.15}$$

Taking the first derivatives in Equation (2.3.15), while considering the vehicle orientated reference frame, yields that

$$\begin{pmatrix} \cos(\psi(t)) & -\sin(\psi(t)) \\ \sin(\psi(t)) & \cos(\psi(t)) \end{pmatrix} \begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix} = \partial_s c(s)\partial_t s(t) + \partial_t \nu(t)\delta(s) + \nu(t)\partial_s \delta(s)\partial_t s(t) \tag{2.3.16}$$

Inserting the description of centre position from Equation (2.3.13) and the orthogonal

vector description (2.3.14) we find that

$$\begin{pmatrix} \cos(\alpha)(1-\nu(t)K(s)) & -\sin(\alpha) \\ \sin(\alpha)(1-\nu(t)K(s)) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} \partial_t s(t) \\ \partial_t \nu(t) \end{pmatrix} = \begin{pmatrix} \cos(\psi(t)) & -\sin(\psi(t)) \\ \sin(\psi(t)) & \cos(\psi(t)) \end{pmatrix} \begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix}. \tag{2.3.17}$$

Solving the System (2.3.17) for the track variables $s(t), \nu(t)$ we find that

$$\begin{pmatrix} \partial_t s(t) \\ \partial_t \nu(t) \end{pmatrix} = \begin{pmatrix} \frac{1}{1-\nu(t)K(s(t))} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}^{-1} \begin{pmatrix} \cos(\psi(t)) & -\sin(\psi(t)) \\ \sin(\psi(t)) & \cos(\psi(t)) \end{pmatrix} \begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix}. \tag{2.3.18}$$

The vehicles angle relative to the track centre line is $\xi(t) = \psi(t) - \alpha(s(t))$. Taking first derivatives yields

$$\xi'(t) = \psi'(t) - \alpha'(s(t))s'(t) = \psi'(t) - K(s(t))s'(t). \tag{2.3.19}$$

Using Equation (2.3.19), we simplify Equation (2.3.18) to

$$\partial_t s(t) = \frac{x'(t)\cos(\xi) - y'(t)\sin(\xi)}{1-\nu(t)K(s(t))}, \tag{2.3.20}$$

$$\partial_t \nu(t) = x'(t)\sin(\xi) + y'(t)\cos(\xi). \tag{2.3.21}$$

With this track model, we can track the vehicles motion around the track and effectively enforce the track constraint within an optimal control problem setting. We thus extend the vehicle model from Equation (2.3.10) including the track variables

and obtain

$$\frac{d}{dt}\mathbf{x}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ y(t) \\ \psi(t) \\ v_x(t) \\ v_y(t) \\ \omega(t) \\ s(t) \\ \nu(t) \\ \xi(t) \end{pmatrix} = \begin{pmatrix} \cos(\psi)v_x(t) - \sin(\psi)v_y(t) \\ \cos(\psi)v_y(t) + \sin(\psi)v_x(t) \\ \omega \\ \frac{1}{m}\left(F_{r,x} - \sin(\delta)F_{f,y}\right) + \omega(t)v_y \\ \frac{1}{m}\left(\cos(\delta)F_{f,y} + F_{r,y}\right) - \omega(t)v_x \\ \frac{1}{I_z z}\left[a\cos(\delta)F_{f,y}(t) - bF_{r,y}\right] \\ \frac{v_x\cos(\xi) - v_y\sin(\xi)}{1-\nu(t)K(s(t))} \\ v_x(t)\sin(\xi) + v_y\cos(\xi) \\ \psi'(t) - K(s(t))s'(t) \end{pmatrix}. \tag{2.3.22}$$

2.3.2 Track design

To apply the track model from Section 2.3.1, we require a description of the track centre line via the orientation function $\alpha : [0, L] \to \mathbb{R}$ or the curvature function $K : [0, L] \to \mathbb{R}$, where $K(x) = \alpha'(x)$. Such a description is not naturally available for typical race tracks. We therefore need to generate the track description, by tracing a given race track using appropriate software tool. For this, we developed a software, which allows use to trace race-tracks, based of image data, such as satellite photographs, into descriptions which our track model can use.

In the following, we approximate two dimensional race-tracks using straights and uniform curves. A straight track-element has single length parameter and a curve-element has two parameters, one for the angular distance in radians and a curve radius. To construct a track, we define a starting point by a position

$$p(0) = \begin{pmatrix} p_{x,0} \\ p_{y,0} \end{pmatrix},$$

a starting direction ω_0 and a track width $W > 0$. A track is then defined as a series of track elements. Using a specialised software we can create sample tracks by placing appropriate track elements over a reference image. Figure 2.9 shows the resulting workflow. The track elements shown in green in Figure 2.9 are already saved and the red element is currently in edit mode and can be moved using the mouse cursor.

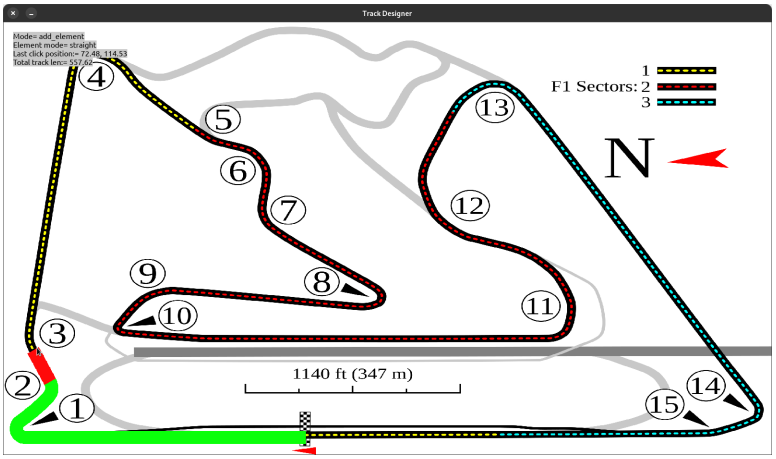


Figure 2.9: Track modelling software

Chapter 3

Collocation methods for optimal control problems

In Chapter 2, we discussed the necessary assumptions for the bicycle model and derived the equations of motion. In this chapter, we expand on this by introducing direct collocation methods and showing how we can use them to solve minimal lap time problems. In collocation methods we choose a finite dimensional function basis to represent the state $\mathbf{x}$ and control $\mathbf{u}$ vectors, as well as a set of collocation nodes $\omega_0, \ldots, \omega_N$. The dynamics equation

$$\frac{d}{dt}\mathbf{x}(t_{\omega_i}) = \mathbf{f}\left(\mathbf{x}(t_{\omega_i}), \mathbf{u}(t_{\omega_i})\right) \quad \forall i \in 0, \ldots, N, \tag{3.0.1}$$

$$\mathbf{x}(t_{\omega_i}) = \mathbf{x}_{\text{init}} \tag{3.0.2}$$

is then imposed at these collocation nodes (see [9]). The concrete implementation of a collocation problem depends on the selection of a quadrature rule to represent equation (3.0.1).

3.1 Control problems in Lagrange from

We commence the discussion of collocation methods by introducing the *Lagrange form* of optimal control problems [13, 93, 81]. A control problem is said to be in Lagrange form, when the cost functional J only includes a running cost term. Costs contributed by the initial and final states are not included. The cost functional in Lagrange form can therefore be written as

$$J(\mathbf{u}, t_{\omega_0}, t_{\omega_N}) = \int_{t_{\omega_0}}^{t_{\omega_N}} L(\mathbf{x}(t), \mathbf{u}(t), t)\, dt. \tag{3.1.1}$$

Such a formulation is particularly important for minimal lap time problems as it is natural to include a cost function for each time unit required to reach a desired terminal state $\mathbf{x}_T$ from the initial state $\mathbf{x}_0$. Other common optimal control problem formulations are the *Mayer* and *Bolza* forms. Problems in Mayer form only include a final state dependent cost term

$$J(\mathbf{u}, t_{\omega_0}, t_{\omega_N}) = \Phi(\mathbf{x}(t_N)),$$

while problems in Bolza form consist of both running cost and final state cost terms, such as

$$J(\mathbf{u}, t_{\omega_0}, t_{\omega_N}) = \int_{t_{\omega_0}}^{t_{\omega_N}} L(\mathbf{x}(t), \mathbf{u}(t), t)\, dt + \Phi(\mathbf{x}(t_N)).$$

For the goal of performing minimal lap time simulations, we focus our attention to problems in Lagrange form. Similar to before, we assume that the system evolves subject to the autonomous dynamics equation given by $\mathbf{f}$, such that

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \quad \forall t \in (0, T). \tag{3.1.2}$$

Furthermore, we assume that the dynamical system develops, subject to the path constraints

$$\mathbf{C}(\mathbf{x}, \mathbf{u}) \geq \mathbf{0}, \tag{3.1.3}$$

where $\mathbf{x}$ and $\mathbf{u}$ are the state and control vector functions, respectively. The resulting problem in Lagrange form is then

$$\begin{aligned} &\min_{T,\mathbf{x},\mathbf{u}} \int_0^T \mathcal{L}(\mathbf{x}(t), \mathbf{u}(t))\, dt, \\ &\text{subject to: } \mathbf{x} \in L^\infty(0, T; \mathbb{R}^{d_x}), \quad \mathbf{u} \in L^\infty(0, T; \mathbb{R}^{d_u}), \\ &\mathbf{u}_m \leq \mathbf{u} \leq \mathbf{u}_M, \\ &\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)), \\ &\mathbf{C}(\mathbf{x}, \mathbf{u}) \geq \mathbf{0}. \end{aligned} \tag{3.1.4}$$

In the following section, we introduce a direct method to approximate a solution to this problem.

3.2 Trapezoidal Collocation

In Section 3.1, we introduced optimal control problems in Lagrange form (3.1.4). Using this description, we now apply trapezoidal quadrature to derive a collocation method. We consider the dynamics equation

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), \tag{3.2.1}$$

$$\mathbf{x}(0) = \mathbf{x}_{\text{init}}. \tag{3.2.2}$$

To derive a numerical approximation we divide the time interval $(0, T)$ into the subintervals $\{T_i = (t_{i-1}, t_i) \subset (0, T) : i = 1, \ldots, N, t_0 = 0, t_N = T\}$. Then, we apply the

trapezoidal quadrature rule

$$g(t+h) = g(t) + \int_t^{t+h} g'(\tau)d\tau \tag{3.2.3}$$

$$= g(t) + \frac{h}{2}\left(g'(t) + g'(t+h)\right) - \frac{h^3}{12}g''(\xi) \tag{3.2.4}$$

$$= g(t) + \frac{h}{2}\left(g'(t) + g'(t+h)\right) + \mathcal{O}(h^3) \tag{3.2.5}$$

to integrate $\mathbf{x}$ over each of the sub-intervals T_i, $i = 1, \ldots, N$. We thus derive the discrete dynamics system

$$\mathbf{x}_i = \mathbf{x}_{i-1} + \frac{h}{2}\left(\mathbf{f}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1}) + \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i)\right), \tag{3.2.6}$$

$$\mathbf{x}_0 = \mathbf{x}_{\text{init}}, \tag{3.2.7}$$

$$\mathbf{u}_m \leq \mathbf{u}_i \leq \mathbf{u}_M \quad \forall i = 1, \ldots, N. \tag{3.2.8}$$

For each pair of track nodes, as described in Section 2.3.1, we add the discrete dynamics equation as a constraint to the non-linear system, resulting in the non-linear equality constraint

$$\begin{bmatrix} -I & I & 0 & \ldots & 0 \\ 0 & -I & I & & \vdots \\ & & \ddots & \ddots & \\ & \ldots & 0 & -I & I \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix} = \frac{1}{2} \begin{bmatrix} h_1\left(\mathbf{f}(\mathbf{x}_0, \mathbf{u}_0) + \mathbf{f}(\mathbf{x}_1, \mathbf{u}_1)\right) \\ h_2\left(\mathbf{f}(\mathbf{x}_1, \mathbf{u}_1) + \mathbf{f}(\mathbf{x}_2, \mathbf{u}_2)\right) \\ \vdots \\ h_N\left(\mathbf{f}(\mathbf{x}_{N-1}, \mathbf{u}_{N-1}) + \mathbf{f}(\mathbf{x}_N, \mathbf{u}_N)\right) \end{bmatrix}. \tag{3.2.9}$$

Equation (3.2.9) provides us with a single constraint for each state and each domain segment. While many problems benefit from specifying the starting condition $\mathbf{x}_0 = \mathbf{x}_{\text{init}}$ directly, we choose the more flexible approach of enforcing bounds on $\mathbf{x}_0$. This option allows us to restrict the vehicles starting position to be any position on the initial nodes, while not specifying an exact state including location, yaw and velocity. A benefit of trapezoidal collocation is that the collocation nodes coincide with the

knot points of the quadrature method. Both sets of points are located at the segment boundaries $\{(t_0, t_1), \ldots, (t_{N-1}, t_N)\}$. For trapezoidal quadrature, we approximate the state derivative by a piecewise linear function

$$\mathbf{x}'(t) \approx \mathbf{p}(t) = \mathbf{f}_{i-1} + \frac{t - t_{i-1}}{t_i - t_{i-1}} \left(\mathbf{f}_i - \mathbf{f}_{i-1}\right).$$

Using this linear approximation for the state derivative, we obtain the continuous description for the state interpolation

$$\mathbf{x}(t) = \mathbf{x}_{i-1} + (t - t_{i-1})\mathbf{f}_{i-1} + \frac{(t - t_{i-1})^2}{t_i - t_{i-1}} \left(\mathbf{f}_i - \mathbf{f}_{i-1}\right)$$

so that the collocation condition

$$\mathbf{x}'(t_i) = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) = \mathbf{f}_i$$

is naturally satisfied by enforcing the quadrature condition in Equation (3.2.9). We compare this to the mid-point collocation rule, where we enforce the dynamics constraint at the centre point of a domain segment. For this, we consider the time segment (t_{i-1}, t_i) with the centre point $t_{i-1/2} = \frac{t_i + t_{i-1}}{2}$ and let $\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{u}_{i-1/2}$ be the decision variables associated with our domain segment. To apply the mid-point rule, we need to enforce that our state interpolate satisfies

$$\mathbf{x}'(t_{i-1/2}) = \mathbf{f}\left(\mathbf{x}(t_{i-1/2}), \mathbf{u}(t_{i-1/2})\right), \tag{3.2.10}$$

while simultaneously enforcing the quadrature condition

$$\mathbf{x}_i = \mathbf{x}_{i-1} + (t_i - t_{i-1})\, \mathbf{f}(\mathbf{x}(t_{i-1/2}), \mathbf{u}_{i-1/2}). \tag{3.2.11}$$

We note that Equation (3.2.11) uses both the discrete decision variables $\mathbf{x}_i, \mathbf{x}_{i-1}, \mathbf{u}_{i-1/2}$ as well as the interpolated state value $\mathbf{x}(t_{i-1/2})$. By Equation (3.2.10), we only have one data point for the state derivative, so that we use the approximation

$$\mathbf{x}(t) = \mathbf{x}_{i-1} + \frac{t - t_{i-1}}{t_i - t_{i-1}} \mathbf{f}(\mathbf{x}_{i-1/2}, \mathbf{u}_{i-1/2}). \tag{3.2.12}$$

To do this, we need to include the decision variable $\mathbf{x}_{i-1/2}$ to our problem. Thus, to apply the mid-point quadrature rule we need to enforce both the derivative condition from Equation (3.2.10) and the quadrature condition (3.2.11). Figure 3.1 shows two adjacent track segments with nodes $\mathbf{n}_1, \ldots, \mathbf{n}_5$. The shading in Figure 3.1 indicates that the nodes $\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$ are associated with the first segment and $\mathbf{n}_3, \mathbf{n}_4, \mathbf{n}_5$ are associated with the second. With the preceding discussion, we can construct apply the mid-point rule by introducing state variables at nodes $\mathbf{n}_1, \ldots, \mathbf{n}_5$ and control variables at the central nodes $\mathbf{n}_2, \mathbf{n}_4$.

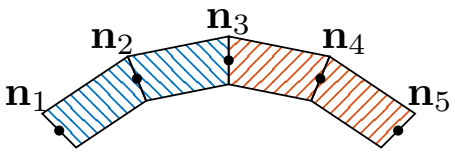


Figure 3.1: Track segments for the mid-point quadrature rule

For our application, the fact that the trapezoidal rule does not have collocation nodes within a domain segment provides a further benefit. This is because we will switch the quadrature domain from a temporal one to a spatial one. A further benefit of the trapezoidal rule comes from the fact that no collocation nodes are placed within a segment. During the track discretisation, we place collocation nodes following the track-distance variable s, as defined in Section 2.3.1. As a consequence, the time variables, which we use for quadrature, do not necessarily match the node locations required by the quadrature method. For this we will again consider the mid-point

quadrature rule. On the first segment, we have the three nodes, consisting of state variables, control variables and a time variable

$$\mathbf{n}_0 = (t_0, \mathbf{x}_0), \quad \mathbf{n}_1 = (t_1, \mathbf{x}_1, \mathbf{u}_1), \quad \mathbf{n}_2 = (t_2, \mathbf{x}_2).$$

In accordance with the bicycle vehicle model, the state component at node $\mathbf{n}_i$ contains the position, yaw, velocity, and track variables

$$\mathbf{x}_i = (x_i, y_i, \psi, x_i', y_i', \psi_i', s_i, \nu_i, \xi_i).$$

By the construction of the track grid, we have know what $s_1 = \frac{1}{2}(s_0 + s_2)$. We want to ensure the quadrature condition

$$\mathbf{x}_2 = \mathbf{x}_0 + \int_{t_0}^{t_1} \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))\, dt, \tag{3.2.13}$$

where we approximate the analytical integral using the mid-point quadrature rule and obtain the condition

$$\mathbf{x}_2 = \mathbf{x}_0 + (t_2 - t_0)\mathbf{f}\left(\frac{t_2 + t_0}{2}\right), \tag{3.2.14}$$

where we have used the shortened notation

$$\mathbf{f}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)).$$

Due to the spatial discretisation, we can generally not expect the time grid to align with our collocation method in the sense that

$$t_1 = \frac{1}{2}(t_0 + t_2).$$

We have two options to evaluate $\mathbf{f}\left(\frac{t_2+t_0}{2}\right)$. The first option is to transform the quadrature domain. For this, we assume that we can construct a function $T : [0, L] \to \mathbb{R}_{\geq 0}$, which maps the track-distance variable s onto the domain and satisfies $\xi = s(T(\xi))$ for all $\xi \in [0, L]$. Using a variable transformation, we obtain

$$\begin{aligned} &\mathbf{x}_0 + \int_{t_0}^{t_2} \mathbf{f}\left(\mathbf{x}(t), \mathbf{u}(t)\right)\, dt \\ =&\mathbf{x}_0 + \int_{T((s_0,s_2))} \mathbf{f}\left(\mathbf{x}(t), \mathbf{u}(t)\right)\, dt \\ =&\mathbf{x}_0 + \int_{s_0}^{s_2} \mathbf{f}\left(\mathbf{x}(T(s)), \mathbf{u}(T(s))\right) |s'(T(s))|^{-1}\, dt. \end{aligned} \tag{3.2.15}$$

We are now able use the mid-point quadrature rule to approximate the integral in Equation (3.2.15) and obtain

$$\mathbf{x}_2 = \mathbf{x}_0 + (s_2 - s_0)\, \mathbf{f}(s_1)|s'(t_1)|^{-1}. \tag{3.2.16}$$

The second option is to interpolate the derivative function $\mathbf{f} : [t_0, t_2]$ using the available data and an appropriate polynomial order. To perform this we choose a polynomial basis $p_0, p_1, p_2 \in \mathbb{P}_2([-1, 1])$ and let

$$\tilde{\mathbf{f}}(t) = \sum_{k=0}^{2} \mathbf{a}_k p_k \circ \Phi^{-1}(t), \tag{3.2.17}$$

where

$$\Phi : [-1, 1] \to [t_0, t_2], \quad \Phi(x) = t_0 + \frac{x+1}{2}(t_2 - t_0).$$

Letting $\tau_i = \Phi^{-1}(t_i)$ we can solve the Vandermonde system

$$\underbrace{\begin{pmatrix} p_0(\tau_0) & p_1(\tau_0) & p_2(\tau_0) \\ p_0(\tau_1) & p_1(\tau_1) & p_2(\tau_1) \\ p_0(\tau_2) & p_1(\tau_2) & p_2(\tau_2) \end{pmatrix}}_{=:V} \begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}, \tag{3.2.18}$$

and find

$$\mathbf{f}\left(\frac{t_2 + t_0}{2}\right) = \begin{pmatrix} p_0(\tau_1), & p_1(\tau_1), & p_2(\tau_1) \end{pmatrix} V^{-1} \begin{pmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}. \tag{3.2.19}$$

With this, we can evaluate the mid-point quadrature rule

$$\mathbf{x}_1 = \mathbf{x}_0 + \int_{t_0}^{t_2} \mathbf{f}(t)\, dt \approx \mathbf{x}_0 + (t_2 - t_0)\mathbf{f}\left(\frac{t_2 + t_0}{2}\right). \tag{3.2.20}$$

A consequence of the discussion of the mid-point rule is that collocation methods that have collocation nodes within elements require either a transformation from a time domain to a track distance domain, or dynamic interpolation of the dynamics $\mathbf{f}$. In the next section, we discuss higher order pseudo-spectral collocation methods.

3.3 Pseudo-spectral collocation methods

In the last decade, higher-order pseudo-spectral methods have seen a great popularity increase in the numerical solution of optimal control problems [37, 38, 29]. We note at this point that many authors use the term orthogonal collocation instead of pseudo-spectral collocation. The three most commonly used sets of collocation points are the *Legendre–Gauss* (LG), *Legendre–Gauss–Radau* (LGR), and *Legendre–Gauss–Lobatto* (LGL) points. Let P_n be the n-th degree Legendre polynomial, which are defined by

the orthogonality condition

$$\int_{-1}^{1} P_i(x)P_j(x)\,dx = 0 \quad \text{if } i \neq j, \tag{3.3.1}$$

and the scaling factor $P_n(1) = 1$. The collocation nodes for the three methods (LG), (LGR), and (LGL) are then

LG:	Roots of the N^{th}-degree Legendre polynomial P_N
LGR:	Roots of the N^{th}-degree polynomial $P_N + P_{N-1}$
LGL:	Roots of the $(N-2)^{\text{th}}$-degree Legendre polynomial P'_{N-1} and $\{-1, +1\}$

Legendre polynomials satisfy the recurrence relationship

$$P_{N+1}(x) = \frac{2N+1}{N+1}P_N(x) - \frac{N}{N+1}P_{N-1}(x). \tag{3.3.2}$$

The recurrence relationship (3.3.2), together with the initial conditions $P_0(x) \equiv 1, P_1(x) = x$ lets us show that $P_{N+1}(-1) = -P_N(-1)$, so that the LGR method contains a collocation node at the left interval bound -1. We can see the general node locations in Figure 3.2.

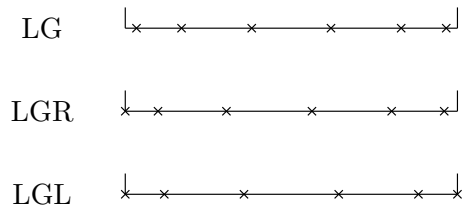


Figure 3.2: Collocation nodes for LG, LGR, LGL nodes using $N = 6$

3.3.1 Legendre–Gauss–Lobatto collocation

We first want to focus on Legendre–Gauss–Lobatto collocation, as this method was historically first developed and is consequentially well documented. For us, it has additional relevance as both the start and end-nodes are collocated. We construct the LGL method following [33]. Let $L_N(t)$ denote the Legendre polynomial of order N on the time interval $[t_0, t_N] = [-1, +1]$. We then choose the collocation nodes $t_0 = -1$, $t_N = +1$ and $\{t_m : m = 1, 2, \ldots, N-1\}$ as the roots of $L'_N(t)$, the first derivative of $L_N(t)$. Given these nodes, we construct the functions

$$\phi_j(t) := \frac{1}{N(N+1)L_N(t_j)} \cdot \frac{(t^2-1)L'_N(t)}{t - t_j} \quad \forall j = 0, \ldots, N, \tag{3.3.3}$$

which satisfy

$$\phi_j(t_i) = \delta_{i,j}. \tag{3.3.4}$$

For a function $F : [-1, +1] \to \mathbb{R}$ we are then able to write the N-th degree interpolating polynomial as

$$F_N(t) = \sum_{j=0}^{N} F(t_j)\phi_j(t). \tag{3.3.5}$$

Taking the derivative of F_N shows us that

$$\partial_t F_N(t) = \sum_{j=0}^{N} F(t_j)\partial_t\phi_j(t), \tag{3.3.6}$$

and hence, we can write the vector $F'_N(t)$, evaluated at the collocation nodes $\{t_0, t_1, \ldots, t_N\}$, as

$$\begin{pmatrix} F'_N(t_0) \\ \vdots \\ F'_N(t_N) \end{pmatrix} = \underbrace{\begin{pmatrix} \phi'_0(t_0) & \ldots & \phi'_N(t_0) \\ \vdots & \ldots & \vdots \\ \phi'_0(t_N) & \ldots & \phi'_N(t_N) \end{pmatrix}}_{=:D_N} \begin{pmatrix} F_N(t_0) \\ \vdots \\ F_N(t_N) \end{pmatrix}. \tag{3.3.7}$$

We can evaluate the pseudo-spectral derivative matrix D_N explicitly with the formula

$$D_N = (d_{i,j})_{i,j=0,\ldots,N}\,, \qquad d_{i,j} = \begin{cases} \frac{L_N(t_i)}{L_N(t_j)(t_i - t_l)}, \text{ if } i \neq j, \\ -\frac{N(N+1)}{4}, \text{ if } i = j = 0, \\ \frac{N(N+1)}{4}, \text{ if } i = j = N, \\ 0, \text{ otherwise.} \end{cases} \tag{3.3.8}$$

We now transcribe the dynamics equation by first approximating $\mathbf{x}$ and $\mathbf{u}$ using N-th degree polynomials

$$\mathbf{x}_N(t) = \sum_{j=0}^{N} \mathbf{x}(t_j)\phi_j(t), \qquad \mathbf{u}_N(t) = \sum_{j=0}^{N} \mathbf{u}(t_j)\phi_j(t). \tag{3.3.9}$$

Using Equation (3.3.7) we transcribe the dynamics equation (3.0.1) as

$$D_N \begin{pmatrix} \mathbf{x}_N(t_0) \\ \vdots \\ \mathbf{x}_N(t_N) \end{pmatrix} = \begin{pmatrix} \mathbf{f}(\mathbf{x}_N(t_0), \mathbf{u}_N(t_0)) \\ \vdots \\ \mathbf{f}(\mathbf{x}_N(t_N), \mathbf{u}_N(t_N)) \end{pmatrix}. \tag{3.3.10}$$

3.3.2 Legendre–Gauss collocation

We now compare the Legendre–Gauss–Lobatto collocation method from the previous section with collocation in Legendre–Gauss nodes. This gives us the freedom to

enforce continuity of states across interval bounds while not requiring continuous differentiability. Consequently, removing the boundary points $\{-1, 1\}$ from the interval allows us to model discontinuous controls. Compared to the Legendre–Gauss–Lobatto collocation in Section 3.3.1, we enforce the collocation condition at the roots of the $N - th$ degree Legendre polynomial $\{t_1, \ldots, t_N\}$. We choose the notation $t_0 = -1$, $t_{N+1} = +1$ although we do not enforce the dynamics equation at t_0, t_{N+1}. With this, we interpolate the state function from values at $t_0, t_1, \ldots, t_N$ so that

$$\mathbf{x}_N(t) = \sum_{j=0}^{N} a_j L_j(t), \qquad L_j(t) = \prod_{\substack{i=0 \\ i \neq j}}^{N} \frac{t - t_i}{t_j - t_i}, \tag{3.3.11}$$

which let us obtain the discrete derivative condition

$$\mathbf{x}'_N(t_i) = \sum_{j=1}^{N} a_j L'_j(t_i) = \mathbf{f}\left(\mathbf{x}(t_i), \mathbf{u}(t_i)\right). \tag{3.3.12}$$

To enforce this condition at every collocation node, we ensure that

$$\underbrace{\begin{pmatrix} L'_0(t_1) & L'_1(t_1) & \ldots & L'_N(t_1) \\ L'_0(t_2) & L'_1(t_2) & \ldots & L'_N(t_2) \\ & \vdots & & \\ L'_0(t_N) & L'_1(t_N) & \ldots & L'_N(t_N) \end{pmatrix}}_{=:D_N} \begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_N \end{pmatrix} = \begin{pmatrix} \mathbf{f}\left(\mathbf{x}(t_0), \mathbf{u}(t_0)\right) \\ \mathbf{f}\left(\mathbf{x}(t_1), \mathbf{u}(t_1)\right) \\ \vdots \\ \mathbf{f}\left(\mathbf{x}(t_N), \mathbf{u}(t_N)\right) \end{pmatrix}. \tag{3.3.13}$$

The $N \times (N-1)$ matrix D_N is called the Gauss pseudo-spectral differentiation matrix. To compute the vector valued interpolation coefficients $\mathbf{a}_0, \ldots, \mathbf{a}_N$, we need to solve

the linear system

$$\underbrace{\begin{pmatrix} L_0(t_0) & L_1(t_0) & \dots & L_N(t_0) \\ L_0(t_1) & L_1(t_1) & \dots & L_N(t_1) \\ L_0(t_2) & L_1(t_2) & \dots & L_N(t_2) \\ & \vdots & & \\ L_0(t_N) & L_1(t_N) & \dots & L_N(t_N) \end{pmatrix}}_{=:M} \begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_N \end{pmatrix} = \begin{pmatrix} \mathbf{x}(t_0) \\ \mathbf{x}(t_1) \\ \mathbf{x}(t_2) \\ \vdots \\ \mathbf{x}(t_N) \end{pmatrix}. \tag{3.3.14}$$

In addition to the linear system (3.3.13), we need to ensure that the quadrature rule is satisfied and that the state function is continuous across intervals. In the following, we will see that enforcing the quadrature rule gives us an easy way to ensure continuity across intervals as long as we include state values at the t_{N+1} node of our last interval.

We enforce continuity at the intersections of intervals. Let the points $t_1^{(0)}, \dots, t_{N^{(0)}}^{(0)} \in (-1, 1)$ be the collocated time points on the preceding interval, transformed to the reference interval $[-1, 1]$. We then enforce continuity via

$$\sum_{i=0}^{N^{(0)}} \mathbf{x}_{N^{(0)}}(t_i) L_i^{(0)}(1) = \sum_{i=0}^{N} \mathbf{x}(t_i) L_i(-1). \tag{3.3.15}$$

In case the current interval is the first or last collocation interval, we replace the continuity condition (3.3.15) with the natural conditions for initial values

$$\sum_{i=0}^{N} \mathbf{x}(t_i) L_i(-1) = \mathbf{x}_{\text{init}}, \tag{3.3.16}$$

and for terminal values

$$\sum_{i=0}^{N} \mathbf{x}(t_i) L_i(-1) = \mathbf{x}_{\text{terminal}}. \tag{3.3.17}$$

3.4 Minimal lap time optimisation problem

In Sections 3.2 and 3.3, we discussed practical methods of transcribing infinite dimensional optimal control problems into finite dimensional non-linear optimisation problems. We now use these methods, together with the dynamic bicycle model introduced in Section 2.1, to solve a minimal lap time simulation problem.

For this, we discuss methods for setting up the collocation problem practically and consider both the use of penalty methods using the `SciPy` and `IPOpt` software packages [95, 98]. When solving non-linear optimisation problems, such as optimal control problems we need to supply an initial guess to the optimiser package. The optimiser performance depends heavily on the quality of our initial guess [9]. In our case, it is difficult to construct a vehicle trajectory, which satisfies the non-linear system dynamics, while finishing the race track quickly. In addition to the system dynamics, we need to additionally satisfy the track constraints. To balance dynamics violation and trajectory duration while satisfying the track constraints, we construct a solution which follows the track centre line. We follow the track centre-line in both the position variables (x, y) and the orientation variable ψ at a constant speed of $10\frac{\mathrm{m}}{\mathrm{s}}$. Algorithm 1 provides a pseudocode description of this initialisation procedure, where $c : [0, L] \to \mathbb{R}^2$ and $\alpha : [0, L] \to [0, 2\pi)$ are functions describing the track centre position and orientation respectively. Using the initialisation from Algorithm 1, we construct an initial guess for all states, controls, and time variables the track nodes indexed by $i = 0, \ldots, N$. We arrange these variables into the matrices

$$X = \begin{bmatrix} \mathbf{x}_0^T \\ \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} \in \mathbb{R}^{N \times n}, \quad U = \begin{bmatrix} \mathbf{u}_0^T \\ \mathbf{u}_1^T \\ \vdots \\ \mathbf{u}_N^T \end{bmatrix} \in \mathbb{R}^{N \times m}, \quad T = \begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_N \end{bmatrix} \in \mathbb{R}^N.$$

Algorithm 1 Initialisation Construction

Require: $N \geq 1, \mathbf{c} : [0, L] \to \mathbb{R}^2, \alpha : [0, L] \to [0, 2\pi)$

for $i \in \{0, \ldots, N\}$ **do**
    $s_i \leftarrow L\frac{i}{N}$
    $t_i \leftarrow L\frac{i}{10N}$
    $(x_i, y_i)^T \leftarrow \mathbf{c}(s_i)$
    $\phi_i \leftarrow \alpha(s_i)$
    $(x_i', y_i')^T \leftarrow (10, 0)^T$
    $\phi_i' \leftarrow 0$
    $(\nu_i, \xi_i) \leftarrow (0, 0)^T$
    $\mathbf{x}_i \leftarrow (x_i, y_i, \ldots, \xi_i)^T$.
end for
return $\mathbf{x}, (t_0, \ldots, t_N)^T$

One particular property of a solution constructed by the initialisation Algorithm 1 is that the track constraints are not violated, while all variable bounds are satisfied. To see this, we first consider an example track with track nodes marked at equidistant intervals, as shown in Figure 3.3. At each of these nodes, we have the state and

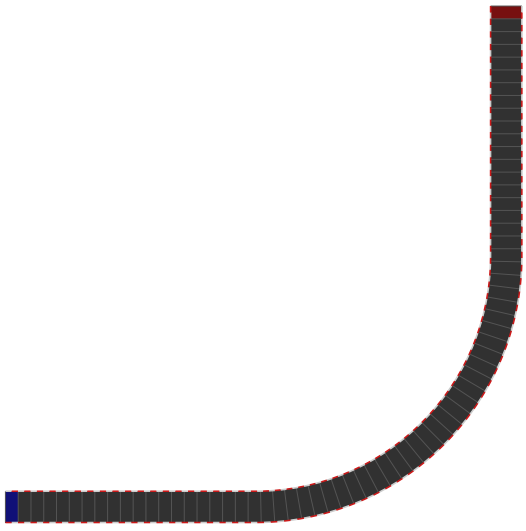


Figure 3.3: Test track with marked track nodes

control vectors

$$\mathbf{x}_i = (x_i, y_i, \psi_i, v_{x,i}, v_{y,i}, \psi_i', s_i, \xi_i, \nu_i),$$

$$\mathbf{u}_i = (f_{r,i}, \delta_i).$$

Our model describes the lateral vehicle position in two ways simultaneously. Once, by the centre line distance ν_i and once with the absolution position (x_i, y_i). While the model dynamics ensure that two methods of describing the vehicles position align, Algorithm 1 does not ensure that the model dynamics are followed exactly. Therefore, we need to ensure that both position descriptions are compatible with the track boundary condition. The condition

$$\frac{-W(s_i)}{2} \leq \nu_i \leq \frac{W(s_i)}{2} \tag{3.4.1}$$

directly ensures the track condition for ν_i. Algorithm 1 sets $\nu_i = 0$ for $i = 0, \ldots, N$, which directly satisfies condition 3.4.1. Secondly Algorithm 1, initialises the position description as

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \mathbf{c}(s_i),$$

which naturally satisfies the node condition

$$-\frac{W(s_i)}{2} \leq \left(\begin{pmatrix} x_i \\ y_i \end{pmatrix} - \mathbf{c}(s_i) \right)^T \begin{pmatrix} -\sin(\alpha(s_i)) \\ \cos(\alpha(s_i)) \end{pmatrix} \leq \frac{W(s_i)}{2}. \tag{3.4.2}$$

This not only ensures that the vehicle position (x_i, y_i) is contained in the track area associated with the centre line distance s_i, but also aligns ν_i and the position variables (x_i, y_i).

Bounds for the optimisation variables are given either by intuition of the underlying physical model, or by the track boundaries. For example, the bounds on x_i, y_i

are derived from Equation (3.4.2)

$$x_i \in \left[\mathbf{c}_x(s_i) - \left\| \frac{W(s_i)}{2\sin(\alpha(s_i))} \right\|, \mathbf{c}_x(s_i) + \left\| \frac{W(s_i)}{2\sin(\alpha(s_i))} \right\| \right], \tag{3.4.3}$$

$$y_i \in \left[\mathbf{c}_y(s_i) - \left\| \frac{W(s_i)}{2\cos(\alpha(s_i))} \right\|, \mathbf{c}_x(s_i) + \left\| \frac{W(s_i)}{2\cos(\alpha(s_i))} \right\| \right]. \tag{3.4.4}$$

We summarise these variable bounds by interpreting the inequalities component-wise

$$X_l \leq X \leq X_u, \quad U_l \leq U \leq U_u, \tag{3.4.5}$$

which lets us write the optimisation problem as

$$\begin{aligned} \min_{X,U,T} \quad & t_N - t_0, \\ \text{subject to: } & \mathbf{x}_i - \mathbf{x}_{i-1} = \frac{1}{2}(t_i - t_{i-1})\left(\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) + \mathbf{f}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1})\right) \quad \forall i, \\ & X_l \leq X \leq X_u, \\ & U_l \leq U \leq U_u, \\ & t_{i-1} < t_i \quad \forall i. \end{aligned} \tag{3.4.6}$$

With problem (3.4.6), we have a formulation of our main collocation problem using the trapezoidal rule. To apply higher order quadrature methods, we transform the problem from the time domain onto a track-oriented domain. We can optimise formulation (3.4.6) by reducing the number of state variables by removing the redundant position description

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} c_x(s(t)) \\ c_x(s(t)) \end{pmatrix} + \nu(t) \begin{pmatrix} \cos(\xi)(s(t)) \sin(\xi)(s(t)) \end{pmatrix}. \tag{3.4.7}$$

In the following, section we discuss this state transformation.

3.4.1 Track-oriented lap simulation

In Section 3.4, we saw a direct description of the lap time optimisation problem on a time oriented domain. Using higher order quadrature methods with this description can be challenging, due to the occurrence of collocation nodes within the quadrature domain. We have outlined these difficulties in Section 3.2. In this section, we build upon Problem 3.4.6 by transforming it onto a track aligned domain. This transformation will show that we can reduce the overall problem dimension by removing the vehicle position variables entirely and relying only on a position description in terms of the track distance s and orthogonal displacement distance ν.

We assume that an admissible control function $\mathbf{u} : [0, T] \to \mathbb{R}^m$, which corresponds to the state solution $\mathbf{x} : [0, T] \to \mathbb{R}^n$ with $\mathbf{x}'(t) = f(\mathbf{x}(t), \mathbf{u}(t))$, exists. Recalling the dynamical system from Equation (2.3.22), we see that the seventh component of $\mathbf{x}$ corresponds to the track distance s. We further assume that our solution $\mathbf{x}$ satisfies

$$\mathbf{x}_7'(t) = s'(t) = \frac{v_x(t)\cos\left(\xi(t)\right) v_y \sin\left(\xi(t)\right)}{1 - \nu(t)K(s(t))} > 0 \quad \forall t \in (0, T). \tag{3.4.8}$$

From Equation (3.4.8) we find a unique arrival time $\tau(s(t)) = t \quad \forall t \in (0, T)$, so that we may define the transformed variable,

$$\tilde{\mathbf{x}} : [0, L] \to \mathbb{R}^8, \qquad \tilde{\mathbf{x}}(s) = \tilde{\mathbf{x}}\left(\tau(s)\right). \tag{3.4.9}$$

In addition to the transformed states,

$$\tilde{\mathbf{x}}(s) = (\tilde{x}(s), \tilde{y}(s), \tilde{\psi}(s), \tilde{v}_x(s), \tilde{v}_y(s), \tilde{\omega}(s), \tilde{\nu}(s), \tilde{\xi}(s))^T$$

we also introduce the control variables

$$\tilde{\mathbf{u}}(s) = (\tilde{F}_{r,\text{throttle}}(s), \tilde{\delta}(s))^T.$$

With these transformed variables we enforce

$$\tilde{\mathbf{x}}'(s) = \mathbf{x}'(\tau(s))\tau'(s) = \frac{\mathbf{f}(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s))}{s'(\tau(s))}. \tag{3.4.10}$$

We can now apply an identical procedure to dynamics equation by approximating the integral

$$\tilde{\mathbf{x}}(s_i) - \tilde{\mathbf{x}}(s_{i-1}) = \int_{s_{i-1}}^{s_i} \frac{f(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s)}{s'(\tau(s))}\, ds \tag{3.4.11}$$

using trapezoidal collocation at the nodes $0 = s_0 < s_1 < \dots s_N = L$. As we are now integrating over the s-domain, we can reduce the state dimension from nine to eight. To further reduce the problem dimension, we notice that we can remove redundant position information contained in $\mathbf{x}$. The vehicle position occurs in $\mathbf{x}$ in an absolute manner in the form of $(\tilde{x}(s), \tilde{y}(s))$ and relative to the track in form of s and $\tilde{\nu}(s)$. We thus remove the $(\tilde{x}(s), \tilde{y}(s))$ variables from our system and obtain the reduced 6 dimensional dynamics

$$\tilde{\mathbf{x}}'(s) = \frac{1}{s'(\tau(s))}\mathbf{f}(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s)) = \tilde{\mathbf{f}}(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s)) \tag{3.4.12}$$

$$= \frac{1 - \tilde{\nu}K(s)}{\tilde{v}_x \cos(\tilde{\xi}) - \tilde{v}_y \sin(\tilde{\xi})} \begin{pmatrix} \tilde{\omega}(s) \\ \frac{1}{m}\left(F_{r,x} - \sin(\tilde{\delta})F_{f,y}\right) + \tilde{\omega}(s)\tilde{v}_y \\ \frac{1}{m}\left(\cos(\tilde{\delta})F_{f,y} + F_{r,y}\right) - \tilde{\omega}(s)\tilde{v}_x \\ \frac{1}{I_z z}\left[a\cos(\tilde{\delta})F_{f,y}(s) - bF_{r,y}\right] \\ \tilde{v}_x(s)\sin(\tilde{\xi}) + v_y\cos(\tilde{\xi}) \\ \tilde{\omega}(s) - K(s)\frac{\tilde{v}_x\cos(\tilde{\xi}) - \tilde{v}_y\sin(\tilde{\xi})}{1-\tilde{\nu}K(s)} \end{pmatrix}. \tag{3.4.13}$$

As discussed in Section 3.3, the switch onto the s-domain allows us to benefit from higher order collocation methods. We now provide explicit formulations for the optimisation problems on the s-domain. The primary difference caused by the change of

optimisation domain is that we no longer have access to the time variables $t_0, \ldots, t_N$. Alternatively, we compute the difference in arrival times τ at two consecutive nodes s_{i-1}, s_i. The state information at these consecutive node positions is $\tilde{\mathbf{x}}_{i-1}, \tilde{\mathbf{x}}_i$ respectively, so that we may compute

$$\begin{aligned}\tau(s_i) - \tau(s_{i-1}) &= \int_{s_{i-1}}^{s_i} \frac{d}{ds}\tau(z)\, dz \\ &= \int_{s_{i-1}}^{s_i} \frac{1}{s'(\tau(s))}\, dz \\ &= \int_{s_{i-1}}^{s_i} \frac{1 - \tilde{\nu}(z)K(z)}{\tilde{v}_x(z)\cos(\tilde{\xi}(z)) - \tilde{v}_y(z)\sin(\tilde{\xi}(z))}\, dz.\end{aligned} \tag{3.4.14}$$

Equation (3.4.14) lets us evaluate the total lap time by approximating the integral in terms of the trapezoidal rule and summing over all interval segments, such that

$$\tau(L) - \tau(0) = \sum_{i=1}^{N} \frac{s_i - s_{i-1}}{2}\left(\frac{d}{ds}\tau(s_i) + \frac{d}{ds}\tau(s_{i-1})\right) + \mathcal{O}\left(\Delta s^2\right). \tag{3.4.15}$$

With objective function (3.4.15), we can write the full s-domain problem for the trapezoidal rule as

$$\begin{aligned}\min_{\tilde{X},\tilde{U}} \quad & \sum_{i=1}^{N} \frac{s_i - s_{i-1}}{2}\left(\frac{d}{ds}\tau(s_i) + \frac{d}{ds}\tau(s_{i-1})\right), \\ \text{subject to: } & \tilde{\mathbf{x}}_i - \tilde{\mathbf{x}}_{i-1} = \frac{1}{2}(s_i - s_{i-1})\left(\tilde{\mathbf{f}}(\tilde{\mathbf{x}}_i, \tilde{\mathbf{u}}_i) + \tilde{\mathbf{f}}(\tilde{\mathbf{x}}_{i-1}, \tilde{\mathbf{u}}_{i-1})\right) \quad \forall i, \\ & \tilde{X}_l \leq \tilde{X} \leq \tilde{X}_u, \\ & \tilde{U}_l \leq \tilde{U} \leq \tilde{U}_u.\end{aligned} \tag{3.4.16}$$

3.5 Interior point methods

In Section 3.4, we discussed how we can construct an non-linear programming problem to solve a minimal lap time problem. In particular, we have transcribed the problem using direct trapezoidal collocation leading us to problem 3.4.6. We will now expand on this by solving problem 3.4.6 using the `trust-constr` algorithm contained in

the `SciPy` library [95] and, discussing performance critical aspects of this algorithm. The `trust-constr` algorithm can handle a wide range of optimisation problems of the form

$$\begin{aligned} \min_{x\in\mathbb{R}^N} \quad & f(x) \\ \text{subject to: } & c_j^l \leq c_j(x) \leq c_j^u \ \ \forall\ j \in \{1,\dots,M\}, \\ & b_i^l \leq x_i \leq b_i^u \ \ \forall\ i \in \{1,\dots,N\}. \end{aligned} \tag{3.5.1}$$

We can enforce one sided inequality constraints in the formulation (3.5.1) by letting $c_j^{u/l} = \pm\infty$ and equality constraints are automatically detected when $c_j^l = c_j^u$. We standardise the inequality constraints in formulation (3.5.2) by using $c_j(x) \leq c_j^u \Leftrightarrow c_j^u - c_j(x) \geq 0$, and including the variable bounds as inequalities, we obtain

$$\begin{aligned} \min_{x\in\mathbb{R}^N} \quad & f(x) \\ \text{subject to: } & c_j(x) = 0 \ \ \forall\ j \in \mathcal{E}, \\ & c_j(x) \geq 0 \ \ \forall\ j \in \mathcal{I}. \end{aligned} \tag{3.5.2}$$

We follow the interior point method construction presented in [65] and reformulate problem (3.5.2) into a barrier problem by including the logarithm of inequality constraints in the objective function. Doing this, we obtain

$$\tilde{f}(x) = f(x) - \mu \sum_{i\in I} \log(c_j(x)), \tag{3.5.3}$$

and ensure that as $c_j(x) \searrow 0$ for some $j \in I$, the objective becomes unbounded $\tilde{f} \nearrow \infty$. In practice using the constraints directly, as done in Equation (3.5.3) may prove problematic, as we do not necessarily have a feasible initial point. As an

alternative, we can introduce the slack variables s and solve

$$\begin{aligned} \min_{x \in \mathbb{R}^N} \quad & f(x) - \mu \sum_{i \in \mathcal{I}} \log(s_i) \\ \text{subject to: } & c_{\mathcal{E}}(x) = 0, \\ & c_{\mathcal{I}}(x) - s = 0. \end{aligned} \tag{3.5.4}$$

Problem (3.5.4) has the Lagrangian function

$$\mathcal{L}(x, s, y, z) = f(x) - \mu \sum_{i \in \mathcal{I}} \log(s_i) - y^T c_{\mathcal{E}}(x) - z^T \left(c_{\mathcal{I}}(x) - s \right), \tag{3.5.5}$$

which lets us construct the KKT conditions

$$\nabla_x f(x) - \nabla_x c_{\mathcal{E}}(x)^T y - \nabla_x c_{\mathcal{I}}^T z = 0, \tag{3.5.6}$$

$$-\mu \operatorname{diag}(s)^{-1} \mathbb{1} + z = 0, \tag{3.5.7}$$

$$c_{\mathcal{E}}(x) = 0, \tag{3.5.8}$$

$$c_{\mathcal{I}}(x) - s = 0. \tag{3.5.9}$$

As the entries of the diagonal matrix $\operatorname{diag} s$ are all positive, we can avoid the non-linearities in Equation (3.5.7) by multiplying with $\operatorname{diag} s$. This yields the equivalent condition

$$-y + \operatorname{diag}(s) z = 0. \tag{3.5.10}$$

To solve this, we apply Newtons method to (3.5.6, 3.5.10, 3.5.8, 3.5.9), and obtain the linear system

$$\begin{bmatrix} \nabla_x^2 \mathcal{L} & 0 & -\nabla_x c_E(x)^T & -\nabla_x c_{\mathcal{I}}(x)^T \\ 0 & \operatorname{diag}(z) & 0 & \operatorname{diag}(s) \\ \nabla_x c_{\mathcal{E}}(x) & & 0 & 0 \\ \nabla_x c_{\mathcal{I}}(x) & I & 0 & 0 \end{bmatrix} \begin{pmatrix} p_x \\ p_s \\ p_y \\ p_z \end{pmatrix} = - \begin{pmatrix} \nabla_x \mathcal{L} \\ Sz - \mu \mathbb{1} \\ c_{\mathcal{E}}(x) \\ c_{\mathcal{I}}(x) - s \end{pmatrix}. \quad (3.5.11)$$

We bring system (3.5.11) into a symmetric form my multiplying the second block-row by $\operatorname{diag}(s)^{-1}$ and swapping the signs of p_y, p_z. This results in the symmetric system

$$\begin{bmatrix} \nabla_x^2 \mathcal{L} & 0 & \nabla_x c_E(x)^T & \nabla_x c_{\mathcal{I}}(x)^T \\ 0 & \Sigma & 0 & I \\ \nabla_x c_{\mathcal{E}}(x) & & 0 & 0 \\ \nabla_x c_{\mathcal{I}}(x) & I & 0 & 0 \end{bmatrix} \begin{pmatrix} p_x \\ p_s \\ -p_y \\ -p_z \end{pmatrix} = - \begin{pmatrix} \nabla_x \mathcal{L} \\ z - \mu \operatorname{diag} s^{-1} \mathbb{1} \\ c_{\mathcal{E}}(x) \\ c_{\mathcal{I}}(x) - s \end{pmatrix}, \quad (3.5.12)$$

where

$$\Sigma = \operatorname{diag}(s)^{-1} \operatorname{diag}(z). \quad (3.5.13)$$

Such symmetric systems can be solved using highly efficient sparse symmetric solvers, such as `MUMPS`, see [2, 1]. The `IPOpt` solver used for this work uses `MUMPS` solver version 5.4.1.

Practical interior point methods often fall into two general categories. The first class of algorithms is based on solving the primal dual system (3.5.12) directly, while adding safeguards in the form of line search procedures to prevent variables from approaching their bounds too quickly and slowing down the following optimisation steps and methods for dealing with rank deficiencies in the Jacobian $\nabla_x c_{\mathcal{E}}(x)$. Other methods, in the form of trust-region interior point methods build a local quadratic

model to the barrier function (3.5.3) and iteratively minimise this local model.
We will first discuss aspects of solving the primal dual system (3.5.12) directly. In this system, we may eliminate the variables p_s using the second block row, leading to the reduced symmetric system

$$\begin{bmatrix} \nabla_x^2 \mathcal{L} & \nabla_x c_E(x)^T & \nabla_x c_{\mathcal{I}}(x)^T \\ \nabla_x c_{\mathcal{E}}(x) & 0 & 0 \\ \nabla_x c_{\mathcal{I}}(x) & 0 & -\Sigma \end{bmatrix} \begin{pmatrix} p_x \\ -p_y \\ -p_z \end{pmatrix} = - \begin{pmatrix} \nabla_x \mathcal{L} \\ c_{\mathcal{E}}(x) \\ c_{\mathcal{I}}(x) - \mu \operatorname{diag} z^{-1} \mathbb{1} \end{pmatrix}. \tag{3.5.14}$$

We can further reduce system (3.5.14) by eliminating p_z using the third block row. After the elimination of p_z we obtain the reduced system matrix

$$\begin{bmatrix} \nabla_x^2 \mathcal{L} + \nabla_x c_{\mathcal{I}}(x)^T \Sigma \nabla_x c_{\mathcal{I}}(x) & \nabla_x c_{\mathcal{E}}(x)^T \\ \nabla_x c_{\mathcal{E}}(x) & 0 \end{bmatrix}. \tag{3.5.15}$$

For small $\mu \searrow 0$, the scaling (3.5.13) will introduce ill-conditioning in system from Equation (3.5.15), as some of the elements of Σ diverge to ∞. When using iterative linear solvers this ill-conditioning may become a grave concern. While $\nabla_x c_{\mathcal{I}}(x)^T \Sigma \nabla_x c_{\mathcal{I}}(x)$ may introduce significant fill-in in the upper left block of our system matrix, we will generally be in the special case when all inequality constraints are a result of variable bounds and thus $\nabla_x c_{\mathcal{I}}(x)^T \Sigma \nabla_x c_{\mathcal{I}}(x)$ will be diagonal. We may now use system (3.5.15) to design a line search interior-point algorithm to solve problem (3.5.2). To compute the step directions, p_x, p_s, p_y, p_z we solve system (3.5.14). To satisfy the slack bounds, we need to limit the step

$$s + \alpha_s p_s \geq 0, \qquad z + \alpha_z p_z \geq 0.$$

We avoid reducing the slack variables too quickly by ensuring

$$s + \alpha_s p_s \geq (1 - \tau)s, \qquad z + \alpha_z p_z \geq (1 - \tau)z, \tag{3.5.16}$$

for some tolerance parameter $\tau \in (0, 1)$ close to 1. A popular choice for τ is $\tau = 0.995$. This lets us compute both the step directions, as well as the step lengths. To define a convergence criterion, we again follow [65, p. 566] by using the KKT conditions from Equations (3.5.6), (3.5.6), (3.5.8), and 3.5.9 to define

$$\begin{aligned} E(x, s, y, z; \mu) := \max\{&\|\nabla f(x) - c_{\mathcal{E}}(x)^T y - c_{\mathcal{I}}(x)^T z\|, \\ &\|z - \mu \operatorname{diag}(s)^{-1} \mathbb{1}\|, \\ &\|c_{\mathcal{E}}(x)\|, \\ &\|c_{\mathcal{I}}(x) - s\|\}. \end{aligned}$$

We present a pseudocode description of the resulting line-search method in Algorithm 2. Algorithm 2 forms the basis of the popular non-linear optimisation li-

Algorithm 2 Line search interior-point

Require: $\mu, \epsilon_{\text{TOL}}, \epsilon_\mu > 0; \tau, \sigma \in (0, 1); x_0, s_0, y_0, z_0$

Standardise bounds and constraints to obtain problem structure (3.5.4).
while $E(x_k, s_k, y_k, z_k) > \epsilon_{\text{TOL}}$ **do**
    while $E(x_k, s_k, y_k, z_k; \mu) > \epsilon_\mu$ **do**
        Compute primal-dual step (p_x, p_y, p_z) by solving system (3.5.14).
        Compute slack step direction $p_s \leftarrow \Sigma^{-1}(z - \mu \operatorname{diag} s^{-1}\mathbb{1} + p_z)$.
        Compute slack step length $\alpha_s^{\max} \leftarrow \max(\{\alpha \in (0, 1] : s + \alpha p_s \geq (1 - \tau)s\}$.
        Compute z step length $\alpha_z^{\max} \leftarrow \max(\{\alpha \in (0, 1] : z + \alpha p_z \geq (1 - \tau)z\}$.
        Update iterate $x_{k+1} \leftarrow x + \alpha_s^{\max} p_x$
        $s_{k+1} \leftarrow s_k + \alpha_s^{\max} p_s$
        $y_{k+1} \leftarrow y_k + \alpha_z^{\max} p_y$
        $z_{k+1} \leftarrow z_k + \alpha_s^{\max} p_z$
        Set $k \leftarrow k + 1$
    end while
    $\mu \leftarrow \mu\sigma$.
end while
return x_k, s_k, y_k, z_k.

brary `IPOpt`(See [98]). While the constrained trust-region interior point method `trust-constr` also works similarly to Algorithm 2, it includes a trust-region to promote convergence. We enforce the trust region using a two step process in which natural equality constraints, as well as slack equality constraints are satisfied up to some residual values $r_{\mathcal{E}}$, $r_{\mathcal{I}}$ respectively. The problem we strive to minimise is

$$\begin{aligned}
\min_{x\in\mathbb{R}^N} \quad & \nabla f(x)^T p_x + \frac{1}{2} p_x \nabla_x^2 \mathcal{L} p_x - \mu \sum_{i\in\mathcal{I}} \frac{p_{s,i}}{s_i} + \frac{1}{2} p_s \Sigma p_s, \\
\text{subject to: } & \nabla_x c_{\mathcal{E}}(x) p_x + c_{\mathcal{E}}(x) = r_{\mathcal{E}}, \\
& \nabla_x c_{\mathcal{I}}(x) p_x - p_s + (c_{\mathcal{I}}(x) - s) = r_{\mathcal{I}}, \\
& \|(p_x, \operatorname{diag}(s)^{-1} p_s)\|_2 \leq \Delta, \\
& \operatorname{diag}(s)^{-1} p_s \geq -\tau \mathbb{1}.
\end{aligned} \tag{3.5.17}$$

We select the constraint residual using the auxiliary problem

$$\begin{aligned}
\min_{\nu_x,\nu_s} \quad & \|\nabla_x c_{\mathcal{E}}(x)\nu_x + c_{\mathcal{E}}(x)\|^2 + \|\nabla_x c_{\mathcal{I}}(x)\nu_x - \operatorname{diag}(s)\nu_s + (c_{\mathcal{I}}(x) - s)\|^2, \\
\text{subject to: } & \|(\nabla_x, \nabla_s)\| \leq 0.8\Delta, \\
& \nu_{s,i} \geq -\frac{\tau}{2} \ \ \forall\ i.
\end{aligned} \tag{3.5.18}$$

With problem (3.5.18), we determine near optimal values

$$r_{\mathcal{E}} = \nabla_x c_{\mathcal{E}}(x)\nu_x + c_{\mathcal{E}}, \tag{3.5.19}$$

$$r_{\mathcal{I}} = \nabla_x c_{\mathcal{E}}(x)\nu_x - \operatorname{diag}(s)\nu_s + (c_{\mathcal{I}}(x) - s)\,, \tag{3.5.20}$$

via minimisation over the smaller trust-region radius 0.8Δ. We then solve the original equality constrained quadratic program (3.5.17) by using a projected CG method. As is standard in trust-region methods, we use predicted and achieved reductions to

decide whether to accept or reject a step. With the merit function

$$\Phi(x, s, \nu) = f(x) - \mu \sum_{i \in \mathcal{I}} \log(s_i) + \nu \|c_{\mathcal{E}}(x)\| + \nu \|c_I(x) - s\|,$$

we define the achieved reduction

$$\text{ared}(p) = \Phi(x, s, \nu) - \Phi(x + p_x, s + p_s, \nu).$$

The predicted reduction, according to our quadratic model, is

$$\text{pred}(p) = f(x) - \left(\nabla f(x)^T p_x + \frac{1}{2} p_x \nabla_x^2 \mathcal{L} p_x - \mu \sum_{i \in \mathcal{I}} \frac{p_{s,i}}{s_i} + \frac{1}{2} p_s \Sigma p_s \right).$$

We then accept a step if

$$\text{ared}(p) \geq 10^{-8} \text{ pred}(p). \tag{3.5.21}$$

The acceptance parameter 10^{-8} is based on [65] and coincides with the equivalent parameter choice in the `trust-constr` algorithm. Finally, we can see pseudocode representation of `trust-constr` in Algorithm 3[1]. We now examine how well the `trust-constr` performs on the minimal lap time problem with trapezoidal collocation.

[1]We note that `trust-constr`, as implemented in the current release of `SciPy` (1.8.0) contains the possibility to accept a step if Equation (3.5.21) is satisfied after a second order correction step, following [65, p.433].

Algorithm 3 Trust-constr

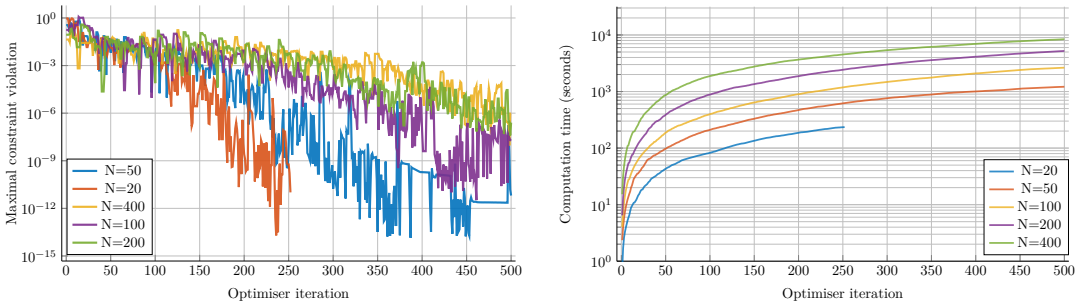
Require: $\mu, \epsilon_{\text{TOL}}, \epsilon_\mu > 0; \tau, \sigma \in (0,1); x_0, s_0, y_0, z_0$

Standardise bounds and constraints to obtain problem structure (3.5.4).
while $E(x_k, s_k, y_k, z_k) > \epsilon_{\text{TOL}}$ **do**
    while $E(x_k, s_k, y_k, z_k; \mu) > \epsilon_\mu$ **do**
        Compute (ν_x, ν_s) by solving problem (3.5.18).
        Compute (p_x, p_s, p_y, p_z) by solving (3.5.17) using projected CG.
        Compute the predicted and achieved achieved value reductions $\text{pred}(p), \text{ared}(p)$
        if $\text{ared}(p) \geq 10^{-8}\text{pred}(p)$ **then**
            Accept step $(x_{k+1}, s_{k+1}, y_{k+1}, z_{k+1}) \leftarrow (x_k, s_k, y_k, z_k) + (p_x, p_s, p_y, p_z)$
        else
            Reject step $(x_{k+1}, s_{k+1}, y_{k+1}, z_{k+1}) \leftarrow (x_k, s_k, y_k, z_k)$
        end if
        $k \leftarrow k + 1$.
    end while
    $\mu \leftarrow \mu\sigma$.
end while
return x_k, s_k, y_k, z_k.

3.6 Optimisation results

In Section 3.4, we have seen methods for formulating the minimal lap time collocation problem. In this section, we proceed to solve the resulting problem (3.4.16) using the `trust-constr` algorithm. As an initial example, we consider the "L"-shaped test track, which we can see in Figure 3.3 and Algorithm 1 to generate an initial trajectory. Figures 3.4a and 3.4b show the reduction constraint violation, measured in terms of the $\|\cdot\|_\infty$ norm, and the required computation time for the first 500 iterations. In addition to this, we can see the reduction in required lap time in Figure 3.5. Figure 3.4b shows that for the case of $N = 20$ nodes the simulation was stopped early after 252 iterations. This early stopping occurs due to a limit condition on the trust-region size. By default, the algorithm stops once the trust-region radius $\Delta < 10^{-8}$.

As discussed in [65, p. 581], we need to solve the linear system with the projection



(a) Constraint violation (b) Computation time

Figure 3.4: Objective value, constraint violation and computation time of `trust-constr` applied to the lap time minimisation problem (3.4.16)

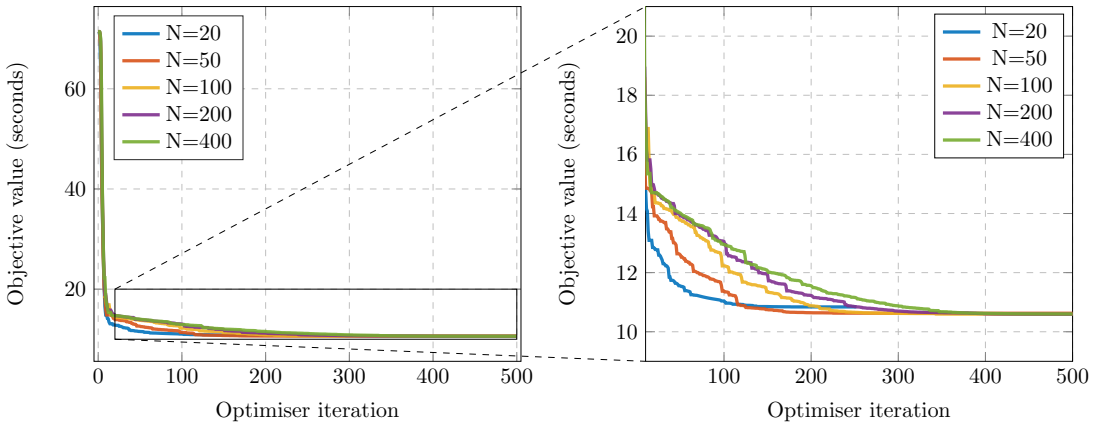


Figure 3.5: Lap time reduction during optimisation with `trust-constr`

matrix

$$\begin{bmatrix} I & 0 & \nabla c_{\mathcal{E}}(x)^T & \nabla c_{\mathcal{I}}(x)^T \\ 0 & I & 0 & -\operatorname{diag}(s) \\ \nabla c_{\mathcal{E}}(x) & 0 & 0 & 0 \\ \nabla c_{\mathcal{I}}(x) & -\operatorname{diag}(s) & 0 & 0 \end{bmatrix}$$

in each iteration to perform a step of the projected CG method. All the inequality constraints in our problem result from variable bounds, thus the inequality Jacobian $\nabla c_{\mathcal{I}}(x)$ is readily available. Additionally, we may use the special structure of our optimal control problem (3.4.6) to get low cost approximations to the equality constraint Jacobian $\nabla c_{\mathcal{E}}(x)$. We proceed to discuss how the projected conjugate gradient algorithm is executed and how we may use gradient approximations to achieve a performance increase. Recalling the quadratic problem (3.5.17), we introduce the notation

$$H = \begin{bmatrix} \nabla_x^2 \mathcal{L} & 0 \\ 0 & \Sigma \end{bmatrix}, \quad c = \nabla_x f(x) - \mu s^{-1}, \quad A = \begin{bmatrix} \nabla_x c_{\mathcal{E}}(x) & 0 \\ \nabla_x c_{\mathcal{I}}(x) & -\operatorname{diag}(s) \end{bmatrix},$$

where s^{-1} is the, component wise, multiplicative inverse of s. During Algorithm 3, we compute a slack variable update ν_x, ν_s, by solving problem (3.5.18). We thus know that we can find p_x, p_s within the reduced trust region 0.8Δ which reduces the constraint violations from $r_{\mathcal{E}}, r_{\mathcal{I}}$ to

$$\tilde{r}_{\mathcal{E}} = c_{\mathcal{E}} + \nabla_x c_{\mathcal{E}} \nu_x, \tag{3.6.1}$$

$$\tilde{r}_{\mathcal{I}} = c_{\mathcal{I}} + \nabla_x c_{\mathcal{I}} \nu_s - s - \operatorname{diag}(s)\nu_s. \tag{3.6.2}$$

We consequently solve the equality constrained quadratic problem

$$\begin{aligned} \min_{p_x,p_s\in\mathbb{R}^n} \quad & \frac{1}{2}\begin{pmatrix} p_x \\ p_s \end{pmatrix}^T H \begin{pmatrix} p_x \\ p_s \end{pmatrix} + c^T \begin{pmatrix} p_x \\ p_y \end{pmatrix} \\ \text{subject to: } & A\begin{pmatrix} p_x \\ p_y \end{pmatrix} = \begin{pmatrix} \tilde{r}_{\mathcal{E}} \\ \tilde{r}_{\mathcal{I}} \end{pmatrix}, \\ & \|(p_x, p_y)\| \leq \sqrt{\Delta^2 - \|(\nu_x, \nu_y)\|^2}. \end{aligned} \tag{3.6.3}$$

Algorithm 4 shows a pseudocode representation of the projected CG algorithm. We

Algorithm 4 Projected conjugated gradient algorithm

Require: $\Delta > 0, H \succ 0, \epsilon > 0$

procedure PCG($H, c, Z, Y, b, \Delta, x^{\text{lower}}, x^{\text{upper}}, N^{\max}$)
  $x_0 \leftarrow -Yb$
  $r \leftarrow Z\left(Hx_0 + c\right)b$
  $g \leftarrow Zr$
  $p \leftarrow -g$
  $i \leftarrow 0$
  while $i < N^{\max}$ and $\|g\| > \epsilon$ **do**
    $i \leftarrow i + 1$
    $\alpha \leftarrow \frac{\|g\|^2}{p^T H p}$
    $x_i \leftarrow x_{i-1} + \alpha p$
    if $\|x_i\| \geq \Delta$ **then** ▷ Stop criterion: Contacted trust-region boundary
      $\theta_1 \leftarrow \max\left\{\theta | \theta \geq 0, \|x_{i-1} + \theta\alpha p\| \leq \Delta\right\}$
      $\theta_2 \leftarrow \max\left\{\theta | \theta \geq 0, x^{\text{lower}} \leq x_{i+1} + \theta\alpha p \leq x^{\text{upper}}\right\}$
      $x_i = x_{i-1} + \min\left\{\theta_1, \theta_2\right\}\alpha p$
      return x_i
    else
      $\beta \leftarrow \|g\|$
      $r \leftarrow r + \alpha H p$
      $g \leftarrow Zr$
      $p \leftarrow \frac{\|g\|^2}{\beta} p - g$
    end if
  end while
end procedure

see that to apply the projected CG algorithm, we require linear operators, which project a vector onto the null-space of A. Additionally, we need a projection operator

onto the row-space of A. We denote the null-space projection operator by Z and the row-space projection by Y. In practice, we compute the projections $z = Zx, y = Yx$ by solving

$$\begin{pmatrix} I & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} z \\ v \end{pmatrix} = \begin{pmatrix} x \\ 0 \end{pmatrix}, \qquad \begin{pmatrix} I & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} y \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ x \end{pmatrix}. \tag{3.6.4}$$

We now discuss two options to accelerate the `trust-constr` algorithm in the context of problem (3.4.6). The first option is to utilise a Graphics Processing Unit (GPU) to evaluate the dynamics Jacobian $\nabla_x c_{\mathcal{E}}$ faster. The second method we present relies on the specific structure of (3.4.6) obtained by trapezoidal collocation, which allows us to quickly compute gradient approximations.

3.6.1 GPU accelerated gradient evaluation

During each iteration of Algorithm 3, we require the dynamics Jacobian $\nabla_x c_{\mathcal{E}}(x)$, which we evaluate using automatic differentiation. As we are using the PyTorch [73] automatic differentiation engine in our computation, we can directly enable GPU computations. To enable GPU computation, we ensure that the required data is on the correct device, as shown in Listing 3.1.

Listing 3.1: GPU accelerated gradient approximation

```
import torch as th
import functorch
from typing import Callable

def eval_jac(objective: Callable, problem_vec: th.Tensor) -> th.Tensor:
    dev = str(problem_vec.device)
    pvec_th = problem_vec.to('cuda:0') if dev != 'cuda:0' else pvec
    out_jac = functorch.jacrev(objective)(pvec_th)
```

```
    out_jac = out_jac.to(dev) # Return data to original device
    return out_jac
```

As each entry of the Jacobian $\nabla_x c_{\mathcal{E}}(x)$ can be evaluated separately from the others, we can take further advantage of FuncTorch [48], the composable function interface build for PyTorch, to evaluate Jacobian entries in parallel. This benefits the highly parallel architecture found on GPUs.

Figure 3.6 shows the profiling results obtained by executing 20 iterations or Algorithm 3. We obtained the results shown in Figure 3.6 by solving a sequence of collo-

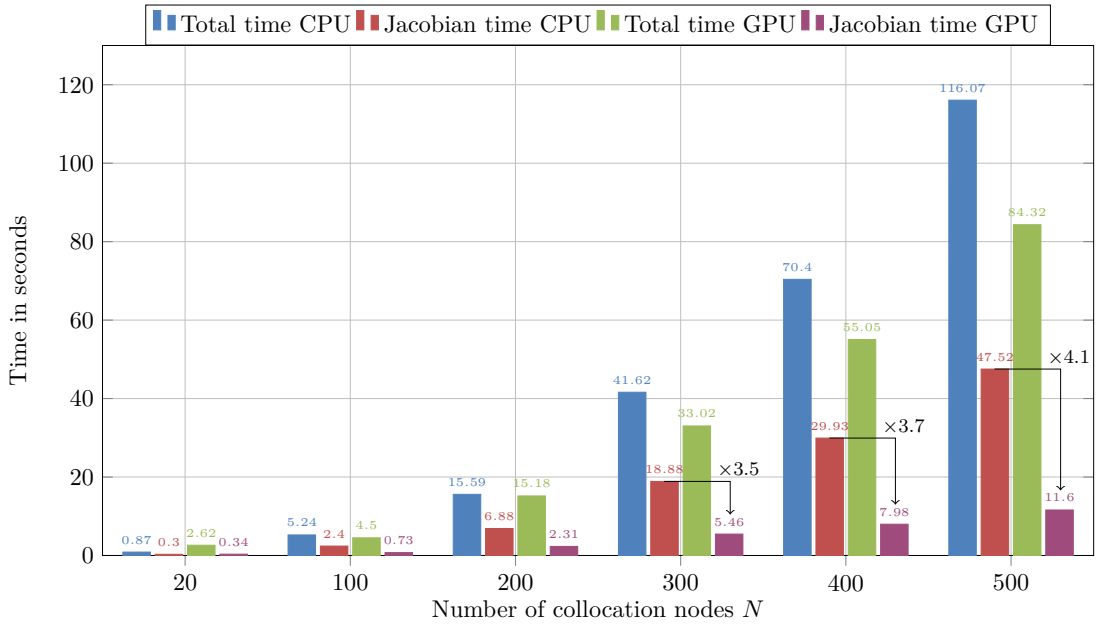


Figure 3.6: Profiling results

cation problems on a system with an i7-10750H CPU and a GTX-1650Ti GPU and measuring timings using the `cProfile` module. We see in Figure 3.6 that a large portion of time in each iteration of Algorithm 3 is spend evaluating the constraint Jacobian. For CPU computation, an average of 42.3% of the computation time is spend on evaluating these Jacobian matrices. We compare this to an average of 14.9% of the computation time in the case of GPU accelerated Jacobian evaluations. This speed-up is represented in the reduction of average total computation time by

23.3%, for problems with $N \geq 200$ collocation nodes nodes. For smaller, systems the additionally overhead incurred by copying data between CPU and GPU memory may result in a performance reduction, as we can see in the case of $N = 20$ nodes.

3.6.2 Gradient approximations

For minimal time optimal control, the objective function coincides with the difference in value of the first and last time variable. During this section, we have benefited from a simplified notation, in which we have collected all optimisation variables present in our non-linear program into a single vector x. This notation hides the complexity of our collocation problem, as the state variable vectors $\mathbf{x}_0, \ldots, \mathbf{x}_{N-1}$ are contained within x. More precisely, the vector x in many problem formulations, such as Equations (3.5.3) and (3.4.6), contains an unrolled version of all the state, control, and time variables. In the case of trapezoidal collocation we can write

$$x = \left(\mathbf{x}_0, \mathbf{u}_0, t_0, \ldots, \mathbf{x}_{N-1}, \mathbf{u}_{N-1}, t_{N-1}\right)^T \in \mathbb{R}^{N\times(n+m+1)}. \tag{3.6.5}$$

The optimal control objective function is thus

$$f(x) = f\left(\mathbf{x}_{\{0,\ldots,N-1\}}, \mathbf{u}_{\{0,\ldots,N-1\}}, t_{\{0,\ldots,N-1\}}\right) = t_{N-1} - t_0. \tag{3.6.6}$$

Using this, we see that as f is linear in the input variables, such that $\nabla_x^2 f(x) = 0$. The equality constraint function, resulting from the quadrature of our system dynamics in Equation (3.2.6), are

$$\nabla_x c_i(x) = \nabla_x \left(\mathbf{x}_i - \mathbf{x}_{i-1} - \frac{t_i - t_{i-1}}{2}\left(\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) + \mathbf{f}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1})\right)\right). \tag{3.6.7}$$

To write the Jacobian matrix of these constraints, we introduce further simplifying notation in the form of the matrices

$$I^{\mathbf{x}_i} \in \mathbb{R}^{L \times n}, \qquad I^{\mathbf{x}_i}_{k,j \in \mathcal{I}(\mathbf{x_i})} = I_n, \tag{3.6.8}$$

$$I^{\mathbf{u}_i} \in \mathbb{R}^{L \times m}, \qquad I^{\mathbf{u}_i}_{k,j \in \mathcal{I}(\mathbf{u_i})} = I_m, \tag{3.6.9}$$

$$I^{t_i} \in \mathbb{R}^{L \times 1}, \qquad I^{\mathbf{u}_i}_k = \begin{cases} 1, & \text{if } k = I(t_i) \\ 0, & \text{otherwise} \end{cases}, \tag{3.6.10}$$

where $\mathcal{I}$ is the function that maps the optimisation variables $\{\mathbf{x}_{0,\dots,N-1}, \mathbf{u}_{0,\dots,N-1}, t_{0,\dots,N-1}\}$ to the set indices in the non-linear programming formulation. As an example, we can see from Equation (3.6.5) that $\mathcal{I}(\mathbf{x}_0) = \{1, \dots, n\}$. With this we write

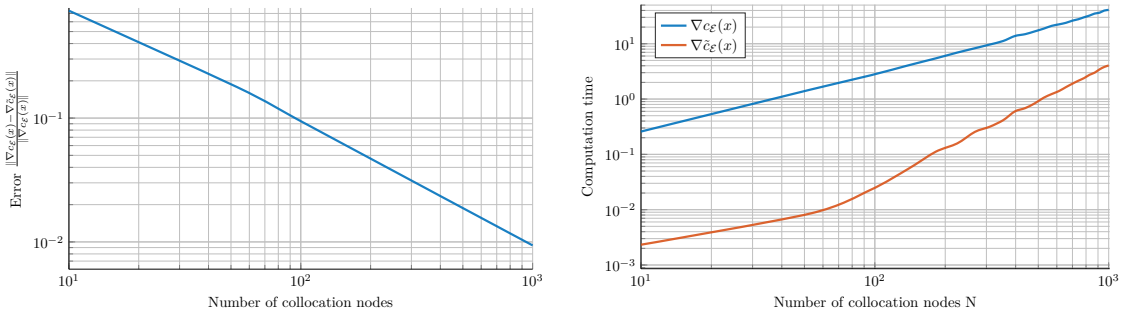
$$\nabla_x c_i(x) = \nabla_x \left(\mathbf{x}_i - \mathbf{x}_{i-1} - \frac{t_i - t_{i-1}}{2} \left(\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) + \mathbf{f}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1}) \right) \right) \tag{3.6.11}$$

$$\begin{aligned} &= I^{\mathbf{x}_i} - I^{\mathbf{x}_{i-1}} - \frac{I^{t_i} - I^{t_{i-1}}}{2} \left(\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) + \mathbf{f}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1}) \right) \\ &\quad - \frac{t_i - t_{i-1}}{2} \left(I^{\mathbf{x}_i} \nabla_x \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) + I^{\mathbf{x}_{i-1}} \nabla_x \mathbf{f}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1}) \right) \end{aligned} \tag{3.6.12}$$

$$= I^{\mathbf{x}_i} - I^{\mathbf{x}_{i-1}} - \frac{I^{t_i} - I^{t_{i-1}}}{2} \left(\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) + \mathbf{f}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1}) \right) + \mathcal{O}(\delta_t). \tag{3.6.13}$$

Without this approximation, we need to evaluate these constraint derivatives either by using a finite difference approximation, or by using automatic differentiation. Automatic differentiation has benefits from a computational standpoint as derivatives are not approximated, but evaluated directly by back-tracing on the performed operations, in particular when evaluating the dynamics function $\mathbf{f}$. Furthermore, automatic differentiation often scales more favourably with the number of variables if Jacobian matrices are sparse. In Figures 3.7a, and 3.7b, we can see the experimental approximation quality of the approximation in Equation (3.6.13) [2]. In Figure 3.7a, we see the expected error reduction of $\mathcal{O}(\delta_t) = (N^{-1})$, from Equation (3.6.13).

[2] All experiments were performed on an Intel i7-10750H processor with no explicit parallelisation.



(a) Equation (3.6.13) approximation quality. (b) Equation (3.6.13) computation time.

Figure 3.7: Quality and required computation time of the approximation in Equation (3.6.13). The exact Jacobian was computed using the PyTorch automatic differentiation toolkit (See [48]).

3.7 Mesh refinement strategies

In the previous sections we reviewed the derivation of dynamics equations for the bicycle model and how we can approach the problem numerically using collocation methods. We now build upon this work by introducing error estimation methods and constructing mesh refinement algorithms. The error estimation strategies presented follow the theory discussed in [9] and [29].

In the following, we discuss the solution of optimal control problems over the entire time domain. Similar to problem (3.4.6) from Section 3, we denote the state and control components by using problem matrices

$$X = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix}, \quad U = \begin{bmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_N \end{bmatrix}, \quad T = \begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_N \end{bmatrix}. \tag{3.7.1}$$

In Section 2.3.1, we discussed the formulation of the minimal time optimal control problem using direct trapezoidal collocation, yielding a problem of the form

$$\begin{aligned}
\min_{X,U,T} \quad & t_N - t_0, \\
\text{subject to} \quad & \mathbf{x}_i - \mathbf{x}_{i-1} - \frac{t_i - t_{i-1}}{2}\left(\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) + \mathbf{f}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1})\right) \ \forall i, \\
& X^l \leq X \leq X^u, \\
& U^l \leq U \leq U^u, \\
& t_i \leq t_{i+1} \ \ \forall i = 0, \ldots, N-1.
\end{aligned} \tag{3.7.2}$$

The underlying domain, in the form of a race-track, is discretised with a collection track segments $\Omega_1, \ldots, \Omega_n$, consisting of two or more track-nodes, depending on the underlying polynomial approximation. By solving the non-linear program, we obtain information about the state and control variables at each track node. Using this information we now aim to build interpolating functions $\tilde{\mathbf{x}}(\cdot)$ and $\tilde{\mathbf{u}}(\cdot)$ to the true, but unknown, state and control functions. At each track-node, we impose the state approximating function

$$\tilde{\mathbf{x}}(t_i) = \mathbf{x}_i \qquad \partial_t \tilde{\mathbf{x}}(t_i) = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i). \tag{3.7.3}$$

Using this, we build a cubic spline representation of the true state variable $\mathbf{x}(\cdot)$. For the control variables, we can impose

$$\tilde{\mathbf{u}}(t_i) = \mathbf{u}_i \tag{3.7.4}$$

to construct a linear interpolating spline. In the following, we assume that the error in the control solution is dominated by the control solution error, to the degree that we may use our control solution as a ground truth for computing errors in the discretisation.

3.7.1 Error estimation

Using the solution representations obtained in Section 3.7, we now follow [9], to derive local error estimates for our approximation. In the classical theory for numerical solution of ordinary differential equations, as described in [62, 52], the local or truncation error is the local error committed by the numerical method in one time step, when applied to the true solution. Given an ordinary differential equation (ODE) of the form

$$\mathbf{x}'(t) = g(\mathbf{x}(t); t), \qquad \mathbf{x}(0) = \mathbf{x}_0$$

a linear multi-step method (LMM) will allow us to simulate the development of such an ODE with

$$\mathbf{x}_n = A(\Delta t) \left(\mathbf{x}_n, \mathbf{x}_{n-1}, \ldots, \mathbf{x}_{n-k}\right)^T. \tag{3.7.5}$$

The truncation error for such a multi-step method is then given by

$$\eta = \frac{\mathbf{x}(t_n) - \mathbf{x}(t_{n-1})}{t_n - t_{n-1}} - A(\Delta t) \left(\mathbf{x}_n, \mathbf{x}_{n-1}, \ldots, \mathbf{x}_{n-k}\right)^T. \tag{3.7.6}$$

By the underlying dynamics equation for an optimal control problem it holds that

$$\mathbf{x}(t_k) = \mathbf{x}(t_{k-1}) + \int_{t_{k-1}}^{t_k} \mathbf{f}(\mathbf{x}(s), \mathbf{u}(s))\, ds. \tag{3.7.7}$$

Instead of computing the integral in (3.7.7) exactly, we employ a numerical quadrature rule in our non-linear program. This numerical quadrature method uses, at most, all the points associated with the track segment Ω_i. Let I_i be the index set of all those track-nodes associated with segment Ω_i. With this information we have discretized

the integral in the form

$$\int_{t_{k-1}}^{t_k} \mathbf{f}(\mathbf{x}(s), \mathbf{u}(s))\, ds \approx |t_k - t_{k-1}| F(\{\mathbf{x}_i : i \in I\}). \tag{3.7.8}$$

Equation (3.7.8) lets us write the truncation error of this method as

$$\eta = \frac{\mathbf{x}(t_k) - \mathbf{x}(t_{k-1})}{t_k - t_{k-1}} - F(\{\mathbf{x}(t_i) : i \in I\}). \tag{3.7.9}$$

We now proceed by modifying equation (3.7.7), using the interpolated state and control functions to derive

$$\begin{aligned} \mathbf{x}(t_k) &= \mathbf{x}(t_{k-1}) + \int_{t_{k-1}}^{t_k} f(\mathbf{x}(s), \mathbf{u}(s))\, ds & (3.7.10)\\ &\approx \mathbf{x}(t_{k-1}) + \int_{t_{k-1}}^{t_k} f(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s))\, ds. & (3.7.11) \end{aligned}$$

A local error estimate can then be computed by assuming that the starting point $\mathbf{x}_{k-1}$ of an interval is accurate. We define the best estimate $\hat{\mathbf{x}}_k$ to be

$$\hat{\mathbf{x}}_k = \mathbf{x}_{k-1} + \int_{t_{k-1}}^{t_k} f(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s))\, ds, \tag{3.7.12}$$

so that we can derive the local error estimate

$$\begin{aligned} |\mathbf{x}_k - \hat{\mathbf{x}}_k| &= \left| \mathbf{x}_k - \mathbf{x}_{k-1} - \int_{t_{k-1}}^{t_k} \mathbf{f}(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s))\, ds \right| & (3.7.13)\\ &= \left| \mathbf{x}_{k-1} - \mathbf{x}_{k-1} - \int_{t_{k-1}}^{t_k} \tilde{\mathbf{x}}'(s) - \mathbf{f}(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s))\, ds \right| & (3.7.14)\\ &\leq \int_{t_{k-1}}^{t_k} \left| \tilde{\mathbf{x}}'(s) - \mathbf{f}(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s))\, ds \right|. & (3.7.15) \end{aligned}$$

3.7.2 Grid refinement

In the preceding Sections 3.7 and 3.7.1, we discussed how one may use the solution of a non-linear program to derive an interpolating function and determine error estimates. With the goal of constructing a locally optimal trajectory for a racing car, we want to use the derived error estimates to inform our mesh refinement. Over the past two decades, we have seen direct collocation methods, such as those used in this report, gain popularity. Most mesh refinement methods employ so called h-methods, where a fixed low-degree polynomial solution is refined by increasing the number of grid points. Convergence of h-methods is achieved as the local step length

$$h_k := t_k - t_{k-1} \tag{3.7.16}$$

approaches zero. An example for such a method would be to use trapezoidal collocation on an initial grid and, as long as a predefined error estimate is exceeded in a given interval, introduce new grid points to reduce the local error. Examples for h-methods can be found in [10, 50, 9, 99]. In contrast to h-methods, we may also improve the solution accuracy of methods by increasing the degree of the underlying polynomial, rather than introducing new grid points and maintaining the polynomial degree. These methods are known as pseudo-spectral or p-methods. In pseudo-spectral methods, we often represent the functions using Legendre or Chebyshev polynomials. For problems with smooth state and control solutions pseudo-spectral methods result in spectral convergence. Treating the entire problem domain as a single segment generally introduces limitations, resulting in slow convergence, if the solution is not well-behaved. An additional computational constraint, resulting from p-methods, is that the Jacobian matrix of the nonlinear program will generally be dense. To make p-methods more practical, they are often used in conjunction with h-methods, so that in intervals, where a smooth solution is expected a high degree polynomial is used, and

singularities are resolved by low order h-methods with a high number of collocation nodes. Examples for p-methods are given in [33, 34, 7], while [5, 31] present examples for hp-methods. While hp-methods are able to combine the benefits of both h- and p-methods, they are much more difficult to implement and thus used less commonly than p-methods or h-methods.

3.7.3 h-refinement

Following the theory of [9], we present a mesh refinement procedure of h-type. The authors of [9, 10] note that numerical methods, when applied to constraint differential equations, often experience an order reduction. We say that a method for integrating an ODE is of order p if the local truncation error η_k behaves like

$$\eta_k = \frac{x_k - x_{k-1}}{t_k - t_{k-1}} - F(\{x_i : i \in I\}) \tag{3.7.17}$$

$$= \mathcal{O}((t_k - t_{k-1})^{p+1}). \tag{3.7.18}$$

It is common to see an order reduction of the form

$$\eta_k = \mathcal{O}((t_k - t_{k-1})^{p+1-r_k}). \tag{3.7.19}$$

In [9], the author presents an *a-posteriori* method for detecting this order reduction, based on two grid levels and shows how to construct a new mesh based on these estimates. We now use the numerical solutions derived on two mesh levels l_0 and l_1, where l_1 was obtained by subdividing the l_0 mesh, so that all nodes ω_j^0 of mesh l_0 are also present in the l_1 mesh. In Figure 3.8, we see a comparison between the different mesh levels. Mesh level l_1 was constructed by adding $I = 2$ additional nodes, thus cutting the interval length by a factor of three. We obtain an estimate for the order

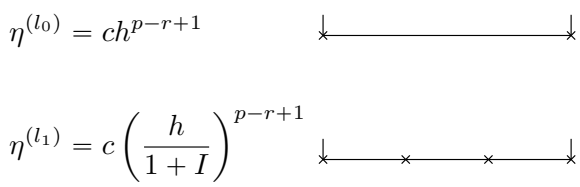


Figure 3.8: Comparison of mesh levels.

reduction r by

$$\hat{r} = p + 1 - \frac{\log(\eta^{(l_0)}/\eta^{(l_1)})}{\log(1+I)}. \tag{3.7.20}$$

Our goal is then to achieve the optimal error reduction, while adding a finite number $M \in \mathbb{N}_{\geq 1}$ nodes to the mesh. After we have derived an order reduction $\hat{r}_k$ and error $\hat{\eta}_k$ for each segment of the coarse grid, we use a greedy algorithm, that adds nodes where the error reduction effect is maximal. On each segment Ω_k, we estimate the change of error by

$$\partial_{I_k}\hat{\eta}_k(I_k) = \partial_{I_k}(ch_k^{p-\hat{r}_k+1}(1+I_k)^{-p-1+\hat{r}_k}) \tag{3.7.21}$$

$$= (-p-1+\hat{r}_k)ch_k^{p-\hat{r}_k+1}(1+I_k)^{-p-2+\hat{r}_k}. \tag{3.7.22}$$

3.7.4 *hp*-refinement

Many grid refinement methods are in practice implemented as *h*-methods, due to their relative simplicity. Over the past two decades however, we have seen a rise in popularity of pseudo-spectral or *p*-methods [29]. The limitation of *p*-methods arise, when they are applied to optimal control problems, which have either non-smooth solutions, or non-smooth problem formulations. Furthermore, we encounter numerical difficulties due to density of Jacobian matrices in pseudo-spectral methods. In cases where they can be applied, *p*-methods are very attractive due to their exponential rate of convergence. In order to increase their utility, while attempting to maintain

as close to exponential convergence as possible, *hp*-methods have been developed. While they are commonly applied in finite element methods for partial differential equations, they are not yet commonly applied in optimal control type problems.

The main difficulty in *hp*-type mesh refinement is to build a heuristic algorithm to decide between increasing the number of track segments in an *h*-method type procedure or increasing the polynomial degree on a track segment, similar to *p*-methods. In the following, we present such an algorithm, based on optimal L^1 polynomial approximations. We consider again a mesh with track-nodes

$$\omega_0, \ldots, \omega_N, \tag{3.7.23}$$

and track segments $\Omega_1, \ldots, \Omega_n$. For each node ω_k we let t_k be the linear programming variable, corresponding to when the vehicle crosses the k-th node. In Figure 3.9 we can see such a division of a track into three different segments of different polynomial degree, where Ω_1 is associated with three nodes $\omega_1, \omega_2, \omega_3$, as such the state variables on Ω_1 are approximated by quadratic polynomials. Similarly, the approximation is also quadratic on Ω_2 and linear on Ω_3.

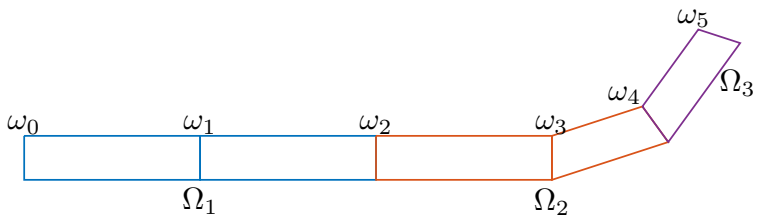


Figure 3.9: A track discretisation with segments of different polynomial degrees

For each track segment Ω_l, we let t_{l-1} denote the time when the vehicle enters Ω_l and t_l be the time when it crosses over into the following segment. By the definition of the t variables, we have that

$$t_{l-1} = \min(\{t_k : \omega_k \in \Omega_l\}), \qquad t_l = \max(\{t_k : \omega_k \in \Omega_l\}). \tag{3.7.24}$$

Similar to Section 3.7.1, we construct spline interpolations $\tilde{\mathbf{x}}, \tilde{\mathbf{u}}$ to the true solution $\mathbf{x}, \mathbf{u}$. These interpolating functions are uniquely determined by the interpolation conditions

$$\tilde{\mathbf{x}}(t_k) = \mathbf{x}_k \qquad \forall k = 0, \ldots, N, \tag{3.7.25}$$

$$\tilde{\mathbf{x}}'(t_k) = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) \quad \forall k = 0, \ldots, N, \tag{3.7.26}$$

$$\tilde{\mathbf{u}}(t_k) = \mathbf{u}_k \qquad \forall k = 0, \ldots, N. \tag{3.7.27}$$

With this, we compute the same local estimate $\hat{\mathbf{x}}$ as in Section 3.7.1 by integrating

$$\hat{\mathbf{x}}(t) = \tilde{\mathbf{x}}(t_{l-1}) + \int_{t_{l-1}}^{t} \mathbf{f}(\tilde{\mathbf{x}}(s), \tilde{\mathbf{u}}(s))\, ds. \tag{3.7.28}$$

We then proceed by approximating $\hat{\mathbf{x}}(\cdot)$ both with a higher degree polynomial and with two lower degree polynomials. Let $p = \|\Omega_l\| - 1$ be the degree of polynomial interpolation used on Ω_l. We will divide the time interval (t_{l-1}, t_l) into the subintervals $(t_{l-1}, t_{l-1/2})$, $(t_{l-1/2}, t_l)$ and interpolate $\hat{\mathbf{x}}$ using polynomials of degree $\tilde{p} = \lceil p/2 \rceil$, such that $2(\tilde{p}+1) \geq p+1$. We compute the approximation errors

$$\epsilon_1 = \min_{q \in \mathbf{P}_{p+1}(t_{l-1}, t_l)} \|\hat{\mathbf{x}} - q\|_{L^1}, \tag{3.7.29}$$

$$\epsilon_2 = \min_{q_1 \in \mathbf{P}_{\tilde{p}}(t_{l-1}, t_{l-1/2})} \|\hat{\mathbf{x}} - q_1\|_{L^1} + \min_{q_2 \in \mathbf{P}_{\tilde{p}}(t_{l-1/2}, t_l)} \|\hat{\mathbf{x}} - q_2\|_{L^1}. \tag{3.7.30}$$

In case that the best estimate $\hat{\mathbf{x}}$ can be approximated better using a higher degree polynomial, rather than two lower degree polynomials, in other words if

$$\epsilon_1 \leq \epsilon_2, \tag{3.7.31}$$

it is natural to increase the degree of polynomial approximation on the underlying track segment Ω_l to $p+1$. If, on the other hand the solution shows discontinuities,

which can no longer be efficiently represented with higher degree polynomials, we find that

$$\epsilon_1 > \epsilon_2, \tag{3.7.32}$$

and choose to divide Ω_l into two segments of polynomial degree $\lceil p/2 \rceil$ in the next mesh refinement step. This new mesh refinement heuristic is given as pseudo-code in Algorithm 5.

Algorithm 5 Mesh refinement

$d_i \leftarrow |\Omega_i|$
$\hat{\mathbf{p}}_i(t) \leftarrow \mathbf{p}_i(t_{i-1}) + \int_{t_{i-1}}^{t} f(\mathbf{p}(t), \mathbf{u}(t))\, dt$
$\tilde{p} \leftarrow \lceil \frac{d_i - 1}{2} \rceil$
$\epsilon_1 = \min_{q \in \mathbf{P}_{p+1}(t_{l-1}, t_l)} \|\hat{\mathbf{x}} - q\|_{L^1}$
$\epsilon_2 = \min_{q_1 \in \mathbf{P}_{\tilde{p}}(t_{l-1}, t_{l-1/2})} \|\hat{\mathbf{x}} - q_1\|_{L^1} + \min_{q_2 \in \mathbf{P}_{\tilde{p}}(t_{l-1/2}, t_l)} \|\hat{\mathbf{x}} - q_2\|_{L^1}$
if $\epsilon_1 < \epsilon_2$ **then**
    $\Omega_i' \leftarrow \text{chebpoints}(t_{i-1}, t_i, d_i + 2)$
    $\Omega_i'' \leftarrow \emptyset$
else
    $t_{i-1/2} \leftarrow \frac{t_{i-1} + t_i}{2}$
    $\Omega_i' \leftarrow \text{chebpoints}(t_{i-1}, t_{i-1/2}, \tilde{p} + 1)$
    $\Omega_i'' \leftarrow \text{chebpoints}(t_{i-1/2}, t_i, \tilde{p} + 1)$
end if
return Ω_i', Ω_i''

3.8 Efficient L^1 approximation

In Section 3.7.4, we introduced a mesh refinement heuristic, which depends on computing optimal L^1 approximating polynomials. Solving the L^1 approximation problem

$$\min_{p \in \mathbf{P}_n} \|f - p\|_{L^1} = \min_{p \in \mathbf{P}_n} \int_{\Omega} |f(x) - p(x)|\, dx \tag{3.8.1}$$

requires integration of a non-smooth function, where we have no *a-priori* knowledge of the roots of $|f(x) - p(x)|$. Problem (3.8.1) is challenging is its own regard as we cannot

apply higher order quadrature methods due to the unknown singularity locations. One globally convergent algorithm has previously been discussed in [64]. In this Section, we present parts of this algorithm and expand on it by discussing the potential to use exact quadrature methods away from singularities. In comparison to the Algorithm presented in [64] we can use asymptotically twice as many quadrature nodes near singularities, leading to a factor 2 error reduction, which we verify experimentally. One possible approximation is to discretise the interval $[0, T]$ using equidistant nodes $(\{x_1, \ldots, x_N\}$ and apply the mid-point rule, leading to a linear program of the form

$$\begin{aligned} &\min_{\mathbf{u},\mathbf{v}\in\mathbb{R}^N,\mathbf{c}\in\mathbb{R}^{k+1}} \sum_{i=1}^{N} w_i(u_i + v_i) \approx \int_0^1 \left| f(x) - \sum_{i=1}^{k+1} c_i U_{i-1}(x) \right| dt, \\ &\text{subject to: } \mathbf{u}, \mathbf{v} \geq 0, \\ &\qquad -\mathbf{v}_i \leq f(x_i) - \sum_{j=1}^{k+1} c_j U_{j-1}(x_i) \leq \mathbf{u}_i. \end{aligned} \tag{3.8.2}$$

Using $\mathcal{O}(N)$ approximation nodes, we would expect the midpoint quadrature error

$$\tau(N) = \left| \int_0^1 f(x)\, dx - \sum_{i=1}^{N} w_i f(x_i) \right|$$

to decrease like $\tau(N) = \mathcal{O}(N^{-2})$. When approximating f using a polynomial function we find a finite number of absolute value type singularities. On each interval, containing such a singularity, we expect an error of size $\mathcal{O}(N^{-2})$, thus leading to an overall quadrature error of

$$\int_0^1 |f(x) - p^{L^1}(x)|\, dx - \sum_{i=1}^{N} w_i \, |f(x_i) - p^*(x_i)| = \mathcal{O}(N^{-2}).$$

We let $p_N(x)$ be the solution of the linear program (3.8.2) in the sense that

$$p_N(x) = \sum_{i=1}^{N} c_i U_{i-1}(x).$$

While the overall quadrature error decreases like $\mathcal{O}(N^{-2})$, the roots of $f - p_N$ only converge to the roots of $f - p^{L^1}$ at a rate of $\mathcal{O}(N^{-1})$. In later parts of this algorithm, we employ Newton iterations on the characterisation of an L^1 approximation

$$\int_0^1 -\operatorname{sign}(f(x) - p^{L^1}(x))q(x)\,dx = 0 \quad \forall q \in \mathbf{P}_k(0,1).$$

An error of $\mathcal{O}(N^{-1})$ is often not sufficient for such Newton iterations to converge. To improve the initial guess for our Newton method, we restructure the quadrature nodes by placing $\mathcal{O}(N^{-1})$ nodes in an $\mathcal{O}(N^{-1})$ region around the estimated roots $\{r_1, \ldots, r_K\}$ of $f - p_N$. Specifically, we let $\delta = \frac{4}{N}$. For each root, we construct a δ ball $B_i = B_\delta(r_i)$. We join intersecting balls B_i to avoid repeated integration over identical intervals, and distribute quadrature nodes equidistantly over the resulting union of intervals. The exact construction of intervals follows from Algorithm 6. Following

Algorithm 6 Interval construction

Require: $N \geq 1, \{r_1, \ldots, r_K\} \subset (0,1)$ ▷ N nodes, r_i roots of $f - p_N$.

res $\leftarrow \{\}$.
$L \leftarrow 0$
for $i \in \{1, \ldots, K\}$ **do**
    $B_i = [r_i - \delta, r_i + \delta]$
    added $\leftarrow$ **False**
    for $B_j \in$ res **do** ▷ Check for intersections
        if $B_i \cap B_j \neq \{\}$ **then**
            $L \leftarrow L - |B_j| + |B_j \cup B_i|$ ▷ Update length
            $B_j \leftarrow B_j \cup B_i$ ▷ Join intersecting intervals
            added $\leftarrow$ **True**
            continue ▷ exit for-loop
        end if
    end for
end for
if not added **then**
    res $\leftarrow$ res $\cup B_i$ ▷ Add new interval
    $L \leftarrow L + |B_i|$ ▷ Update length.
end if
return res, L

Algorithm 6 we have constructed the non-intersecting intervals $\{B_i : i = 1, \ldots, \tilde{K}\}$

with $\tilde{K} \leq K$. We place $\max(\tilde{K}, \lceil N/2 \rceil)$ quadrature nodes equidistantly on these intervals. We follow Algorithm 7 to ensure at least one node is placed on each interval.

Algorithm 7 Node placement

Require: $N \geq 1, \{B_1, \ldots, B_{\tilde{K}}\}, L > 0$ ▷ N nodes, B_i, L from Algorithm6.

nodes $\leftarrow \{\}$.
weights $\leftarrow \{\}$.
for $i \in \{1, \ldots, \tilde{K}\}$ **do**
    $B_i = (l_i, r_i)$
    $N_i \leftarrow \max(2, \lceil \frac{(r_i - l_i) \times N}{2L} \rceil)$
    $\mathbf{x} \leftarrow \texttt{linspace}(l_i, r_i, \text{num} = N_i)$
    nodes $\leftarrow$ nodes $\cup \frac{1}{2}(\mathbf{x}[2:] + \mathbf{x}[1:-1])$ ▷ Add midpoint rule nodes
    weights $\leftarrow$ weights $\cup (\mathbf{x}[2:] - \mathbf{x}[1:-1])$ ▷ Add midpoint rule weights
end for
return nodes, weights

Using Algorithms (6) and (1), we construct the required quadrature nodes and weights near roots of $f - p_N$. Algorithm 6 is able to handle intersecting intervals $B_i \cap B_j \neq \emptyset$ correctly. Algorithm 7 ensures that we place nodes equidistantly and that we assign at least one node to each interval. As an example, we consider the approximation of $f(x) = \exp(x)\sin(10\pi x)$ using a polynomial of degree 5. In Figure 3.10, we see the nodes created by Algorithms 6 and 7 for $N = 100$ and $K = 11$ roots.

3.8.1 Integration away from singularities

Using the previous two node placing algorithms, we place N quadrature nodes at most $\delta = \frac{4}{N}$ away from approximated roots of $f - p_{L^1}$. Despite the singularities this will lead to a quadrature error of $\mathcal{O}(N^{-4})$[3]. Away from the roots, we avoid absolute value type singularities. Let $B_i = (l_i, r_i), B_{i+1} = (l_{i+1}, r_{i+1})$ be two subsequent intervals, as

[3] we are integrating a function of value $\mathcal{O}(N^{-2})$ over an interval of length $\mathcal{N}^{-\epsilon}$.

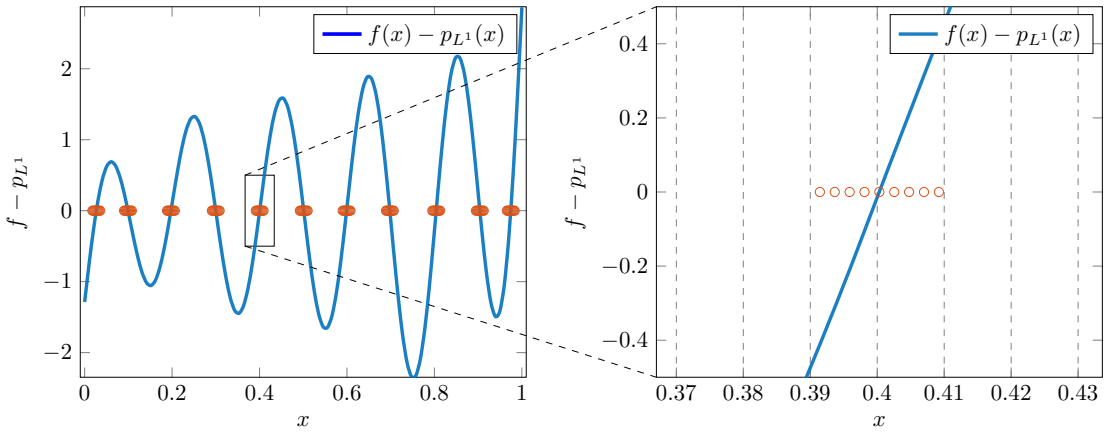


Figure 3.10: Nodes place by Algorithms 6 and 7 on $f(x) = \exp(x)\sin(10\pi x)$ for $N = 100$.

identified by the Algorithm 6. On the interval (r_i, l_{i+1}), we find that

$$\begin{aligned}\int_{r_i}^{l_{i+1}} \left| f(x) - \sum_{j=1}^{k+1} c_j U_{j-1}(x) \right| dx &= \left| \int_{r_i}^{l_{i+1}} f(x) - \sum_{j=1}^{k+1} c_j U_{j-1}(x)\, dx \right| \\ &= \left| \int_{r_i}^{l_{i+1}} f(x)\, dx - \int_{r_i}^{l_{i+1}} \sum_{j=1}^{k+1} c_j U_{j-1}(x)\, dx \right| \\ &= \left| \int_{r_i}^{l_{i+1}} f(x)\, dx - \sum_{j=1}^{k+1} c_j \int_{r_i}^{l_{i+1}} U_{j-1}(x)\, dx \right|.\end{aligned}$$

We denote the area between any to adjacent intervals $B_i = (l_i, r_i),\ B_{i+1} = (l_{i+1}, r_{i+1})$ by $C_i = (r_i, l_{i+1})$ for $i = 1, \ldots, \tilde{K}-1$. Additionally, we let $C_0 = (0, l_1), C_{\tilde{K}} = (r_{\tilde{K}-1}, 1)$. One each of the intervals C_i we can integrate f efficiently using higher order methods, like Clenshaw–Curtis quadrature and integrate the Chebyshev polynomials of second kind U_j for $j = 0, \ldots, k$ exactly. We include the error contribution on the segments

C_i in our linear program formulation using

$$\begin{aligned}
&\min_{\mathbf{u},\mathbf{v}\in\mathbb{R}^N,\tilde{\mathbf{u}},\tilde{\mathbf{v}}\in\mathbb{R}^{\tilde{K}+1},\mathbf{c}\in\mathbb{R}^{k+1}} \sum_{i=1}^{N} w_i(u_i+v_i) + \sum_{i=1}^{\tilde{K}+1} \tilde{u}_i + \tilde{v}_i \\
&\quad \text{subject to: } \mathbf{u},\mathbf{v},\tilde{\mathbf{u}},\tilde{\mathbf{v}} \geq 0, \\
&\qquad\qquad -v_i \leq f(x_i) - \sum_{j=1}^{k+1} c_j U_{j-1}(x_i) \leq u_i, \\
&\qquad\qquad -\tilde{v}_i \leq \int_{C_{i-1}} f(x)\,dx - \sum_{j=1}^{k+1} c_j \int_{C_{i-1}} U_{j-1}(x)\,dx \leq \tilde{u}_i.
\end{aligned} \tag{3.8.3}$$

where $x_i, w_i \in \mathbb{R}\ \ \forall i = 1,\ldots,N$ are the nodes and weights obtained from Algorithm 7. Using this treatment, we obtain a linear program with $k + 2(N + \tilde{K}) + 3$ variables.

3.8.2 Numerical experiments

We have discussed how to place quadrature nodes using Algorithms (6) and (7), resulting in the linear program formulation in equation (3.8.3). An alternative method, which does not require higher order quadrature methods on the intermediate intervals $C_0,\ldots,C_{\tilde{K}}$ is to place $N/2$ equidistantly on the intervals $B_1,\ldots,B_{\tilde{K}}$ and a further $N/2$ nodes equidistantly on $C_0,\ldots,C_{\tilde{K}}$. We place these nodes by applying Algorithm 7 using $N/2$ nodes to both sets of intervals $\{B_1,\ldots,B_{\tilde{K}}\}, \left\{C_0,\ldots,C_{\tilde{K}+1}\right\}$, obtaining the node and weight vectors $\mathbf{x},\mathbf{w}$. Using these nodes and midpoint quadrature weights, we obtain the linear program formulation

$$\begin{aligned}
&\min_{\mathbf{u},\mathbf{v}\in\mathbb{R}^N,\mathbf{c}\in\mathbb{R}^{k+1}} \sum_{i=1}^{N} w_i(u_i+v_i) \\
&\quad \text{subject to: } \mathbf{u},\mathbf{v} \geq 0, \\
&\qquad\qquad -v_i \leq f(x_i) - \sum_{j=1}^{k+1} c_j U_{j-1}(x_i) \leq u_i.
\end{aligned} \tag{3.8.4}$$

We now compare the linear program formulations (3.8.3) and (3.8.4) on increasingly oscillatory test functions

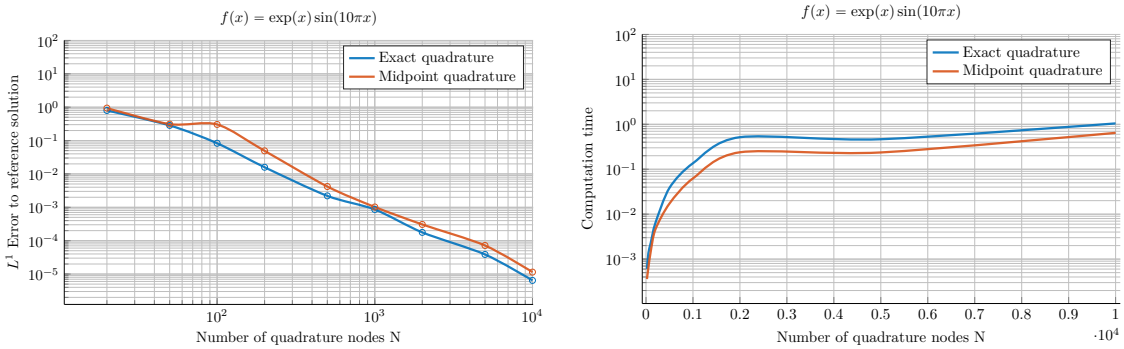


Figure 3.11: Convergence and Computation time of the linear program 3.8.3 and 3.8.4 on f_1

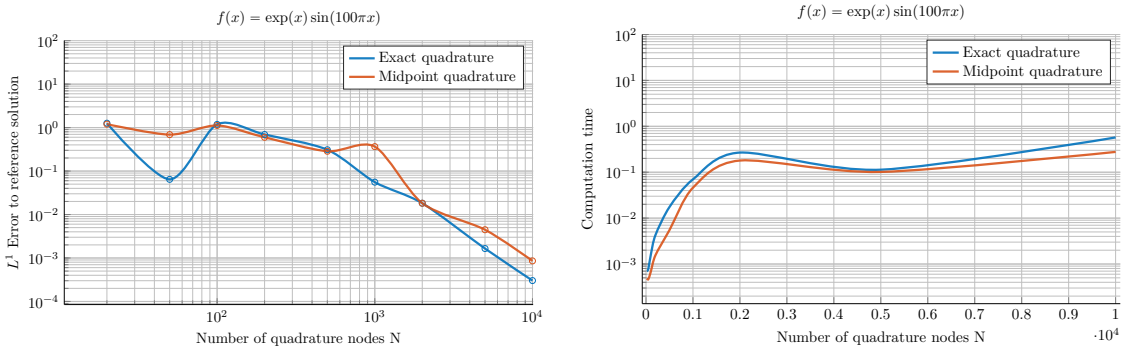


Figure 3.12: Convergence and Computation time of the linear program 3.8.3 and 3.8.4 on f_2

- $f_1(x) = \exp(x)\sin(10\pi x)$,
- $f_2(x) = \exp(x)\sin(100\pi x)$,
- $f_3(x) = \exp(x)\sin(1000\pi x)$.

We see convergence results for the test functions f_1, f_2, f_3 in Figures 3.11, 3.12, 3.13. In each of the tests, we fix the polynomial degree to be $k = 5$ and we computed reference solutions using the linear program formulation 3.8.2 and $N = 10^6$ quadrature nodes. The special choice of increasingly oscillatory test functions allows us to see the effects of more concentrated sampling. In the cases of f_2 and f_3, we can also see regions where the presence of too many absolute value singularities broke the constant sign assumption on the intermediary intervals C_i.

In the linear program 3.8.3, we placed N nodes on intervals, of size $K \times \frac{4}{N}$, leading

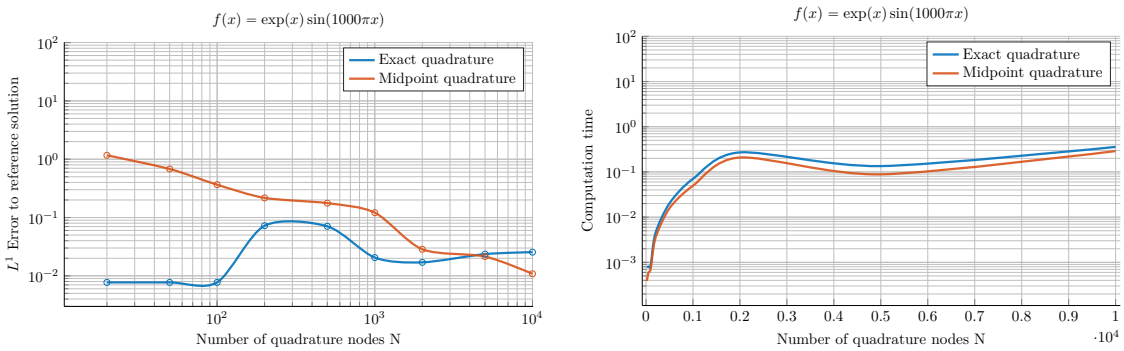


Figure 3.13: Convergence and Computation time of the linear program 3.8.3 and 3.8.4 on f_3

to an average node distance of $\frac{4K}{N^2}$. Hence we place nodes in program 3.8.3, are placed at twice the average density compared to program 3.8.4. From this, we expect a constant factor 2 error reduction in Figures 3.11, 3.12, 3.13. In Figure 3.11, we see this expected behaviour. In Figure 3.12, we observe this behaviour for a sufficiently large number of quadrature nodes $N \geq 2000$.

As our test functions are increasingly oscillatory, we expect the number of roots of $f - p_{L^1}$ to grow. Consequently to see the benefit of the program formulation (3.8.3) over program (3.8.4), we need an increasingly large number of quadrature nodes N. We can observe this behaviour in Figure 3.13 from the slow convergence when using the program formulation 3.8.3.

Chapter 4

Reinforcement learning for lap time minimisation

Reinforcement learning comprises algorithms to maximise the expected cumulative reward an agent achieves within a reinforcement learning environment. Common examples for reinforcement learning agents and environments are robots, both physical and simulated, characters in video games, or stock trading algorithms on simulated exchanges. In this chapter we will reformulate the optimal control problem from Chapter 3 in as a reinforcement learning environment. Our focus will be on the construction of feature sets to improve the training process and quality of the resulting algorithms. The trained agent will then be used to provide a high-quality initial solution to the collocation problem, with the goal of improving both solver performance and accuracy.

4.1 Introduction to reinforcement learning algorithms

A reinforcement learning problem consists of at least two main components (i) an agent and (ii) an environment. The flow diagram in Figure 4.1 shows the interaction between agent and environment. In each time-step, the agent A receives an update

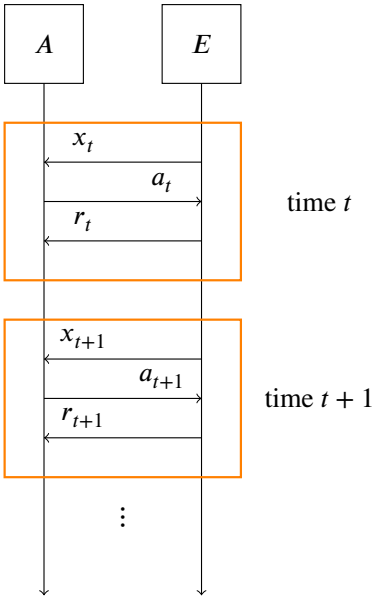


Figure 4.1: Reinforcement learning environment information flow.

about the state information, called observation $x_t \in \mathcal{X}$, from the environment E, to which it responds with a corresponding action $a_t \in \mathcal{A}$. The environment then evaluates the action and responds with a reward r_t, as well as a termination signal $d_t \in \{\text{True}, \text{False}\}$, to indicate whether or not the environment has reached the terminal state.

In recent years, we saw a variety of reinforcement learning algorithms, which rely on deep neural networks as general function approximators. Some of the most well known algorithms of this type are deep Q-learning [63] (DQN), Deep Deterministic Policy Gradient [88] (DDPG), Proximal Policy Optimisation [85] (PPO), and Soft actor critic [41] (SAC). The objectives in developing a deep reinforcement learning

algorithm are to achieve high data efficiency, robustness, and scalability to larger models as well as training on parallel computing infrastructure. In this context, we say that an algorithm is robust if it achieves high performance on a variety of environments, with only minor hyper-parameter tuning. In the following we provide a brief introduction to reinforcement learning algorithms, with a focus on implementation specific details.

The goal of reinforcement learning is to select actions which maximise the expected cumulative reward. For this, we define the trajectory of an agent as

$$\tau = \{(x_t, a_t, r_t) : t \in \{1, \dots, T\}\},$$

as a sequence of state-action-reward tuples. For such a trajectory, we define the cumulative reward as

$$R_t(\tau) = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'},$$

where $\gamma \in (0, 1)$ is called a *discount factor*, such that bounded reward functions on infinite time domains result in finite cumulative rewards. In environments with deterministic rewards functions, we often write $r_t = r(x_t, a_t)$, such that

$$R_t(\tau) = \sum_{t'=t}^{\infty} \gamma^{t'-t} r(x_t, a_t).$$

Additionally, for the purpose of time-optimal control, the discount factor will be important as is encourages the agent to obtain rewards sooner, rather than later.

4.1.1 Q-learning

Q-learning was introduced in 1989 as a model free reinforcement learning algorithm [96]. It provides agents with the ability to learn optimal behaviours in Markov do-

mains, without building a map of the domain. Similar to temporal difference learning [89, 30], an agent evaluates an action, at a state x_t, based on the immediate reward r_t and an estimate of the following state.

In the setting of Q-learning, the environment is a controlled Markov process, with the agent as the controller. The agent moves in a finite and discrete state space $\mathcal{X}$, selecting actions from an action space $\mathcal{A}$, which is also finite. In each time step, the state of the environment changes stochastically according to the transition probability distribution function

$$p(x|x_t, a_t) = \mathbb{P}\left[x_{t+1} = x \mid x_t, a_t\right]. \tag{4.1.1}$$

The agent faces the task of determining the optimal policy $\pi : \mathcal{X} \to \mathcal{A}$, as a map from state-space to the action-space, which maximises the expected discounted sum of future rewards, known as the value function

$$V^{\pi}(x_t) := r(x_t, \pi(x_t)) + \gamma \sum_{x \in \mathcal{X}} V^{\pi}(x) p(x|x_t, \pi(x_t)). \tag{4.1.2}$$

According to the dynamic programming theory of Bellman and Dreyfus [6], we are guaranteed the existence of at least one optimal stationary policy π^*, such that

$$V^{\pi^*}(x_t) \equiv \max_{a_t \in \mathcal{A}} \left(r(x_t, a_t) + \gamma \sum_{x_{t+1}} V^{\pi^*}(x \in \mathcal{X}) p(x|x_t, a_t) \right). \tag{4.1.3}$$

During Q-learning, the agent is tasked with determining an optimal policy π^*, without initial knowledge of the reward function $r(x_t, a_t)$ or the transition probabilities $p(x_{t+1}|x_t, a_t)$. In [96], Watkins introduces Q-learning as incremental dynamic programming, due to the step by step manner, in which the optimal policy is approximated. For a policy π the state-action value function, called Q-function, is defined

as

$$Q^{\pi}(x_t, a_t) = r(x_t, a_t) + \gamma \sum_{x_{t+1} \in \mathcal{X}} V^{\pi}[x_{t+1}] p(x_{t+1}|x_t, a_t), x_{t+1}), \tag{4.1.4}$$

where it is easy to show that

$$a_t^* = \text{argmax}_{a_t} Q^{\pi}(x_t, a_t), \qquad V^{\pi}[x_t] = Q^{\pi}(x_t, a_t^*).$$

In each time step of the reinforcement learning environment, the Q-learning agent will perform a slight adjustment to $Q(x_t, a_t)$, based on a learning rate $\alpha > 0$, by letting

$$Q_n(x, a) = \begin{cases} (1 - \alpha_n) Q_{n-1}(x, a) + \alpha_n \left(r_n + \gamma V_{n-1}(y_n) \right), & \text{if } x = x_n, a = a_n \\ Q_{n-1}(x, a), & \text{otherwise.} \end{cases} \tag{4.1.5}$$

Note that this description requires a look-up-table representation for the $Q_n(x, a)$. The storage and computational requirements for this approach thus scale exponentially with the state and action space dimension. This makes, in particular, discrete approximations to high dimensional continuous action- and state-spaces prohibitively expensive. In [96], a convergence proof for Q-learning, under conditions on the learning rate, is provided leading to Theorem 4.1.1.

Theorem 4.1.1. *Let $n^i(x, a)$ be the index of the step in which action a is chosen in state x for the i^{th} time. If the exploration process is such that $n^i(x, a)$ can be computed for every $i \in \mathbb{N}_{\geq 1}$, the rewards are bounded $|r_n| \leq R$, and the learning rates $\alpha_{n^i(x,a)}$ satisfy*

$$\sum_{i=1}^{\infty} |\alpha_{n^i(x,a)}| = \infty, \quad \sum_{i=1}^{\infty} |\alpha_{n^i(x,a)}|^2 < \infty \ \ \forall x \in \mathcal{X}, a \in \mathcal{A},$$

then $Q_n(x, a) \to Q^*(x, a)$ as $n \to \infty$.

4.1.2 Deep Q-learning

As outlined in section 4.1.1, Q-learning is inherently computationally expensive to apply to environments with large, or continuous, state and action spaces. The idea described in [63] is to utilise the advances made by the use of deep neural networks in computer vision [86, 57]. At the time of writing, most successful reinforcement learning applications operated on domains with hand-crafted features, combined with linear value functions, or policy representations. The performance of such algorithms depended heavily on the quality of these features. From a machine learning perspective, reinforcement learning problems require us to solve several challenging problems simultaneously. The reward function signal deep reinforcement agents have to learn on is often delayed, sparse, or noisy. In some environments, the feedback between actions and reward can even be delayed by several hundred time steps. This is often the case when intermediate rewards are difficult to define, such that we may only provide a sparse reward signal at a terminal states, once we can determine if the agent has solved a given problem successfully. In supervised learning, we often assume that data samples are independent. This is inherently difficult to achieve in a reinforcement learning setting, as consecutive environment observation are naturally highly correlated. If the learning algorithm assumes the underlying sample distribution to be stationary, the agent's own learning progress could also prove problematic.

In [63], an agent is trained on an environment E with a finite action space $\mathcal{A} = \{1, \ldots, K\}$. The particular environment used in [63] was an Atari-2600 emulator. The state of this Atari-2600 environment is taken directly from the emulator's pixel data. In each step, the agent also receives the change in game score as a reward signal.

In many game environments, it is not possible to understand the game state fully from a single frame observation. We consider, as an example, the velocity of a game character. While velocity information can not be obtained from a single frame, it

is possible to extract this information from a sequence of frames. Thus the state data passed to the agent is given by the pixel data from multiple previous frames $x_t = (y_{t-k+1}, \ldots, y_t) \in \mathbb{R}^{k\times(d_h\times d_v)}$, where $d_h \times d_v$ is the resolution of the game environment we observe. The goal of the agent is to interact with the emulator in a manner that maximises expected future reward. As the full internal state of the emulator can not be observed, and state transitions are potentially non-deterministic, the discounted expected future reward function, up to the time index T, at which the terminal state is attained,

$$R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k,$$

is maximised. Similar to classical Q-learning, the deep Q-learning algorithm seeks to compute the optimal state-action value function $Q^*(x, a)$. This optimal state-action value function will satisfy the *Bellman* equation

$$Q^*(x_t, a_t) = \mathbb{E}[r + \gamma \max_{a_{t+1}} Q(x_{t+1}, a_{t+1}) \mid x_t, a_t]. \tag{4.1.6}$$

We use an iterative update for the state-value function, based on the Bellman property given in equation (4.1.6)

$$Q_{i+1}(x_t, a_t) = \mathbb{E}[r + \gamma \max_{a_{t+1}} Q_i(x_{t+1}, a_{t+1}) \mid x_t, a_t]. \tag{4.1.7}$$

On its own, the value iteration given in (4.1.7) is impractical as $Q(\cdot, \cdot)$ requires iterative updates for each state-action value pair. For this reason, a deep neural network function approximator is used

$$Q(x_t, a_t; \theta_i) \approx Q^*(x_t, a_t).$$

This network $Q(x_t, a_t; \theta_i)$ is referred to as the Q-network an optimised iteratively by minimising a sequence of loss functions

$$L_i(\theta_i) = \mathbb{E}[(v_i - Q(x_t, a_t; \theta_i))^2],$$

where

$$y_i = \mathbb{E}[r + \gamma \max_{a_{t+1}} Q(x_{t+1}, a_{t+1}; \theta_{i-1}) \mid x_t, a_t]. \tag{4.1.8}$$

Following [63], differentiating the loss function with regard to the policy weights, we derive

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma \max_{a_{t+1}} Q(x_{t+1}, a_{t+1}; \theta_{i-1}) - Q(x_t, a_t; \theta_i)\right) \nabla_{\theta_i} Q(x_t, a_t; \theta_i)\right].$$

The deep Q-learning algorithm works by updating the state-value network parameters θ in every learning step according to

$$\theta_{i+1} = \theta_i - \nabla_{\theta_i} L_i(\theta_i). \tag{4.1.9}$$

Note that this algorithm is *model free*, in the sense that it does not construct a model for the underlying environment E, and that it is also of *action-value* type as it learns the greedy policy

$$a_t = \pi_\theta(x_t) := \text{argmax}_{a_t} Q(x_t, a_t; \theta). \tag{4.1.10}$$

4.1.3 Policy Optimisation methods

The Q-learning methods which we have seen so far fall into the category of *action-value* methods. Policy functions in action-value methods does not exist explicitly,

as such, we need to evaluate of action values and extract the policy according to Equation (4.1.10). In policy optimisation methods, we parametrise the policy function directly. Consequently, we will have a parametrised map π_θ from the state space $\mathcal{X}$ to the set of probability measures on our action space $\mathcal{A}$

$$\pi_\theta : \mathcal{X} \to \mathcal{M}_1(\mathcal{A}). \tag{4.1.11}$$

The natural goal of policy methods is to compute the largest possible policy parameter improvement, given limited available data, while preventing destructively large policy changes. A popular strategy for choosing an appropriately sized policy step is called Proximal Policy Optimisation (PPO). Originally presented in [85] is an *on-policy* algorithm, which prevents destructively large policy steps optimising a purposefully designed policy value proxy function. We commence our discussion of the PPO algorithm by a comparison of on-policy methods compared to off-policy methods. On policy methods collect a new sample data-set before computing a policy update. As a consequence, the training data always matches the current policy. This match comes however at additional computational cost. *Off-policy* methods, such as deep Q-learning, store previous samples in a large *replay buffer*. During training off-policy methods, sample random batches from this replay buffer. The random sampling is used to reduce the correlation of subsequent samples in the replay buffer. In comparison to action-value methods, policy optimisation has some practical and theoretical advantages. By parametrising the policy directly using smooth functions, the resulting policy will react smoothly to changes in the parameter vector. This is not the case when we select actions in an ϵ-greedy manner from action-value samples. On-policy methods collect samples into a *roll-out buffer* data structure and train only on the recently collected data. The roll-out buffer is usually smaller in size compared to an off-policy replay buffer. As such off-policy methods have the ability to use sam-

ples more often compared to on-policy methods, and we often observe higher sample efficiency with off-policy learning methods. This roll-out buffer is a data structure consisting of three matrices $M_{\text{state}}, M_{\text{state}'}, M_{\text{action}}$ and two vectors $b_{\text{reward}}, b_{\text{final}}$. We will discuss the generation of roll-out samples in more detail later when we discuss Algorithm 11. Using the samples collected in a roll-out buffer, on-policy algorithms will proceed to maximise some scalar performance measure of the networks parameter vector $J(\theta)$. In our case, we focus on environments with non-random initial states x_0, so that we may use

$$J(\theta) = V^{\pi_\theta}(x_0) = \sum_{t=0}^{\infty} \gamma^t r(x_t, a_t). \tag{4.1.12}$$

From the policy gradient theorem (see [90, p. 325]), we obtain

$$\nabla J(\theta) \propto \sum_{x \in \mathcal{X}} \mu_\pi(x) \sum_{a \in \mathcal{A}} \nabla \pi_\theta(a|x) Q^\pi(x, a), \tag{4.1.13}$$

where $Q^\pi(x, a)$ is the expected value of executing action a in state state x and following policy π thereafter and μ_π is the on-policy state distribution following π. We extend Equation (4.1.13) by including a baseline value function $V : \mathcal{X} \to \mathbb{R}$ to obtain

$$\nabla J(\theta) \propto \sum_{x \in \mathcal{X}} \mu_\pi(x) \sum_{a \in \mathcal{A}} \nabla \pi_\theta(a|x)(Q^\pi(x, a) - V(x)). \tag{4.1.14}$$

Equation (4.1.14) remains valid as the quantity we subtracted is

$$\sum_{a \in \mathcal{A}} \nabla \pi_\theta(a|x) V(x) = V(x) \nabla \underbrace{\sum_{a \in \mathcal{A}} \pi_\theta(a|x)}_{=1 \quad \forall \theta} = 0.$$

We call the function $A(x,a) := Q^\pi(x,a) - V(x)$ the advantage function. With this we can write the resulting policy gradient as

$$\nabla J(\theta) \propto \sum_{x\in\mathcal{X}} \mu_\pi(x) \sum_{a\in\mathcal{A}} \nabla\pi_\theta(a|x) \left(Q^\pi(x,a) - V(x)\right), \tag{4.1.15}$$

$$= \sum_{x\in\mathcal{X}} \mu_\pi(x) \sum_{a\in\mathcal{A}} \nabla\pi_\theta(a|x) A^\pi(x,a), \tag{4.1.16}$$

where x' is the environment state following from executing action a in state x and. As the advantage function is not known directly, we employ advantage estimation techniques from [84]. To do this, we introduce the γ-discounted temporal difference residual, as in [90], as

$$\delta_t = r_t + \gamma V(x_{t+1}) - V(x_t). \tag{4.1.17}$$

Using this we find $A(x_t, a_t) = \mathbb{E}\left[\delta_t\right]$. We expand this estimate using a telescoping sum to find

$$\hat{A}^{\pi,1} := \delta_t = r_t + \gamma V(x_{t+1}) - V(x_t), \tag{4.1.18}$$

$$\hat{A}^{\pi,2} := \delta_t + \gamma\delta_{t+1} = r_t + \gamma r_t + \gamma^2 V(x_{t+2}) - V(x_t), \tag{4.1.19}$$

$$\vdots$$

$$\hat{A}^{\pi,N} := \sum_{k=0}^{N-1} \gamma^k \delta_{t+k} = \sum_{k=0}^{N-1} \gamma^k r_{t+k} + \gamma^N V(x_{t+N}) - V(x_t). \tag{4.1.20}$$

We use exponential averaging to obtain the general advantage estimate

$$\hat{A}^{\pi,\mathrm{GAE}(\gamma,\lambda)} := (1-\lambda)\left(\hat{A}^{\pi,1} + \lambda\hat{A}^{\pi,2} + \lambda^2\hat{A}^{\pi,3} + \dots\right) \tag{4.1.21}$$

$$= \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k}. \tag{4.1.22}$$

We now have a parameter $\lambda \in [0, 1]$ to interpolate smoothly between the two extreme cases of one step temporal difference error and a Monte-Carlo advantage function estimate which uses the entire episode

$$\hat{A}^{\pi,\mathrm{GAE}(\gamma,0)} = \delta_t = r_t + \gamma V(x_{t+1}) - V(x_t), \tag{4.1.23}$$

$$\hat{A}^{\pi,\mathrm{GAE}(\gamma,1)} = \sum_{k=0}^{\infty} \gamma^k r_{t+k} - V(x_t). \tag{4.1.24}$$

Following the experimental analysis in [84], we see that the most promising values for λ are in the interval $(0.9, 0.99)$. To use this, we first re-write our performance measure gradient in involve the sampled states and actions x_t, a_t

$$\nabla J(\theta) \propto \sum_{x \in \mathcal{X}} \sum_{a \in A} \nabla \pi_\theta(a|s) A^\pi(x, a) \tag{4.1.25}$$

$$= \mathbb{E}\left[\sum_{a \in A} A^\pi(x_t, a)\pi(a|x_t)\frac{\nabla \pi(a|x_t)}{\pi(a|x_t)}\right] \tag{4.1.26}$$

$$= \mathbb{E}_\pi \left[A^\pi(x_t, a_t)\nabla \log\left(\pi(a_t|x_t)\right)\right]. \tag{4.1.27}$$

The operator $\mathbb{E}_\pi$ is used to signify that actions are generated according to our policy π. Replacing the exact advantage function with our estimator from Equation (4.1.22) and using the sample expectation $\hat{\mathbb{E}}$ yields the performance gradient estimate

$$\nabla \hat{J}(\theta) = \hat{\mathbb{E}}_\pi \left[\hat{A}^{\pi,\mathrm{GAE}(\gamma,\lambda)}(x_t, a_t)\nabla \log\left(\pi(a_t|x_t)\right)\right]. \tag{4.1.28}$$

One option is to perform a gradient ascent step using this gradient estimate from Equation (4.1.28). The resulting gradient step forms the foundation for the classical REINFORCE algorithm [97]

$$\theta_{t+1} = \theta_t + \alpha \hat{\mathbb{E}}_\pi \left[\hat{A}^{\pi,\mathrm{GAE}(\gamma,\lambda)}(x_t, a_t)\nabla \log\left(\pi(a_t|x_t)\right)\right]. \tag{4.1.29}$$

To see the approach taken by PPO we backtrack from Equation (4.1.28) and observe, by undoing the gradient operation, that our goal is to maximise the approximate performance measure

$$\hat{J}(\theta) = \hat{\mathbb{E}}_{\pi}\left[\hat{A}^{\pi,\mathrm{GAE}(\gamma,\lambda)}(x_t, a_t)\log\left(\pi(a_t|x_t)\right)\right]. \tag{4.1.30}$$

One intuitive option to avoid destructively large policy changes is to introduce a trust region parameter $\Delta > 0$ and limit the KL-divergence of a policy update

$$\begin{aligned} &\max_{\theta' \in \mathbb{R}^N} \hat{J}(\theta + \theta') \\ &\text{subject to: } \hat{\mathbb{E}}\left[\mathrm{KL}\left[\pi_\theta(\cdot, x_t), \pi_{\theta+\theta'}(\cdot, x_t)\right]\right] \leq \Delta. \end{aligned} \tag{4.1.31}$$

The objective (4.1.31) results in an algorithm known as trust-region policy optimisation (TRPO) [83]. Proximal policy optimisation reduces the computational complexity of solving (4.1.31) by modifying the objective function in a manner that discourages large policy changes. For this, we let $\hat{A} = \hat{A}^{\pi,\mathrm{GAE}(\gamma,\lambda)}(x_t, a_t)$ and

$$r(\theta', \theta) = \frac{\pi_{\theta'}(a_t|x_t)}{\pi_\theta(a_t|x_t)}. \tag{4.1.32}$$

The PPO objective function cuts of the policy change benefits which are larger than some constant $\epsilon > 0$, such that we solve

$$\max_{\theta' \in \mathbb{R}^N} \hat{\mathbb{E}}_{\pi}\left[\min\left\{\hat{A}r(\theta', \theta), \mathrm{clip}\left(r(\theta', \theta), 1-\epsilon, 1+\epsilon\right)\hat{A}\right\}\right] \tag{4.1.33}$$

The benefits of PPO over TRPO is the reduced computational complexity at similar, or sometimes even higher, empirical learning performance. This learning performance is discussed in [85].

Recalling the advantage estimation in equation (4.1.22), we see that to evaluating $\hat{A}^{\mathrm{GAE},\gamma,\lambda}$ requires the exact value function V. To deal with this in praxis, we will

replace the exact value function using a parametrised function approximator $V_\omega : \mathcal{X} \to \mathbb{R}$, where $\omega \in \mathbb{R}^M$ is a parameter vector. For a given trajectory

$$\tau = \{(x_t, a_t, r_t), \ldots, (x_T, a_T, r_T)\},$$

we can then use the generalised advantage estimate

$$\hat{A}_t^{\gamma,\lambda,\omega} = \sum_{k=t}^{T} (\gamma\lambda)^{k-t} \left(r_k + \gamma V_\omega(x_{k+1}) - V_\omega(x_k)\right).$$

For the optimal policy π^*, the advantage function $A^*(x_t, \pi^*(x))$ satisfies

$$\begin{aligned}
A^*(x_t, \pi^*(x_t)) &= Q^*(x_t, \pi^*(x_t)) - V^*(x_t) \\
&= \max_{\tilde{a}\in\mathcal{A}} r(x_t, \tilde{a}) + \gamma V^*(x_{t+1}) - V^*(x_t) \\
&= \max_{\tilde{a}\in\mathcal{A}} r(x_t, \tilde{a}) + \gamma \sum_{k=t+1}^{\infty} \gamma^{k-(t+1)} r(x_k, a_k) - \sum_{k=t}^{\infty} \gamma^{k-t} r(x_k, a_k) \\
&= r(x_t, a_t) + \gamma \sum_{k=t+1}^{\infty} \gamma^{k-(t+1)} r(x_k, a_k) - \sum_{k=t}^{\infty} \gamma^{k-t} r(x_k, a_k) \\
&= 0.
\end{aligned}$$

We thus improve the value function by updating the value function parameter ω using, similar to [85, 90]

$$\min_{\omega'} \hat{\mathbb{E}}_\pi \left[\left(V_{\omega'}(x_t) - V_{\omega_k}(x_t) - \hat{A}_t^{\gamma,\lambda,\omega} \right)^2 \right]. \tag{4.1.34}$$

For practical optimisation, we construct a loss function from 4.1.34 and (4.1.33) using a balancing parameter $\kappa > 0$

$$L(\theta, \omega) = \kappa \left(\hat{A}^{\gamma,\lambda,\omega} \right)^2 - \hat{\mathbb{E}}_\pi \left[\min \left\{ \hat{A} r(\theta, \theta_{\text{old}}), \text{clip}\left(r(\theta, \theta_{\text{old}}), 1-\epsilon, 1+\epsilon\right) \hat{A} \right\} \right]. \tag{4.1.35}$$

For the final algorithm we minimise the loss function $L(\theta, \omega)$ using a general stochastic minimisation algorithm, such as the Adaptive Moment Estimation Algorithm (ADAM [55]). In this algorithm, we collect M step samples from the environment in a phase called "roll-out". Using the roll-out samples we construct the loss function L as in Equation (4.1.35) and perform a gradient step. The total number of environment steps is limited to N. As training progresses, we generally reduce the learning α according to a so called learning rate schedule $\alpha = \alpha(k)$, where $t \in \{1, \dots, N\}$ is the current step index. In the following sections 4.2 and 4.2.2 we discuss the impact

Algorithm 8 PPO learning algorithm

Require: $N \geq M \geq 1, \theta, \omega, \alpha : \mathbb{N} \to \mathbb{R}_{\geq 0}$.
$t \leftarrow 0$.
while $t + M \leq N$ **do**
    Initialise a set of trajectories $D \leftarrow \emptyset$.
    Initialise an empty trajectory $\tau \leftarrow \emptyset$.
    for $i = 1, \dots, M$ **do** ▷ Execute M steps in the environment.
        Sample an action from the policy $a_t \sim \pi(x_t)$.
        Execute the step in the environment $x_{t+1}, a_{t+1}, r_{t+1} \leftarrow \texttt{step}(a_t)$.
        $\tau \leftarrow \tau \cup \{(x_{t+1}, a_{t+1}, r_{t+1})\}$.
        if x_{t+1} is a terminal state **then**
            Reset the environment $x_{t+1} \leftarrow \texttt{reset}$.
            Store the sampled trajectory $D \leftarrow D \cup \{\tau\}$.
            Initialise a new trajectory $\tau \leftarrow \emptyset$.
        end if
        $t \leftarrow t + 1$.
    end for
    Add current remaining trajectory to D, $D \leftarrow D \cup \{\tau\}$.
    Compute the advantage function estimates $\hat{A}_t, \dots, \hat{A}_{t+M-1}$.
    Compute loss function $L(\theta, \omega)$.
    Evaluate the loss gradient $\nabla L(\theta, \omega)$ using automatic differentiation.
    Minimise the loss function via some gradient descent algorithm, such as `ADAM` using a learning rate $\alpha(t)$ to obtain θ', ω'.
    Update the parameter vectors $\theta \leftarrow \theta', \omega \leftarrow \omega'$.
end while

of learning rates and reward signals on the PPO learning process.

4.2 Lap time optimisation environment

In the previous section, we provided an introduction to reinforcement learning and discussed some of the seminal algorithms. We first need to construct a reinforcement learning environment to apply these algorithms to a lap time optimisation problem. In the following, we use the classical bicycle model, as described in Chapter 2, as the underlying dynamical model and construct a reinforcement learning environment based on this physical model. While reformulating the lap time problem, we have some freedom how we present the available state information to the neural network controller. We commence with a baseline environment, which only provides vehicle state information, taken directly from the bicycle model, to the optimisation algorithm. During the environment construction, we put particular emphasis on the track representation within the model. In later sections, we will discuss the impact of an expanding the observation space on the learning progress.

To commence, we consider only the states present in the dynamic version of the classical bicycle model. In this model, the vehicle is described by six state variables. Similar to the bicycle model, the state variables to track the vehicle's position in

Table 4.1: Dynamic bicycle model vehicle state variables

Variable	Symbol
Centre of mass position	$(x, y)^T \in \mathbb{R}^2$
Vehicle yaw angle	ψ
Centre of mass velocity	$(x', v')^T \in \mathbb{R}^2$
Vehicle yaw rate	ψ'

relative to the race track are also included in the observation space. We therefore add the three variables from Table 4.2 to the observation. As outlined in Section 2.1, the dynamics equations for the vehicle states are derived by force and momentum balance, while ignoring pitch and roll moments. The dynamics for the position relative to the track follow from computation in curvilinear coordinates.

Table 4.2: Dynamic bicycle model track state variables

Variable	Symbol
Distance along centre line	s
Relative yaw angle	ξ
Distance from centre line	ν

The resulting dynamics for path tracing variables are

$$\partial_t s(t) = \frac{\partial_t x(t)\cos(\xi(t)) - \partial_t y(t)\sin(\xi(t))}{1 - \nu(t)C(s(t))}, \tag{4.2.1}$$

$$\partial_t \nu(t) = \partial_t x(t)\sin(\xi(t)) + \partial_t y(t)\cos(\xi(t)), \tag{4.2.2}$$

$$\xi(t) = \partial_t^2 \psi(t) - \partial_t s(t) C(s(t)). \tag{4.2.3}$$

We assume that the driving force at the rear wheel can be directly controlled by the driver. This is a simplifying assumption, as it does not include the natural time delay effects between throttle movement and engine response [25]. In the following, we use

Table 4.3: Dynamic bicycle model control variables

Variable	Symbol	Range
Rear wheel acceleration	f_r	$[-20\frac{m}{s^2}, 20\frac{m}{s^2}]$
Front axis steering angle	δ	$[-\frac{\pi}{6}, \frac{\pi}{6}]$

the shorthand notation $\mathbf{x}$ and $\mathbf{u}$, with

$$\mathbf{x}(t) = \Big(p_x(t), p_y(t), v_x(t), v_y(t), \psi(t), \psi'(t), s(t), \nu(t), \xi(t)\Big)^T, \tag{4.2.4}$$

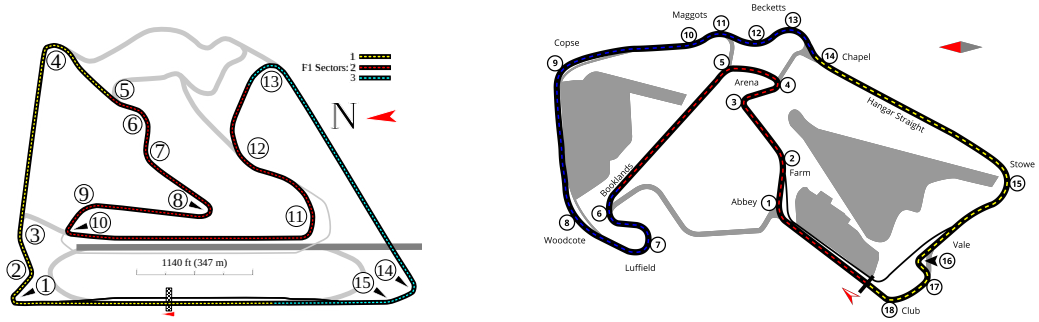
$$\mathbf{u}(t) = \Big(f_r(t), \delta(t)\Big)^T \tag{4.2.5}$$

for the vectors of all state and control variables respectively. To complete the underlying model for the track environment, a flexible track description is required. In particular, we want to be able to procedurally generate new racing tracks. The goal of this is to enable agents that are trained on a selection of randomly generated tracks. This randomised track generation is then used to aid in the generalisation of agents

to new race tracks.

4.2.1 Track description

We seek a model for two dimensional tracks of constant width, which allows for automatic track generation. In Figure 4.2, we can see two exemplary racing circuits



(a) Bahrain International Circuit [76] (b) Silverstone Circuit [27]

Figure 4.2: Example racing tracks in the 2021 formula one calendar

from the 2022 Formula One calendar. Our environment will represent racing tracks using two different track elements, straight pieces and curves. Straight pieces have only their length l as a parameter, while curves contain an angular distance α, as well as a radius r. In addition to an enumeration of track elements, a race track description also contains a starting position x_0, starting angle ω_0, and a track width $W > 0$. This formulation enables procedural generation of new tracks. We have seen in Section 2.1, that we require the track curvature to evaluate the bicycle models dynamics equation. Using the outlined track description, the curvature is given by

$$c(s) = \begin{cases} 0, & \text{for straight elements} \\ \frac{\text{sign}(\alpha)}{r}, & \text{for curve elements} \end{cases}.$$

During each reinforcement learning step, the environment receives an action $\mathbf{u}(t)$ from the neural network controller. Using this action, the environments updates its

internal state using explicit Euler time stepping scheme

$$\mathbf{x}_{t+\delta_t} = \mathbf{x} + \delta_t f(\mathbf{x}(t), \mathbf{u}(t), t).$$

Following the standard truncation error analysis (see [18]), we obtain the truncation error formulation as

$$\begin{aligned} \tau(t) =& \frac{\delta_t^2}{2}\frac{d}{dt} f(\mathbf{x}(t), \mathbf{u}(t), t) + \mathcal{O}(\delta_t^3) \\ =& \frac{\delta_t^2}{2}\left(\frac{d}{dx} f \partial_t \mathbf{x}(t) + \frac{d}{du} f \partial_t \mathbf{u}(t) + \partial_t f\right) + \mathcal{O}(\delta_t^3). \end{aligned} \tag{4.2.6}$$

In Equation (4.2.6), we dropped the explicit dependence of f on $\mathbf{x}(t), \mathbf{u}(t), t$. Considering the truncation error for the vehicle's position, along the track centre line s, we find that

$$\partial_t s(t) = \partial_s \left(\frac{v_x(t)\cos(\xi(t)) - v_y(t)\sin(\xi(t))}{1 - \nu(t)C(s)}\right) \tag{4.2.7}$$

$$= v_x(t)\cos(\xi(t)) - v_y(t)\sin(\xi(t))\frac{\nu(t)C'(s)}{(1-\nu(t)C(s))^2}. \tag{4.2.8}$$

To perform the error analysis in equation (4.2.6), we require that all the derivatives of f are well defined. From Equation (4.2.8) we can observe that this can only be the case if the curvature C is at least continuously differentiable with respect to the centre line distance s. To ensure compatibility with our track description, we smoothly interpolate the curvature C near the boundary of track elements.

We note that both the straight track piece and the curved piece have constant curvature. Let s_0 be a point of intersection between two track pieces of differing, but constant, curvature, such that for $\epsilon > 0$ sufficiently small

$$C(s_0 - \epsilon) = c_{\text{left}}, \qquad C(s_0 + \epsilon) = c_{\text{right}}. \tag{4.2.9}$$

We choose the 5-th degree Hermite-spline $q : [-1, +1] \to [0, 1]$ interpolation function with

$$q(-1) = q'(\pm 1) = q''(\pm 1) = 0, \quad q(1) = 1, \tag{4.2.10}$$

$$q(x) = \frac{3}{16}x^5 - \frac{5}{8}x^3 + \frac{15}{16}x + \frac{1}{2}, \tag{4.2.11}$$

and interpolate

$$\tilde{C}(s) = \begin{cases} C(s), & \text{if } |s - s_0| > \epsilon, \\ c_{\text{left}} + \left(1 - q(\frac{s - s_0 + \epsilon}{2\epsilon})\right) c_{\text{right}}, & \text{otherwise.} \end{cases} \tag{4.2.12}$$

The symmetry of q about $(0, \frac{1}{2})$ ensures that for all $\epsilon > 0$ sufficiently small

$$\int_{-\epsilon}^{\epsilon} C(s_0 + x)\, dx = \int_{-\epsilon}^{\epsilon} \tilde{C}(s_0 + x)\, dx, \tag{4.2.13}$$

so that the track orientation remains aligned with the underlying track description, despite the described curvature interpolation.

With this track description, we now have a clearly defined procedure for simulating the reinforcement learning environment in time using the explicit Euler procedure. The final component required for completing this environment is the reward function. In many cases reward functions have to be specifically designed to work with a given algorithm and on a specific environment. The difficulty in designing such a reward function is in balancing learning progress with ensuring that the correct objective is maximised.

In our particular case, it is natural that we want to minimise the total lap time , while following a valid trajectory around the track. A trajectory would become invalid if the vehicle violates either the track constraint $|\nu(t)| > \frac{W}{2}$, or the bicycle model dynamics. Additionally, we prevent undesirable behaviour, such as driving backwards

or stopping, by including a large negative penalty term in our reward function. As a result, the reward function we use is given in Algorithm 9.

Algorithm 9 Reward function

Require: $\mathbf{x}(t)$
if $|\nu(t)| > \frac{W}{2}$ **then** ▷ Contacted the track boundary.
    Return 0.
end if
if $s'(t) < 0$ **then** ▷ Driving backwards
    Return 0.
end if
if $x'(t)^2 + y'(t)^2 < 1.0$ **then** ▷ Vehicle is driving too slowly
    Return 0.
end if ▷ Penalise duration but reward progress
Return $(s'(t) - 1)\delta_t$.

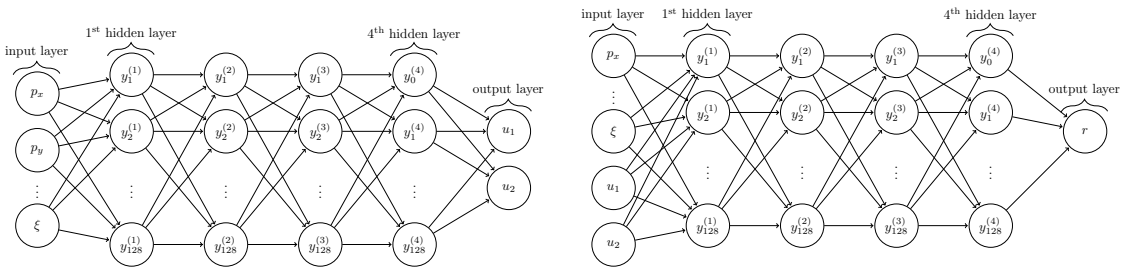
Similar to the reward function, we need a methods of determining whether we have reached a terminal state in our model. We will do this analogous to how we determined rewards in Algorithm 9 by terminating the environment if we have reached an undesirable state or if we have finished the track. We give a pseudocode description of this terminal state function in Algorithm 10.

Algorithm 10 Check terminal state

Require: $\mathbf{x}(t)$
if $|\nu(t)| > \frac{W}{2}$ **then** ▷ Contacted the track boundary.
    Return **true**.
end if
if $s'(t) < 0$ **then** ▷ Driving backwards.
    Return **true**.
end if
if $x'(t)^2 + y'(t)^2 < 1.0$ **then** ▷ Vehicle is driving too slowly.
    Return **true**.
end if
if $|\omega(t)| > 2\pi$ **then** ▷ Lost vehicle control.
    Return **true**.
end if
if $s(t) \geq L$ **then** ▷ Finished the track.
    Return **true**.
end if ▷ Terminal state is not reached
Return **false**.

4.2.2 Learning process and reward signals

At this point, we determined the main components of our learning environment. In particular, we are able to compute the change of system state variables and return a reward signal to the agent. With this, we are able to commence the learning process. For training our neural network controllers, we make use of the project Stable-Baselines3 (SB3). The SB3 project contains an implementation of many seminal reinforcement learning algorithms in using the PyTorch [74] toolkit as well as supportive data structures, such as replay buffers, which aid in the learning process. In particular, we use the implementation of proximal policy optimisation (PPO) from SB3. The PPO agent processes a vehicle state, in the form given in Equation (4.2.5), using a neural network structure consisting of four hidden layers with 128 nodes in each layer. This observation vector $\mathbf{x}$ is mapped to output vector $\mathbf{u} = (u_1, u_2)^T$, corresponding to the vehicle actions. We can see the network structure in Figure 4.11a[1].



(a) Actor network layout

(b) Critic network layout

Figure 4.3: PPO agent network layouts for training on the baseline environment

Here, we will call the input to the k^{th} layer $\mathbf{y}^{(k-1)}$. We use the hyperbolic tangent function tanh as our non-linear activation function. With this, we can write the

[1]The default behaviour of SB3 shares layers between the actor and the critic network. We avoid this behaviour by specifying the network architecture parameters explicitly.

output of the k^{th} layer as

$$\Phi_k(x) := \tanh(W_k x + b_k), \qquad \Phi := \Phi_4 \circ \Phi_3 \circ \Phi_2 \circ \Phi_1. \tag{4.2.14}$$

Where W_k is a matrix of network weights and b_k a vector of biases in layer k. Using the dimensions shown in Figure 4.11a we can see that these matrices have the dimension

$$W_k \in \begin{cases} \mathbb{R}^{128\times 9} & \text{if } k = 1, \\ \mathbb{R}^{2\times 128} & \text{if } k = 5, \\ \mathbb{R}^{128\times 128} & \text{otherwise} \end{cases}, \qquad b_k \in \begin{cases} \mathbb{R}^{2} & \text{if } k = 5, \\ \mathbb{R}^{128} & \text{otherwise} \end{cases}. \tag{4.2.15}$$

We find that the total number of network parameters for our actor network is

$$L_0 = 4(128 \cdot 128) + (2 + 9 + 4) \cdot 128 + 2 = 67458.$$

While we can use the deterministic outputs of our neural network directly as actions, this often leads to poor training performance. For improved training performance, we sample actions from a diagonal Gaussian distribution

$$\mathbf{u} \sim N(\Phi(x), \operatorname{diag}(\sigma_1, \sigma_2)), \tag{4.2.16}$$

where σ_1, σ_2 are additional training parameters with we obtain during the training process. With this, we find

$$\mathbf{u} \sim N\left(\begin{pmatrix} u_1 \\ u_2 \end{pmatrix}, \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}\right), \qquad \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \Phi(\mathbf{x}). \tag{4.2.17}$$

The learning process is split into two phases called *roll-out* and *training*. During *roll-out* state, action, reward samples are collected and stored into a roll-out buffer. The

roll-out buffer is a data structure consisting of three matrices $M_{\text{state}}, M_{\text{state}'}, M_{\text{ctrl}}$ and two vectors $b_{\text{reward}}, b_{\text{final}}$. In Algorithm 11 we see the roll-out process in detail. We see that to execute the roll-out algorithm we we require the two functions `step`, `reset`, which, as the name suggests, step the environment in time or reset the environment to an initial state. During the training process of a PPO agent we collect samples

Algorithm 11 Rollout sample collection

Require: $N \geq 1$, `reset`, `step`

$M_{\text{state}} \leftarrow 0 \in \mathbb{R}^{N \times n}$
$M_{\text{state}'} \leftarrow 0 \in \mathbb{R}^{N \times n}$
$M_{\text{ctrl}} \leftarrow 0 \in \mathbb{R}^{N \times m}$
$b_{\text{reward}} \leftarrow 0 \in \mathbb{R}^{N}$
$b_{\text{final}} \leftarrow \textbf{false}^N \in \{\textbf{true}, \textbf{false}\}^N$
$\mathbf{x}$, final $\leftarrow$ `reset`
for $i \in 1, \ldots, N$ **do**
    if final is **true then**
        $\mathbf{x}$, final $\leftarrow$ `reset`
    end if
    Sample action $\mathbf{u}$ from policy $\pi_\theta(\mathbf{x})$
    $\mathbf{x}', r, \text{final}' \leftarrow \texttt{step}(\mathbf{u})$
    $M_{\text{state}}[i, :] \leftarrow \mathbf{x}$
    $M_{\text{state}'}[i, :] \leftarrow \mathbf{x}'$
    $M_{\text{ctrl}}[i, :] \leftarrow \mathbf{u}$
    $b_{\text{reward}}[i] \leftarrow r$
    $b_{\text{final}}[i] \leftarrow \text{final}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
end for
return $M_{\text{state}}, M_{\text{state}'}, M_{\text{ctrl}}, b_{\text{reward}}, b_{\text{final}}$

using the current policy and use these samples to update the weights of our neural network controller. As discussed in Section 4.1.3, we update the policy and value function weights according to Algorithm 8. To solve this optimisation problem, we sample batches from our replay buffer to obtain stochastic advantage function estimates. Using these estimates, we perform an optimisation step using the popular stochastic Adaptive Moment Estimation Algorithm (ADAM, see [55]), to minimise the loss objective described in Equation (4.1.35). At the current point in time, we only use each sample once to inform our policy function. This may lead to low sample

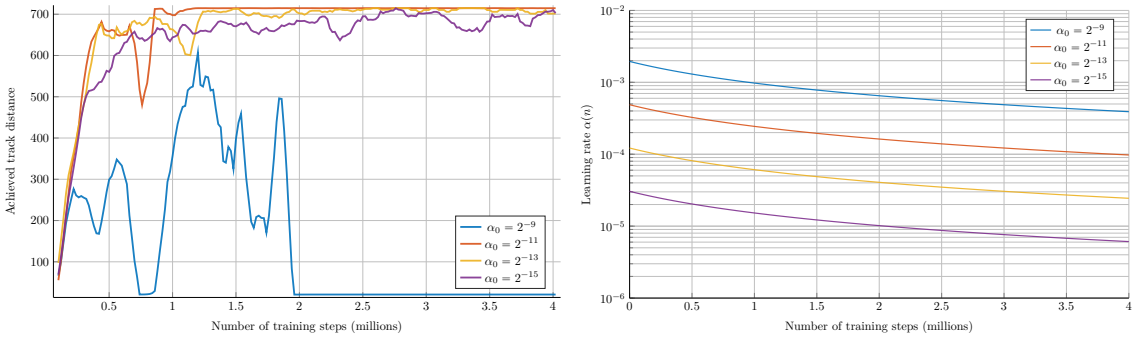
efficiency. To counteract this, we may train multiple times on the collected replay buffer, while re-ordering the also reduce the sample correlation.

We now use the described roll-out and training algorithms to examine training success and discuss how we can adjust training parameters and environment components to improve the training process. With the current configuration, we first analyse the impact of our learning rate. We use harmonically decaying learning rates

$$\alpha(n) = \frac{\alpha_0}{1 + \beta n}, \tag{4.2.18}$$

where n is the current learning step and $\beta = 10^{-6}$ is used as a decay factor. For our analysis, we saved snapshots of the model parameters every 10^4 environment steps. We run the training process in parallel on a 24 thread Ryzen R9-3900X processor at a roll-out length of $N = 512$ steps. This roll-out length corresponds to the number of environment steps executed in line seven of Algorithm 11. As we are collecting training samples in parallel the total roll-out buffer size is $N_{\text{roll-out}} = 24 \times 512 = 12280$. By saving these snapshots, we can gain further insight into stored model than we can glean from training logs. To evaluate the quality of a current stored model we simulate different trajectories while sampling actions according to the currently saved action distribution. In Figure 4.4a, we see the mean performance of models with different initial learning rates α_0, over the course of the training process. The learning rate decay follows Equation (4.2.18), resulting in the learning rate schedule shown in Figure 4.4b.

We see in Figure 4.4a that, while the training process generally has a positive impact, the training performance however is volatile and dependent on the learning rate. Initially, we see that the training process fails to achieve positive results for higher learning rates, such as $\alpha = 2^{-9} \approx 0.02$. In contrast to this, we see more training success at lower learning rates $\alpha \in \{2^{-11}, 2^{-13}, 2^{-15}\}$. In the current learning



(a) Model performance on the example track in Figure 3.3

(b) Learning rate schedule following Equation (4.2.18) for different α_0

Figure 4.4: Base environment progress and learning rate schedule

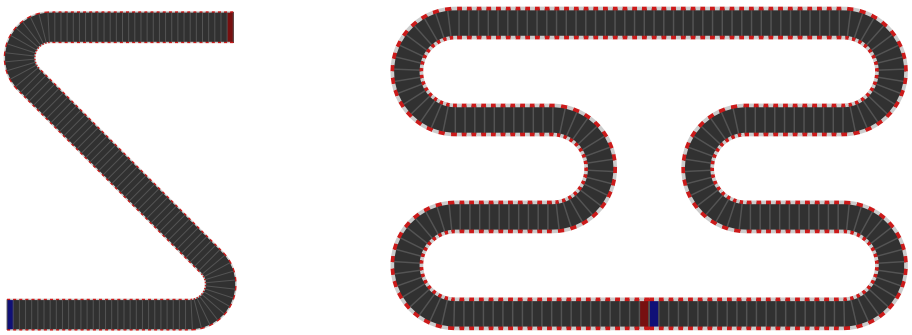
environment, the agent has only access to immediate model variables. While observations, such as the distance to the track boundary, can inform our agent about the immediate surrounding area, most track features need to be learned from the location variables x, y, s. The variables x, y, s represent the global vehicle position, thus models trained only on variables shown in Figure 4.3 will likely not be able to generalise well to different tracks, and learning on complex tracks may prove difficult. For a comparison across different test-tracks we introduce two further test tracks, called "Z"-track and "Double S"-track. We see plots of both these tracks in Figure 4.5, as well as some of the most important track features in Table 4.4. We perform the

Table 4.4: Test tracks

Track Name	Length	Straights	Curves	Total curve angle	Plot
"L" track	714.159	1	2	90.0°	3.3
"Z" track	1260.119	3	2	270.0°	4.5a
"Double-S" track	2142.478	7	6	1080.0°	4.5b

same learning process on the Double S-shape track (see Figure 4.5b), and we can see the evaluated performance, measured by achieved track distance, in Figure 4.6.

Similar to the training results we saw in Figure 4.4a, we see that high learning rates, such as $\alpha_0 \approx 2^{-9}$, lead to unsatisfactory results. While training performance on the 714.16m long "L" track, depicted in Figure 4.4a, is not optimal, the agents are



(a) "Z"-shape test track with 2 curve elements.

(b) "Double S"-shape test-track with 6 curve elements.

Figure 4.5: Visualisation of test tracks from Table 4.4 with nodes at 10m intervals

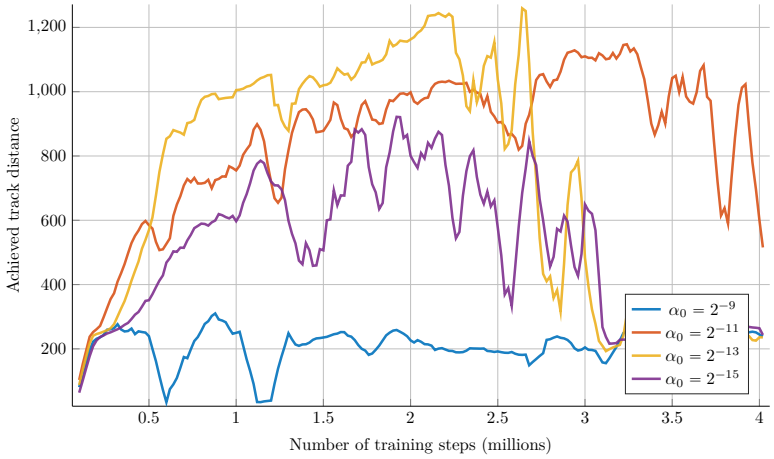


Figure 4.6: Model performance on the example track in Figure 4.5b

generally able to finish the race, given sufficient training time and an appropriately chosen initial learning rate α_0. In contrast to this we see the training performance on the 2142.48m long test track "Double S"-shape. In this case none of the agents were able to finish the track. There are multiple likely reasons for this reduction in learning performance, not least of which is the higher track complexity. An increase in track length increases the complexity of our reward signal. As the vehicle progresses further into the track rewards accumulate, such that the variance in our reward signal increases in cases where agents make mistakes earlier in the race process. As we observe that vehicles are not able to finish the track, we want to encourage further exploration of the track. In this regard we can often benefit from having some controlled variance in our reward signal. In the following we provide a small positive reward, each time the vehicle enters a new track segment. We construct these track segments from the area between two subsequent track nodes. Let $\{s_i : i = 0, \dots, N\}$ be a set of track distances, at which nodes are placed. To remember the track nodes we have already crossed we let $L(t) \in \mathbb{N}_{\geq 0} \ \ \forall t$ denote the index of the track segment, on which the vehicle is currently located, such that

$$s(t) \geq s_i \ \forall i = 0, \dots, L(t).$$

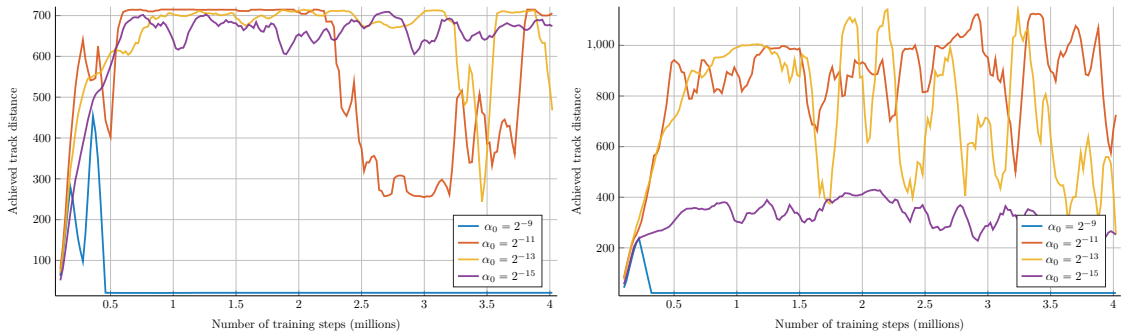
Using this, we can write the pseudocode description of this check-point based reward function as Algorithm 12.

Algorithm 12 Check-point reward function

Require: $s_{t_k}, s_{t_{k+1}}, \nu_{t_{k+1}}, L$ ▷ Track distance during current time step.

```
if |ν_{k+1}| > W/2 then
    return 0
end if
if s_{t_{k+1}} ≥ s_{L+1} then
    L ← L + 1
    return 1
end if
return 0
```

Another notable modification Algorithm 12 makes over Algorithm 9, is that time costs are no longer included. We omit the time cost in Algorithm 12 to encourage exploration where slow driving is required. Later in Section 4.3.2 we will recover some time optimal aspects by considering multi-lap environments. In Figure 4.7b we see the achieved track distances when applying our PPO learning algorithm to a racing environment using the check-point reward function (12).



(a) Model performance on the example track in Figure 3.3 using the check-point reward function (12).

(b) Model performance on the example track in Figure 4.5b using the check-point reward function (12).

Figure 4.7: Model performance using the check-point reward function (12).

The results in Figure 4.7b show the benefit of using the checkpoint reward function. Especially for the learning rates $\alpha = 2^{-11}, \alpha = 2^{-13}$, we observe improvements to the training success during the first one million training steps. We can also see the impact of using the checkpoint based reward function, when applying our learning algorithm to the shorter "L"-shape test track in Figure 4.7a. Comparing Figures 4.4a and 4.7a we see the benefit of check-point based rewards more clearly. Previously the training run with $\alpha = 2^{-9}$ was not able to finish the race track successfully, however using checkpoint rewards all learning rate lead to successfully training algorithms and, perhaps more importantly, satisfactory runs are obtained using fewer environment steps and with less variance in achieved distance.

To obtain algorithms which are able to finish more complex tracks, such as Double-S

successfully, we will consider extensions to the learning environment in the following Sections.

4.3 Extended lap time optimisation environments

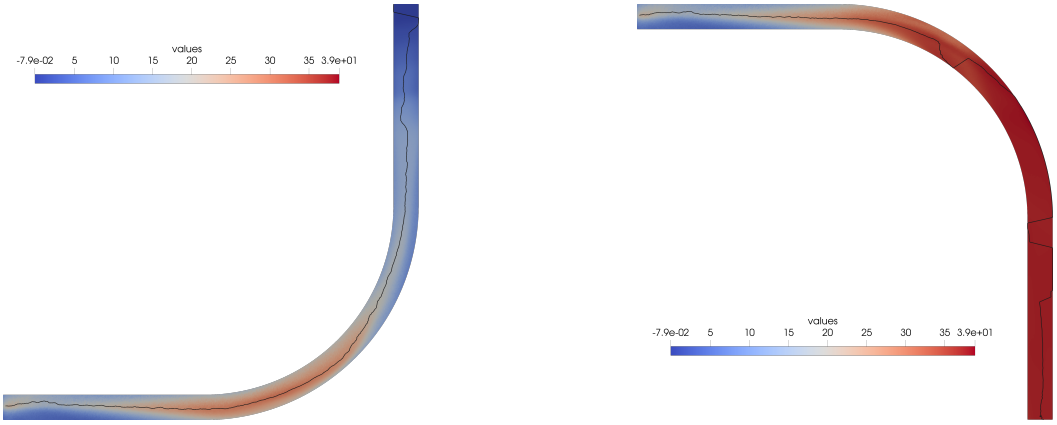
Up to this point, we have limited our discussion to reinforcement learning on a baseline environment, where the observation space is given by vehicle and track state variables. We discuss potential limitations of this approach, and introduce extensions to the observation space formulation. We know, from the optimal control formulation of the lap time optimisation problem, that it is possible to construct an open-loop formulation of the solution. In open-loop solutions, the optimal policy is given in a form, that only depends on the centre line distance s.

In the following, we evaluate how much information a neural network control can get from these state variables. If the neural network would approximate such an open-loop solution, two tracks of the same length would produce identical state-value estimates. Let V_θ be the value network of our PPO agent. We proceed to estimate the position values on a sample track. We have pre-trained a PPO agent on a sample track. This sample track consists of only a single left curve. The agent then predicts position values on the known track and a flipped version of the track, consisting of a single right curve. We describe each position using the coordinates $(s, \nu)^T \in [0, L] \times [-\frac{W}{2}, \frac{W}{2}]$, where L is the track length and W is the track width. For each, coordinate we sample the remaining vehicle states from the distributions

$$
\begin{aligned}
v_x &\sim \text{Unif}\left([1.0, 30.0]\right), & v_y &\sim \text{Unif}\left([-3.0, 3.0]\right), \\
\psi &\sim \text{Unif}\left(\left[-\frac{\pi}{6}, \frac{\pi}{6}\right]\right), & \psi' &= 0.
\end{aligned}
$$

Similar to before, we denote the vector of all vehicle states by $\mathbf{x}$. We can then estimate the value function by evaluating both the actor and the critic networks con-

tained in a PPO model. Given a state sample $\mathbf{x}$, the actor model π_θ generates and action sample $\mathbf{u} \sim \pi_\theta(\mathbf{x})$. Using both $\mathbf{x}$ and $\mathbf{u}$, we can estimate the value using the critic network $v = V_\theta(\mathbf{x}, \mathbf{u})$. For each position (s, ν) we average the values predicted by $V_\theta(\cdot, \cdot)$, using 512 state samples. Figures 4.8a and 4.8b show the experimental evaluations of state values. The line in Figures 4.8a and 4.8b indicates the trajectory of highest state values along the track distance variable s. While the lines in Figures 4.8a, 4.8b do not generally contain valid trajectories, they show how a trained agent views the track values.



(a) Left curve track position values

(b) Right curve track position values

Figure 4.8: Track position values predicted by a PPO agent trained on the left curve track in a baseline environment.

In Figure 4.8a, we see that the highest state value trajectory largely follows a feasible racing line. Towards the end of the race track, the trajectory in Figure 4.8a starts to oscillate, indicating that learning state values for later track positions is more difficult. We compare this to the trajectory on the right curve track, as shown in Figure 4.8b. Figure 4.9 shows a direct comparison of the trajectories on the two curves.

The differences in position values between Figures 4.8a and 4.8b suggest poor generalisability of our agent. While the lateral track position variable ν and the relative orientation variable ξ would allow our agent to follow the tracks curvature,

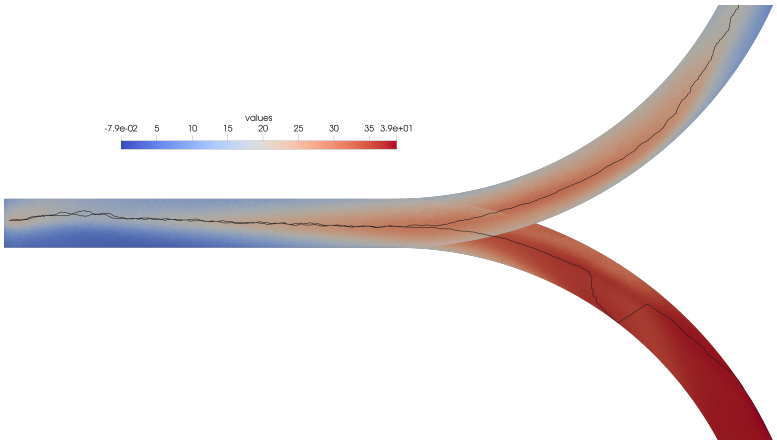


Figure 4.9: Combined track position values on both left and right curve tracks

the position values displayed in Figure 4.8 indicate that this trajectory following manoeuvre is not what the agent has learned. To encourage better generalisability, we will expand the agent observation by including additional features.

4.3.1 Ray casting environment

When we consider the variables in our baseline environment, as described in Tables 4.1 and 4.2, we see that, apart from the track position variable, we have now information about the upcoming track geometry. The agent needs to learn the track geometry using the variables x, y, s, which describe the vehicles absolute position. The assumption that the distances to the track boundary will provide useful information to the agent. While the state variable ν represents the vehicles distance from the centre line, we do not measure the vehicles distance in any direction, other than orthogonal to the centre line. To include this information, we measure the vehicles distance to the track boundary at N_{rays} equidistant angles

$$\omega_i \in \left[-\frac{\pi}{2}, \frac{3\pi}{2}\right].$$

This distance measurement is performed by casting rays

$$r_i = \left\{ \begin{pmatrix} p_x \\ p_y \end{pmatrix} + t \begin{pmatrix} \cos(\psi + \omega_i) \\ \sin(\psi + \omega_i) \end{pmatrix} : t \geq 0 \right\} \tag{4.3.1}$$

and checking collisions with the track boundary. We denote the boundary intersection point that is closest to the vehicle by $(b_x, b_y)^T \in \overline{\mathbb{R}}^2$, where $(b_x, b_y)^T = (\infty, \infty)^T$ if no intersection exists. Using the boundary $(b_x, b_y)^T$ point, the distance along ray r_i is given by $d_i = \min\{100, \|(b_x, b_y)^T - (p_x, p_y)^T\|_2\}$. The ray length is limited to 100, in order to avoid problems caused by large differences in the magnitude of observation variables. We can see a visual representation of such an observation in Figure 4.10. by

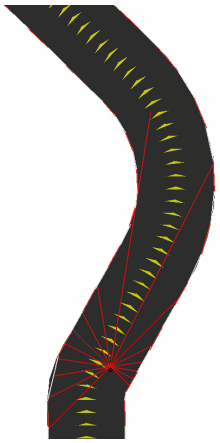
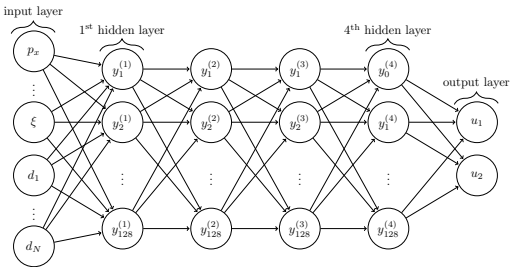
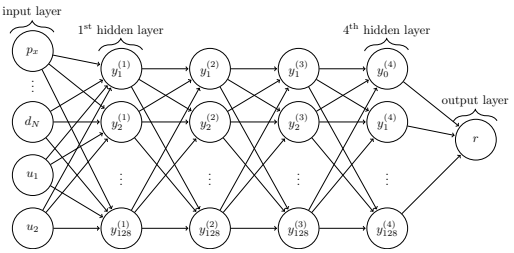


Figure 4.10: Representation of the ray casting observation

including these measured distances $\{d_i : i = 1, \ldots, N_{\text{rays}}\}$ we have modified the shape of our observation space, and thus also changed the shape of our neural network. To keep the changes of our model minimal, we only change the first network layer to accommodate the additional N_{rays} inputs. We can thus see the resulting actor and critic network layouts can be seen in Figure 4.11. To commence our discussion of the rays based network we first compare the training performance of a rays-environment agent, which we see in Figure 4.12a, to a base-environment agent on the L-shape test track from Figure 4.7a. Comparing Figures 4.12a and 4.7a, we see that the agent

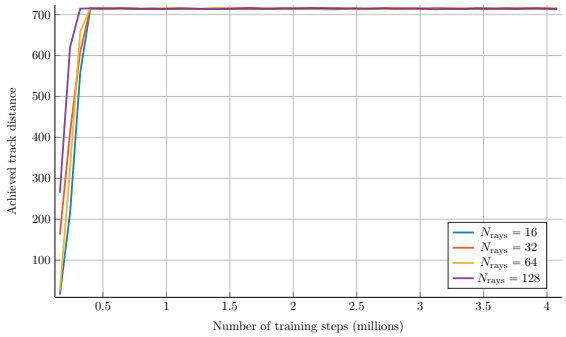


(a) Actor network layout

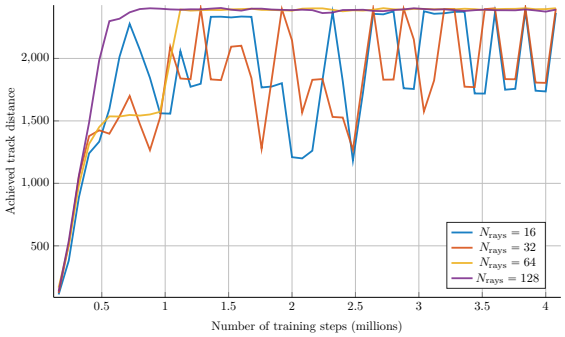


(b) Critic network layout

Figure 4.11: PPO agent network layouts for training on the ray casting environment



(a) PPO model performance, on the example track in Figure 3.3 using the check-point reward function and ray observations environment.



(b) PPO model performance using ray distance observations, trained on the example track in figure 4.5b using the check- point reward function (12).

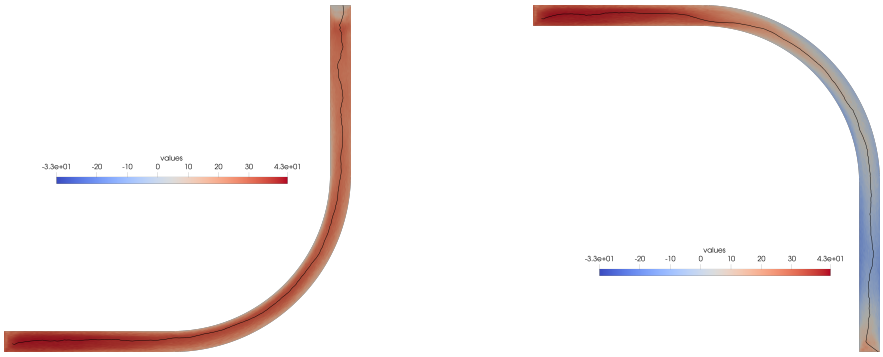
Figure 4.12: PPO model performance using ray distance observations using the checkpoint reward function (12)

was able to learn quicker and more reliably with the inclusion of ray observations. The question arises whether agents in the ray observation environment are able to complete longer and more challenging tracks, such as the "Double-S" track, displayed in Figure 4.5.

In our baseline environment, the agent only sees information about the current track via the global position variables x, y, s and the local variables ν, ξ as described in Tables 4.1 and 4.2. For successful vehicle control the agents needs to learn properties of the upcoming track using the global position variables x, y, s. Consequently, we cannot expect our trained agents to generalise across different race tracks. In contrast to this, the distance measurements d_i of the ray-casting environment provide agent with information about the upcoming track, independent of global position variables. Based on this, we can hope to train models which are able to generalise across different race tracks.

To address the question of performance on more complex race tracks we can see the model performance on the longer "Double-S" track in Figure 4.12b. From Figure 4.12b we see that, while the model performance is unstable, the model is able to finish more challenging race tracks, given sufficient training time. Figures 4.12a and 4.12b also show increased performance with increasing number of rays.

To examine how well an agent, trained on ray-casting distance observations, is able to generalise we now evaluate the model performance. Similar to before, we evaluate how a PPO agent, now trained on the rays environment, values track positions. We note again the significant difference in state evaluations between the two track depicted in the Figures 4.13a and 4.13b. However, in contrast to the previous environment the trajectory of highest valued states follows the track curvature on both tracks.



(a) Track position values on left curve track

(b) Track position values on right curve track

Figure 4.13: Track position values predicted by an agent trained with $N_{\text{rays}} = 32$.

4.3.2 Environment considerations

In the previous section, we successfully expanded the observation space of a racing environment by including ray projection observations. This resulted in longer running trajectories. For long trajectories, we observe an accumulation of errors in the lateral track position variable $\nu(\cdot)$. Let $\mathbf{x}_k$ be the current state and $\mathbf{u}_k$ be the action chosen by our neural network agent and learning algorithm. We evaluate the quadrature error error of our explicit method by

$$\mathbf{x}(t_{k+1}) - \mathbf{x}_{k+1} = \int_{t_k}^{t_{k+1}} \mathbf{f}\left(\mathbf{x}(t), \mathbf{u}_k\right)\, dt - \mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right). \tag{4.3.2}$$

For the lateral track position, we obtain

$$\nu(t_{k+1}) - \nu_{k+1} = \int_{t_k}^{t_{k+1}} x'(t)\sin(\xi(t)) + y'(t)\cos(\xi(t))\, dt - \nu'(t_k).$$

To examine the accumulation of quadrature errors, we let $\mathbf{x}(\cdot)$ be the state function obtained under exact quadrature and $\mathbf{x}_k$ be our approximates at time t_k such that

$$\mathbf{x}(t_k) = \mathbf{x}_k + \epsilon_{\mathbf{x},k},$$

with the error term $\epsilon_{\mathbf{x},k}$. We thus have

$$\begin{aligned}\nu(t_k) - \nu_k &= \nu(t_{k-1}) + \int_{t_0}^{t_k} \nu'(t)\, dt - (\nu_{k-1} + \Delta t_{k-1}\nu'(\mathbf{x}_{k-1}, \mathbf{u}_{k-1})) \\ &= \epsilon_{\nu,k-1} + \int_{t_0}^{t_k} \nu'(t) - \nu'(t_{k-1})\, dt.\end{aligned}$$

At each $t \in (t_{k-1}, t_k)$ we can select a $\tau(t)$, such that

$$\nu'(t) - \nu'(t_{k-1}) = (t - t_{k-1})\nu''(\tau(t)).$$

With this, we can bound the truncation error

$$\begin{aligned}|\nu(t_k) - \nu_k| &= |\epsilon_{\nu,k-1} + \Delta t_{k-1} \int_{t_{k-1}}^{t_k} (t - t_{k-1})\nu''(\tau(t))\, dt| \\ &\leq \epsilon_{\nu,k-1} + \frac{1}{2}\Delta t_{k-1}^2 \|\nu''(t)\|_{L^\infty(t_{k-1},\tau)}.\end{aligned} \tag{4.3.3}$$

For bounded curvature tracks, dynamic model (2.3.22) satisfies a local Lipshitz condition with the Lipshitz constant L. By appealing to the discrete Gronwall inequality [39, 26], we find the error accumulation for the explicit Euler time-stepping scheme applied to ordinary differential equations

$$\|\epsilon_{\nu,k}\| \leq e^{L(t_k - t_0)} \left[\|\epsilon_{\mathbf{x},0}\| + \frac{1}{2}(t_k - t_0) \max_k \Delta_{t_k} \max_{t \in (t_0, t_n)} \|\mathbf{x}''(t)\| \right]. \tag{4.3.4}$$

One option to reduce this error is to subdivide the time step, however this will reduce the speed at which we may generate observations for our learning environment. What is also important is a difference in the error accumulation for different state components. From the model description in Equation (2.3.22) we can see that the differential equations for components, such as x, y, ψ, ν, ξ, satisfy a global Lipshitz condition as long as the track curvature is bounded. In contrast to this the Lipshitz condition for the v_x, v_y, ω, s requires additional bounds on the state variables.

This difference in error accumulation may lead to a situation where a simulated vehicle position $(x(t), y(t))$ is located within the race circuits geometry, while the track condition $|\nu(t)| \leq \frac{W}{2}$ is not satisfied.

In the following we present an alternative track-geometry based method for bound checking. Instead of using the track description presented in Section 2.3.1, we discretised the track using quadrilateral elements as we can see in Figure 4.14a. Using the quadrilateral track approximation, we can check the track condition by testing if the vehicle position $(x(t), y(t))$ is located within one of the quadrilateral pieces shown, in Figure 4.14a. Once we have found the quadrilateral our vehicle is contained in, we can reconstruct approximations to ν, ξ using an orthogonal projection onto the centre-line. To simplify the centre-line projection we project onto the centre-line of a given quadrilateral.

During experiments from the previous sections, we find that vehicles sometimes show undesired behaviour, such as spinning or driving backwards. To prevent this undesired behaviour without introducing penalty terms, we stop the environment early. Stopping the environment simulation early in the case of undesired behaviour prevents the agent from collection future rewards according to the Check-Point rewards Algorithm 12. We detect a spinning vehicle if $|\omega(t)| > \text{TOL}_\omega$. Using the quadrilateral track approximation we can also allow the vehicle to complete multiple laps. Driving multiple laps increases the impact of reward signal gained later into the race and thus favours more consistent driving behaviour. Instead of using the start and finish line of a track to detect if an environment simulation has terminated, we limit the total runtime of our experiments to $T_{\max}$. Let $Q_1, \ldots, Q_N$ be the quadrilaterals in our track approximation, such that we can write an improved version of the terminal state check in the form of Algorithm 13.

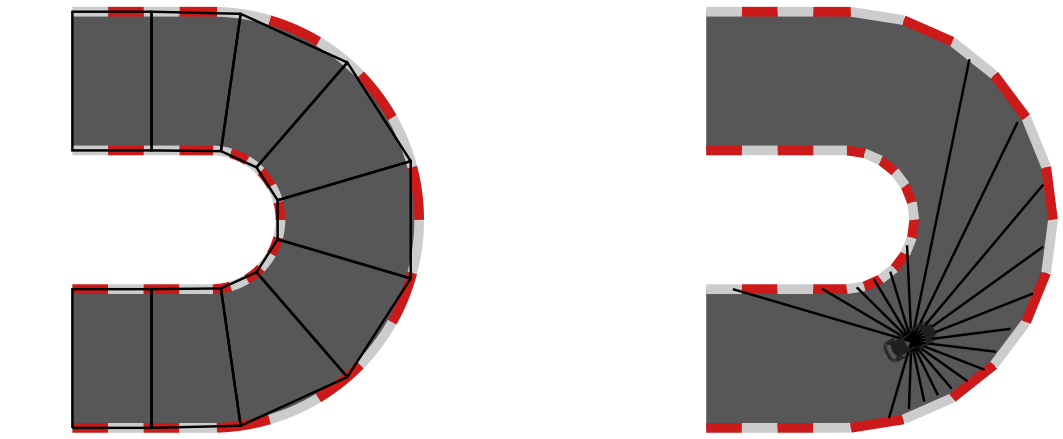
Algorithm 13 Check terminal state II

Require: $\mathbf{x}(t)$

$i \leftarrow -1$
$\mathbf{p} \leftarrow (x(t), y(t))^T$
for $j = 1, \ldots, n$ **do**
    if $\mathbf{p} \in Q_i$ **then**
        $i \leftarrow j$
        break ▷ Leave the surrounding while loop
    end if
end for
if $i < 0$ **then** ▷ Vehicle is not on track.
    Return **true**.
end if
if $|\omega(t)| > 2\pi$ **then** ▷ Lost control
    Return **true**.
end if
if $x'(t)^2 + y'(t)^2 < 1.0$ **then** ▷ Vehicle is driving too slowly.
    Return **true**.
end if
if $t > T_{\max}$ **then** ▷ Maximal time exceeded
    Return true.
end if ▷ Terminal state is not reached
Return **false**.

4.3.3 Visual environment

In Section 4.3.1, we extended the observation space of environments by including distance measurements to the track boundary. From Figure 4.10, we see that, while these measurements let us determine the radius of the current curve, we can not anticipate upcoming curves. We can see this more clearly in Figure 4.14b, which shows the track boundary visible to the agent. In Figure 4.14b, we can see that no



(a) Quadrilateral track approximation (b) Example ray projection view

Figure 4.14: Quadrilateral track discretisation and ray projection

rays intersect with the inner 11 boundary elements and similarly no rays intersect with the external 6 elements. Each boundary elements appears in Figure 4.14b as either a red or light-gray box at the track boundary. We may increase the number of ray measurements in this environment to increase the information we capture. As the ray measurements are taken in straight lines, we can not capture measurements in the highlighted area in Figure 4.14b.

To increase the information available to agents we introduce a new environment, called the *visual environment*. An agent training on this environment observes the environment as grey scale pixel data. For this, we implement a top-down view for our environments using the $2D$-graphics library *PyGame* [87]. The visualisation of a ray distance measurements we saw in Figure 4.10 from the previous Section was generated

using this visual environment. The visual environment takes a resolution p_y as a parameter and creates a virtual screen of resolution $\left(\frac{16}{9}p_y, p_y\right)$. We generate images by observing an area of $200m$ width around the vehicle. The vehicle position remains fixed on-screen at the pixel position $\left(\frac{16}{18}p_y, \frac{5}{6}p_y\right)$. It is the standard in computer imaging applications to label the top-left corner of an image with position $(0, 0)$. Towards the right of an image the x coordinate grows, and similarly the y coordinate grows towards the bottom of the image. With this convention, the vehicle is located close to the middle of the lower edge of our virtual display.

This far we have discussed agents that work on vector based input data. It is natural to use the spacial structure present in $2D$ image data to aid in the extraction of useful information from image data. A common approach for this is to use convolutional neural network layers. A $2D$ convolutional layer acts on an input image of dimension $H \times W$ using the convolution operator. In convolutional kernels, we define both a kernel and a bias term. Let the dimension of the kernel be $h_k \times w_k$. Usually, the kernel size is much smaller than the input image size. In our environment, typical image sizes range from 107×60 pixels to 213×120 pixels. The kernel applied to such an images is relatively small at around 5×5 pixels. A padding parameter $p \geq 0$ controls how we perform convolution operations at the edge of an image by adding p virtual 0 around the edges of our input image. Especially for larger kernels, we often want to omit convolution operations, with high spatial correlation, to increase computational efficiency and reduce the output image dimension. The stride parameter $s \in \mathbb{N}_{\geq 1}$ controls how many pixels we move a kernel after each application. With this, we obtain the formula

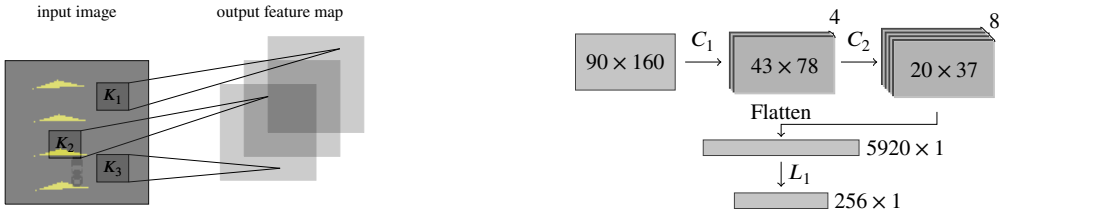
$$\begin{pmatrix} H_{\text{out}} \\ W_{\text{out}} \end{pmatrix} = \left\lfloor \frac{1}{s} \begin{pmatrix} H_{\text{in}} - h_k + 2p + 1 \\ W_{\text{in}} - w_k + 2p + 1 \end{pmatrix} \right\rfloor \tag{4.3.5}$$

for the output dimension of a convolution operation. We thus apply kernel K to an

image I^{in} and obtain the output pixel values

$$I^{\text{out}}_{i,j} = b_{i,j} + I^{\text{in}}\left[is - p : is - p + h_k, js - p : js - p + w_k\right] \star K, \tag{4.3.6}$$

where $b_{i,j}$ is a bias term which, similar to the kernel values, will be determined during training and $\star$ is the $2D$ convolution operator. Figure 4.15a shows a visualisation for the convolution procedure. In many computer imaging applications, we have more



(a) $2D$ convolution visualisation (b) $2D$ Convolutional feature extraction

Figure 4.15: $2D$ Convolution and feature extraction

than than one information channel per image. For example, we often have three separate colour channels in an image. Thus, we let C^{in} be the number of channels in an image, resulting in the input dimension $C^{\text{in}} \times H^{\text{in}} \times W^{\text{in}}$. Multiple information channels are important in our application as well, even though we only process grey scale images. This is because we often wish to apply multiple kernels to the same input image. Doing this, we obtain n output channels by applying the n kernels $K_1, \ldots, K_n$ according to

$$I^{\text{out}}_{i,j} = b_{i,j} + \sum_{i=1}^{n} I^{\text{in}}\left[is - p : is - p + h_k, js - p : js - p + w_k\right] \star K_i. \tag{4.3.7}$$

We use the convolution operation from Equation (4.3.7) to extract features from visual information. For feature extraction, we use a two convolutional layers, C_1, C_2, both with kernels of size 5×5 and a stride and padding parameters $s = 2, p = 0$. We can see precise parameter description in Table 4.5. Layer C_2 outputs a total of 8 feature layers. Due to the stride parameter $s = 2$, the size of these feature layers

Table 4.5: Convolutional neural network parameters

Layer	Kernel size	Input channels	Output channels	Stride s	Padding p
C_1	5×5	1	4	2	0
C_2	5×5	4	8	2	0

is significantly reduced compared to the input layer. We further process the output of C_2 by flattening the resulting values and applying a single linear output map. Figure 4.15b shows this feature extraction process. Table 4.6 lists the number of parameters for each network layer, as well as the and computational cost for applying each convolution layer. From Equation (4.3.7), we see that the computational cost for applying a convolutional layer depends on the number of input channels N_{in}, the number of output channels N_{out}, the size of the output image $D_{\text{out}} = (H_{\text{out}}, W_{\text{out}})$, as well as the kernel size. Applying a kernel K of size $h_k \times w_k$ requires $h_k \cdot w_k$ multiplications and $h_k \cdot w_k - 1$ additions. This lets us obtain that the number of required floating point operations to apply is

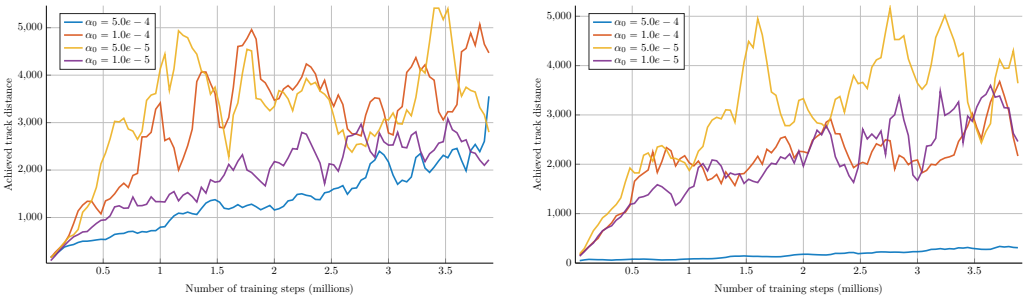
$$\#\text{FLOPS}(K, N_{\text{in}}, N_{\text{out}}, D_{\text{out}}) = (2h_k \cdot w_k - 1) \cdot (N_{\text{in}} N_{\text{out}} H_{\text{out}} W_{\text{out}}) \,.$$

We now evaluate our model on the visual learning environment. Figure 4.16 shows

Table 4.6: Network sizes

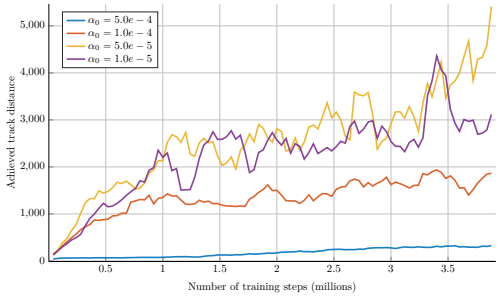
Layer	# Parameters	# Operations
C_1	104	7.0×10^5
C_2	808	1.1×10^6
L_1	$557,312$	3.0×10^6
Total	$561,550$	4.8×10^6

the mean achieved distances on the Bahrain track 4.2a at different visual resolutions and initial learning rates. We saved the parameters of each network every $20,000$ training steps and evaluated the performance by averaging 8 runs at each network state. The learning rates follow the schedule from Equation (4.2.18) with $\beta = 10^{-6}$. From Figures 4.16a, 4.16b, 4.16c we see see that the visual model, shown in Fig-



(a) Achieved distances at 107×60 resolution

(b) Achieved distances at 160×90 resolution



(c) Achieved distances at 213×120 resolution

Figure 4.16: Achieved distances on the Bahrain track 4.2a using the visual model

ure 4.15b, was able to learn better on lower resolutions. This result is surprising, as we may expect that higher resolutions are able to represent track features better than lower resolutions. In the following we will call the observation that increased visual fidelity, measured in terms of image resolution, decreases the training performance the *resolution effect*. Independent of the environment resolution we find that learning rates $\alpha \in [5 \times 10^{-5}, 1 \times 10^{-4}]$ were able to achieve the best learning performance.

One potential explanation is that the increased feature complexity of higher resolution images causes us to take destructively large policy steps. Another potential explanation is that, as image resolution increases, the physical area, corresponding to a convolution kernel of constant size $k \times k$, decreases. As the physical area covered by a convolution kernel decreases, so does the ability of our neural network to detect track features.

We examine the impact of increased resolutions on policy steps by comparing

differences between two consecutive agent policies π_0, π_1, where we assume that π_0 is the current policy. As such, samples in our roll-out buffer are generated according to π_0. A common distance measure for probability distributions is the *Kullback–Leibler* divergence (KL). The policies π_0, π_1 are not probability distributions themselves, but they map the state space to probability distributions on the actions space

$$\pi_i : \mathcal{X} \to M_{\geq 0}^1(\mathcal{A}) \quad \text{for } i = 1, 2.$$

To compute the KL divergence between π_0, π_1, we consider the total action distribution while following policy π_0. For this, we let $\kappa_{\pi_0} \in M_{\geq 0}^1(\mathcal{X})$ be the state probability distribution while following policy π_0. With this we can define the total action probability distribution

$$\tilde{\pi}_0(\mathbf{a}) := \int_{\mathbf{x} \in \mathcal{X}} \pi_0(\mathbf{a}|\mathbf{x})\, d\kappa_{\pi_0}(\mathbf{x}), \quad \tilde{\pi}_1(\mathbf{a}) := \int_{\mathbf{x} \in \mathcal{X}} \pi_1(\mathbf{a}|\mathbf{x})\, d\kappa_{\pi_0}(\mathbf{x}).$$

Both $\tilde{\pi}_0$ and $\tilde{\pi}_1$ assume that states are sampled following the state distribution κ_{π_0}. The KL divergence is defined as

$$\mathrm{KL}[\tilde{\pi}_0, \tilde{\pi}_1] := \int_{\mathbf{a} \in \mathcal{A}} \pi_0(\mathbf{a}) \log\left(\frac{\tilde{\pi}_0(\mathbf{a})}{\tilde{\pi}_1(\mathbf{a})}\right) = \mathbb{E}_{\mathbf{a} \sim \tilde{\pi}_0}\left[\log\left(\frac{\tilde{\pi}_0(\mathbf{a})}{\tilde{\pi}_1(\mathbf{a})}\right)\right]. \tag{4.3.8}$$

For computational efficiency we cannot compute the integral over the action space directly. The most common approach to compute the KL divergence is to replace the expected value with Monte-Carlo (MC) samples from our roll-out buffer B. The state and action samples will naturally follow the previous policy π_0. A direct approach for estimating $\mathrm{KL}[\pi_0, \pi_1]$ is

$$\hat{\mathrm{KL}}_1\left[\tilde{\pi}_0, \tilde{\pi}_1\right] := \hat{\mathbb{E}}_{\mathbf{a}_i \in B}\left(\log(\tilde{\pi}_0(\mathbf{a}_i)) - \log(\tilde{\pi}_1(\mathbf{x})),\right), \tag{4.3.9}$$

$$= \frac{1}{|B|} \sum_{\mathbf{x}_i, \mathbf{a}_i \in B} \log(\tilde{\pi}_0(\mathbf{a}_i)) - \log(\tilde{\pi}_1(\mathbf{a}_i)), \tag{4.3.10}$$

where $|B|$ corresponds to the number of samples in our roll-out buffer. We first show that $\hat{\mathrm{KL}}_1$ is unbiased by computing

$$\begin{aligned}
\mathbb{E}\left[\hat{\mathrm{KL}}_1[\tilde{\pi}_0, \tilde{\pi}_1]\right] &= \frac{1}{|B|}\mathbb{E}\left[\sum_{\mathbf{a}_i \in B} \log(\tilde{\pi}_0(\mathbf{a}_i)) - \log(\tilde{\pi}_1(\mathbf{a}_i))\right], \\
&= \frac{1}{|B|}\sum_{i=1}^{|B|} \mathbb{E}\left[\tilde{\pi}_0(\mathbf{a}_i)\left(\log(\tilde{\pi}_0(\mathbf{a}_i)) - \log(\tilde{\pi}_1(\mathbf{a}_i))\right)\right], \\
&= \frac{1}{|B|}\sum_{i=1}^{|B|} \mathrm{KL}\left[\tilde{\pi}_0, \tilde{\pi}_1\right] = \mathrm{KL}\left[\tilde{\pi}_0, \tilde{\pi}_1\right].
\end{aligned}$$

While $\hat{\mathrm{KL}}_1$ is unbiased, its variance is often high and $\log\left(\tilde{\pi}_0(\mathbf{a})\right) - \log\left(\tilde{\pi}_1(\mathbf{a})\right)$ can be negative, while $\mathrm{KL}[\tilde{\pi}_0, \tilde{\pi}_1]$ is always non-negative. We denote the relative total action probability by $r(\mathbf{a}) = \tilde{\pi}_1(\mathbf{a})\tilde{\pi_0}(\mathbf{a})^{-1}$, and find that

$$\begin{aligned}
\mathbb{E}_{\mathbf{a}\sim\tilde{\pi}_0}\left[r(\mathbf{a})\right] &= \int_{\mathbf{a}\in\mathcal{A}} \frac{\tilde{\pi}_1(\mathbf{a})}{\tilde{\pi_0}(\mathbf{a})}\tilde{\pi_0}(\mathbf{a})\, d\mathbf{a} = \int_{\mathbf{a}\in\mathcal{A}} \tilde{\pi}_1(\mathbf{a})\, d\mathbf{a} \\
&= \int_{\mathbf{x}\in\mathcal{X}} \kappa_{\pi_0}(\mathbf{x}) \underbrace{\int_{\mathbf{a}\in\mathcal{A}} \pi_1(\mathbf{a}|\mathbf{x})\, d\mathbf{a}}_{=1 \quad \forall \mathbf{x}}\, d\mathbf{x} \\
&= \int_{\mathbf{x}\in\mathcal{X}} \kappa_{\pi_0}(\mathbf{x})\, d\mathbf{x} = 1.
\end{aligned}$$

As log is concave we find that $(x-1) \geq \log(x) \quad \forall x \in \mathbb{R}$, and by the previous computation we have

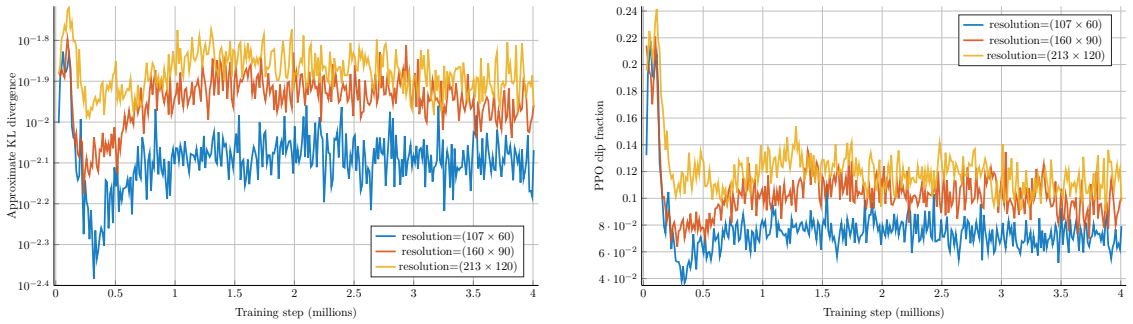
$$\mathbb{E}_{\mathbf{a}\sim\tilde{\pi}_0}\left[r(\mathbf{a}) - 1 - \log\left(r(\mathbf{a})\right)\right] = \mathbb{E}_{\mathbf{a}\sim\tilde{\pi}_0}\left[-\log\left(\frac{\tilde{\pi}_1(\mathbf{a})}{\tilde{\pi}_0(\mathbf{a})}\right)\right] + \underbrace{\mathbb{E}_{\mathbf{a}\sim\tilde{\pi}_0}\left[r(\mathbf{a}) - 1\right]}_{=0}.$$

We therefore use the KL estimate

$$\hat{\mathrm{KL}}_2[\pi_0, \pi_1] := \hat{\mathbb{E}}_{\mathbf{a}_i \in B}\left[r(\mathbf{a}_i) - 1 - \log\left(r(\mathbf{a}_i)\right)\right]. \tag{4.3.11}$$

If, during training, the KL divergence is to large, the risk for destructively large

policy steps increases. On the other hand, a small KL divergence can indicate that we can choose a higher learning rates to speed up the training process. Figure 4.17a shows KL divergence estimator $\hat{\mathrm{KL}}_2$ over the course of the training process at the initial learning rate $\alpha = 5 \times 10^{-5}$. We observe from Figure 4.17a that larger visual



(a) Estimated KL divergence during training on the visual environment

(b) Clip fraction during training of the visual environments

Figure 4.17: KL divergence and clip fraction on visual environment

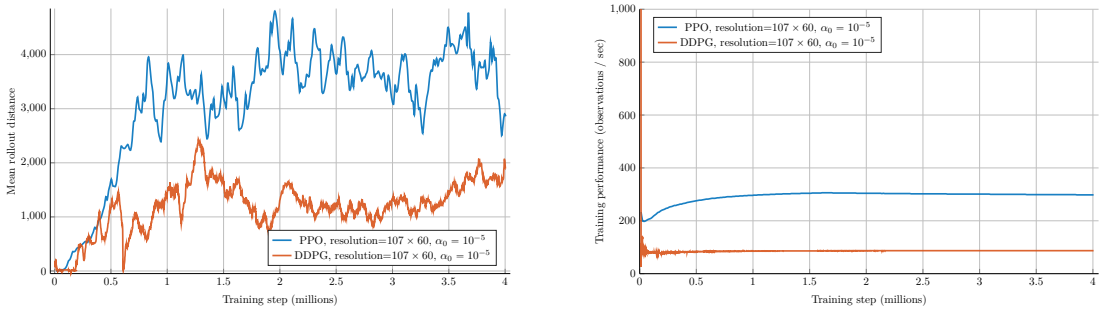
resolutions tend to lead to higher KL divergences, thus requiring smaller policy steps. During PPO learning large policy steps are discouraged by artificially reducing the potential advantage gained from changes in action probability that exceed a given threshold. From the description of PPO in Section 4.1.3, we know that advantage function contributions are reduced if the relative action probability

$$r(\theta', \theta) = \frac{\pi_{\theta'}(\mathbf{a}_i|\mathbf{x}_i)}{\pi_{\theta'}(\mathbf{a}_i|\mathbf{x}_i)}$$

is larger than 1.2 or smaller than 0.8. Figure 4.17b shows the clip fraction during training of our visual model on the Bahrain track under the initial learning rate $\alpha = 5 \times 10^{-5}$ and varying environment resolutions. The results shown in Figures 4.17a and 4.17b show successive increases in the estimated KL divergence, each time we increase the observation resolution.

4.3.4 Comparison to Off-Policy algorithms

We have limited our considerations to the proximal policy algorithm (PPO). PPO operates in an on-policy manner, as such only observations and reward signals collected on the current policy are used during training. On-policy algorithms often exhibit more stable training behaviour, as no samples from previous policies are considered. In contrast to this, off-policy algorithms, such as the Deep Deterministic Policy Gradient Algorithm (DDPG) [58], often show higher sample efficiency at the cost of less stable learning. Figure 4.18 shows a comparison of PPO and DDPG learning on the visual environment. In Figure 4.18 both the PPO and DDPG agent were trained on



(a) Track distance comparison of PPO and DDPG

(b) Training performance comparison of PPO and DDPG

Figure 4.18: Comparison of PPO and DDPG on the visual environment

a visual environment with 107×60 pixel resolution with an initial learning rate of $\alpha_0 = 10^{-5}$ for a total of 4×10^6 observations. Overall we see a positive learning effect for both algorithms, however PPO is able to achieve higher track distances and faster training at approximately 300 observations per second, compared to approximately 87 in the case of DDPG.

4.3.5 Visual environment with frame stacking

The environments presented in Section 4.3.3 do not include all the information of the previous baseline or ray-casting environments. Information about vehicle motion

is included in the environment and it can not be reconstructed from a single frame. A common method of including temporal information, such as velocities, is to include previous frames in the observation space. The network observation will thus not only depend on the current state, but also a short-term history of previous states. We pass multiple frames to the neural network agents by treating them as different channels. Therefore, the input of C_1 has multiple channels, corresponding to the number of frames in our observation. We first compare the impact of learning rates on the visual environment with frame stacking. Figure 4.19 shows the achieved track distances for an agent trained on the Bahrain track, shown in Figure 4.2a, with four consecutive frames in the observation. From Figure 4.19, we see that we can achieve positive train-

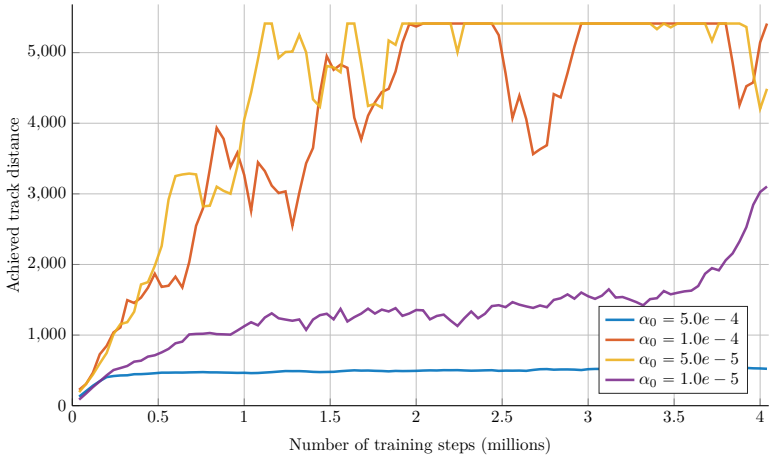
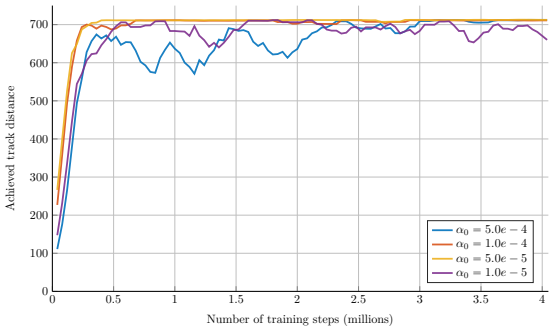


Figure 4.19: Achieved distances using four stacked frames on the Bahrain track

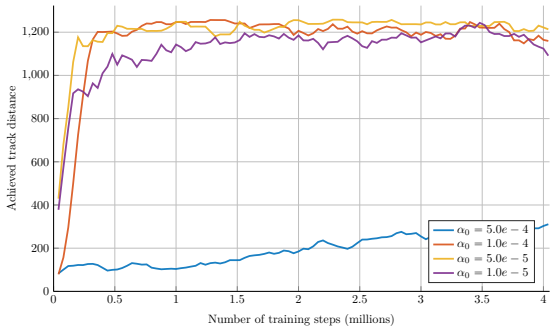
ing results for with all learning rates $\alpha \in \{5 \times 10^{-4}, 1 \times 10^{-4}, 5 \times 10^{-5}, 1 \times 10^{-5}\}$, however $\alpha = 5 \times 10^{-5}$ is able to cover the largest track mean distance in the given simulation time.

We now consider how well the trained agents are able to generalise and evaluate how agents perform on unseen test tracks. Similar to the single frame visual environment, observations only depend on the vehicle position and velocity relative to the local track geometry. Track dependent variables, such as the global track distance s, or the global vehicle position x, y do not impact the observation. We expect

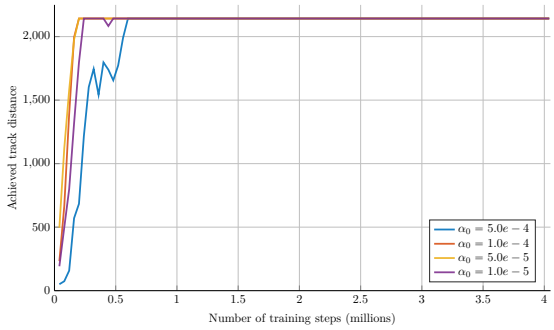
that the local nature of our observation has a positive impact on the ability of our agents to generalise to unseen tracks. In Figures 4.20a, 4.20b, and 4.20c, we see the achieved track distances of agents, which were trained on the Bahrain, evaluated on the "L"-shape, "Z"-shape and "Double-S"-shape test tracks respectively. We see



(a) Distances on L-shape track.



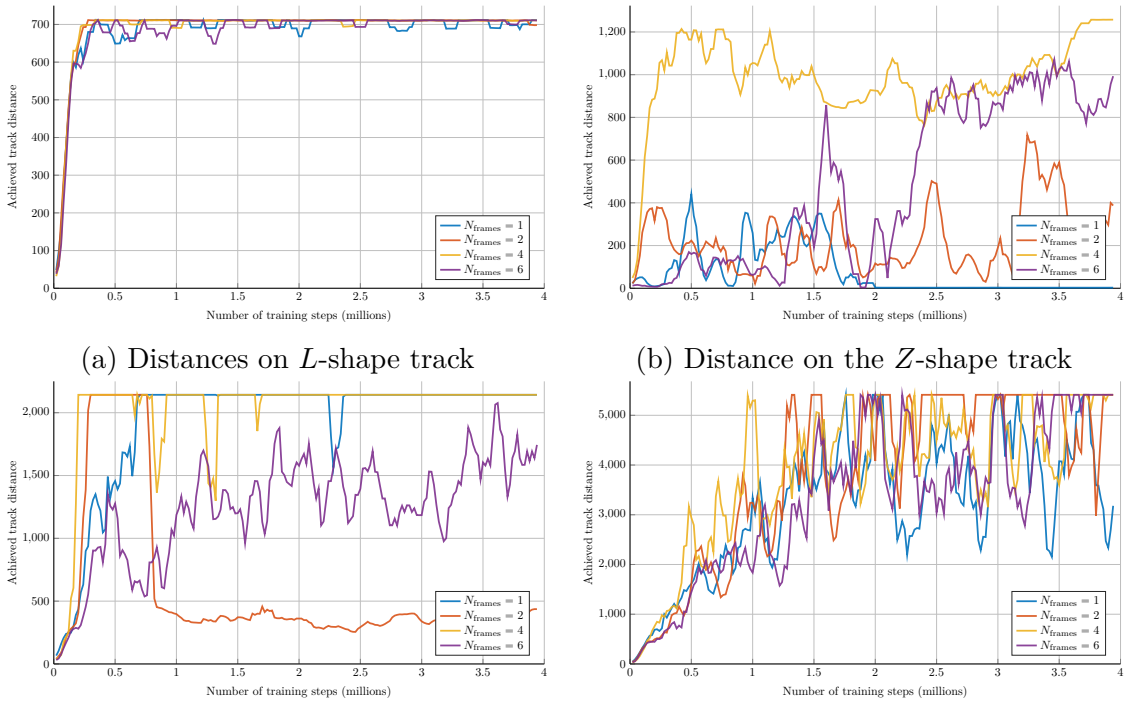
(b) Distance on the Z-shape track.



(c) Distance on the Double-S-shape track.

Figure 4.20: Achieved distances on unseen track using four stacked frames.

from Figure 4.20a that all agents were able to consistently solve the simple L-shape test track. When we analyse the achieved distances on the Z-shape and Double-S-shape tracks, we see that lower learning rates are able to generalise better to unseen race tracks. To see the impact of frame-stacking, we now compare the achieved track distances on multiple tracks using different numbers of stacked frames for our observation. While Figures 4.21a and 4.21c show little impact of the frame stack on the achieved distance, Figure 4.21b shows a negative impact on the ability to generalise as the number of stacked frames increases.



(a) Distances on L-shape track

(b) Distance on the Z-shape track

(c) Distance on the Double-S-shape track

(d) Distance on the Bahrain circuit

Figure 4.21: Achieved distances on unseen track using varying number of stacked frames and an initial learning rate $\alpha_0 = 1 \times 10^{-5}$.

4.3.6 Combined environment

In Sections 4.3.1 and 4.3.3 we discussed two extensions of the baseline environment from Section 4.2. Comparing Figures 4.7b, 4.12b, and 4.21c, shows that both extended environments are able to outperform the baseline environment. While we can capture information beyond the drivers field of view in the visual environment, we need to recover boundary distances and velocities using one or multiple frames.

In this Section we combine all the information contained the three environments, resulting on the so called *combined environment*. To make use of both vector-form and pixel-form data, we separate the observation and apply the corresponding neural network models from Section 4.3.1 and 4.3.3 respectively. The latent feature outputs from both models is in vector form. We then concatenate the feature vectors from both models, and apply a final fully connected layer. We see a visualisation of this combined model in Figure 4.22 where we used the visual layers defined in Table 4.7.

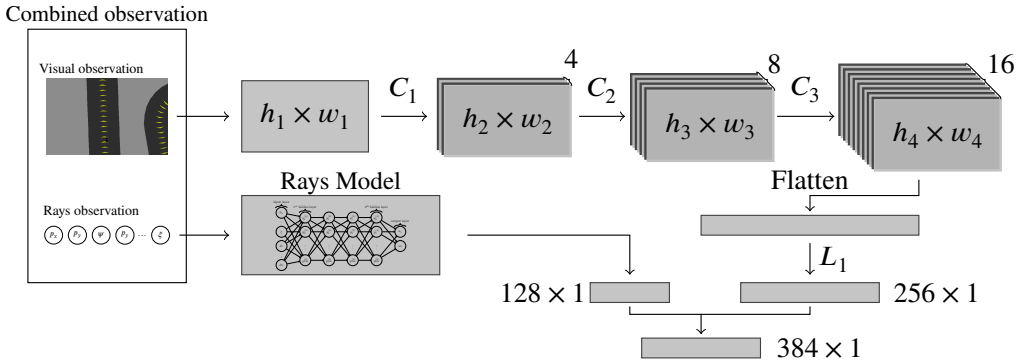


Figure 4.22: Combined environment feature extraction

Table 4.7: Convolutional neural network parameters

Layer	Kernel size	Input channels	Output channels	Stride s	Padding p
C_1	5×5	1	4	2	0
C_2	5×5	4	8	2	0
C_3	5×5	8	16	2	0

We trained this combined model on a selection of eight Formula One tracks by selecting a new track at random each time we reset the environments. The eight training tracks we used are (i) Bahrain, (ii) Baku, (iii) Imolah, (iv) Jeddah, (v) Melbourne, (vi) Miami, (vii) Montreal, and (viii) Silverstone. To make full use of our largest and final model we train it for 25×10^6 steps at an initial learning rate $\alpha_0 = 3 \times 10^{-4}$ with a decay factor $\beta = 5 \times 10^{-7}$. We see the achieved track distances after 1×10^6, 10×10^6, and 25×10^6 environment steps in Figure 4.23. Figure 4.23 includes the maximal achieved distance throughout the training process in cyan. We observe that, while the training process is not monotone, we were able to finish most tracks successfully at some point during the optimisation process. As evaluating such a trajectory requires significantly less computational effort, compared to training, we assume that one can select the optimal model for each track. The model we select is thus not necessarily the same for different tracks. Of the 20 circuits in our test set we were able to find valid trajectories for 18, while only training on eight, showing the generalisability of our model. The most challenging circuits in our test set were the Monaco and

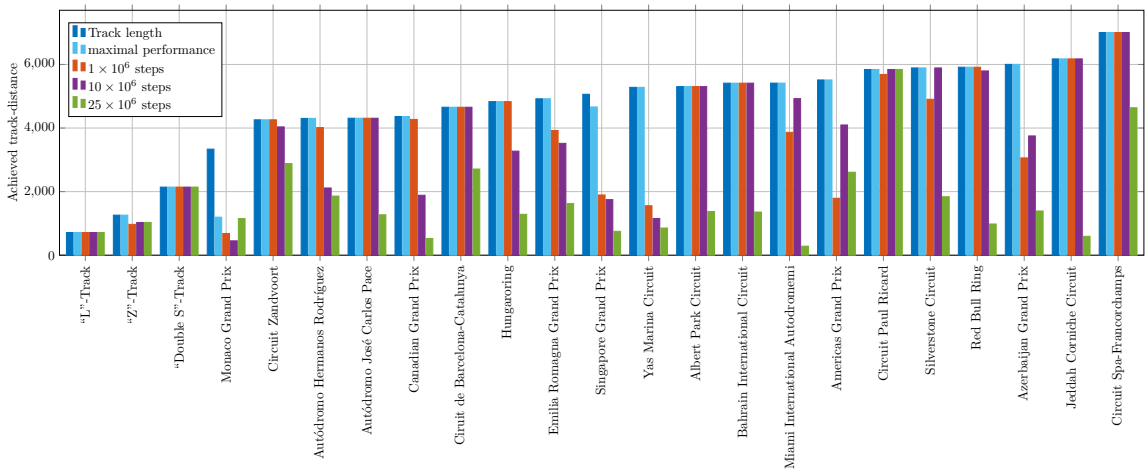


Figure 4.23: Training results for the combined environment

Singapore Grand Prix circuits.

4.4 Strategy extraction

In the previous section we presented a series of learning environments and explored useful hyper-parameter choices. For the base-line environment we experimented with reward signal sparsity and with the learning rate. On the extended environments, such as the ray-casting environment and the visual as well as combined environment we explored the choices for number of rays and resolutions of visual environment representations. In our final model we were able to train a model that is able to perform well complex race circuits, such as the tracks from the 2022 Formula One calendar and even generalise to previously unseen circuits.

Due to this ability to generalise to unseen circuits we are able to train the neural network agent independent of other optimisation methods. Using these agents we can then simulate trajectories around a circuit at low cost. In this section we discuss the extraction of trajectories and how we may use them to inform other methods, such as our collocation solver from Chapter 3. Extracting vehicle trajectories from pre-trained probabilistic policies is not a straightforward task. Complexities arise due to non-linearities in the dynamics function f. Our agents learn a probabilistic policy function policy function

$$\pi : \mathcal{X} \to M^1(\mathcal{A}).$$

In the stable baselines (SB) project [43] actor critic algorithms, such as PPO, model this distribution $\pi(\mathbf{x}_t)$ using a state dependent diagonal Gaussian distribution. Similar to previous sections we let $\mathbf{x}_k \in \mathcal{X} \subset \mathbb{R}^n$ be the state vector and $\mathbf{u}_k \in \mathcal{A} \subset \mathbb{R}^m$ be the control vector at time $t_k \geq 0$. The action distribution is thus

$$\begin{aligned} \pi(\mathbf{x}_k) &= \overline{\mathbf{u}}_k + N(0, \mathrm{diag}(\sigma_1, \ldots, \sigma_m)), \\ \mathbf{u}_k &\sim \pi(\mathbf{x}_k). \end{aligned} \tag{4.4.1}$$

The action distribution $\pi(\mathbf{x}_k)$ puts non-zeros weight onto actions outside the feasible region $\mathcal{A}$, so that instead of executing $\mathbf{u}_k$ directly, we project onto $\mathcal{A}$ first. The feasible region $\mathcal{A}$ takes the from

$$\mathcal{A} = [u_{1,\text{lower}}, u_{1,\text{upper}}] \times \cdots \times [u_{m,\text{lower}}, u_{m,\text{upper}}]\,. \tag{4.4.2}$$

We thus compute the state trajectory using

$$\mathbf{x}_{k+1} = \mathbf{x}_k + (t_{k+1} - t_k) f(\mathbf{x}_k, P_{\mathcal{A}}\mathbf{u}_k), \tag{4.4.3}$$

such that the orthogonal projection operator $P_{\mathcal{A}}$ takes the form

$$P_{\mathcal{A}}\mathbf{u} = \text{clip}(u_i, u_{i,\text{lower}}, u_{i,\text{upper}})_{i=1,\ldots,m}. \tag{4.4.4}$$

To extract the mean trajectory we need to consider the effects introduced by following the non-linear dynamics function $f : \mathbf{X} \times \mathbf{A} \to \mathbb{R}^n$. As f is non linear the expected value of $f(\mathbf{x}_k, \cdot)$ under $\mathbf{u} \sim \pi(\mathbf{x}_k)$ will in general not be realised at $f\left(\mathbf{x}_k, P_{\mathcal{A}}\mathbb{E}\left[\pi(\mathbf{x}_k)\right]\right)$. We are thus interested in the expected state

$$\mathbb{E}\left[\mathbf{x}_{k+1}\right] = \mathbf{x}_k + (t_{k+1} - t_k)\mathbb{E}_{\mathbf{u}_k \sim \pi(\mathbf{x}_k)}\left[f(\mathbf{x}_k, P_{\mathcal{A}}\mathbf{u}_k)\right]. \tag{4.4.5}$$

We can recover an action $\mathbf{a}_t$ that approximates our expected step by solving the least squares problem

$$\begin{aligned} \min \quad & \left[f(\mathbf{x}_k, \mathbf{u}_k) - \mathbb{E}_{\mathbf{u} \sim \pi(\mathbf{x}_k)}\left[f(\mathbf{x}_k, P_{\mathcal{A}}\mathbf{u})\right]\right]^2, \\ \text{subject to:} \quad & \mathbf{u}_k \in \mathcal{A}. \end{aligned} \tag{4.4.6}$$

To solve problem (4.4.6) we first need to evaluate the expected value

$$\mathbb{E}_{\mathbf{u}\sim\pi(\mathbf{x}_k)}\left[f(\mathbf{x}_k, P_{\mathcal{A}}\mathbf{u})\right]. \tag{4.4.7}$$

One effective way for this evaluation is to sample realisations for $\pi(\mathbf{x}_k)$ and construct a Monte-Carlo approximation. For this let $\mathbf{u}_k^{(1)}, \ldots, \mathbf{u}_k^{(n)}$ be independent samples from $\pi(\mathbf{u})$, and we let $\Sigma_{\mathbf{y}}$ be the covariance matrix of $\mathbf{y}^{(1)} = f(\mathbf{x}, \mathbf{u}_k^{(1)})$, then, by the central limit theorem

$$\frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}, \mathbf{u}_k^{(i)}) \xrightarrow{d} \mathcal{N}\left(\mathbb{E}\left[f(\mathbf{x}_k, P_{\mathcal{A}}\mathbf{u})\right], N^{-1/2}\Sigma_{\mathbf{y}}\right). \tag{4.4.8}$$

In order to limit the variance of our Monte-Carlo estimate we require the covariance matrix $\Sigma_{\mathbf{y}}$. For this let us proceed by noting that

$$\mathbb{E}\left[f(\mathbf{x}, \mathbf{u}_k^{(i)})\right] = \mathbb{E}\left[f(\mathbf{x}, \overline{\mathbf{u}} + \Sigma v)\right], \tag{4.4.9}$$

with $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_m)$ and $\mathbf{v} \sim \mathcal{N}(0, \mathbb{1})$ an m-dimension diagonal Gaussian random variable. Using $\mathbb{E}\left[\Sigma\mathbf{v}\right] = 0$ we proceed by making a second order Taylor expansion of f in $\mathbf{u}$ around $\overline{\mathbf{u}}$. As $f(\mathbf{x}_k, \cdot) : \mathbb{R}^m \to \mathbb{R}^n$ is a vector valued function we use the identification theorem for vector valued functions from [59, p.123]. We make the definitions

$$\begin{aligned}
\mathrm{d}\, f(\mathbf{x}_k, \overline{\mathbf{u}}; \mathbf{z}) &:= \frac{d}{d\mathbf{u}} f(\mathbf{x}_k, \overline{\mathbf{u}})\mathbf{z} \\
\mathrm{d}^2 f(\mathbf{x}_k, \overline{\mathbf{u}}; \mathbf{z}) &:= \mathbf{z}^T \frac{d^2}{d\mathbf{u}^2} f(\mathbf{x}_k, \overline{\mathbf{u}})\mathbf{z} \\
&= (I_n \otimes \mathbf{z}^T)\frac{d}{d\mathbf{u}} \operatorname{vec}\left[\frac{d}{d\mathbf{u}} f(\mathbf{x}_k, \overline{\mathbf{u}})\right]\mathbf{z} \\
&= (I_n \otimes \mathbf{z}^T \otimes \mathbf{z}^T)\frac{d}{d\mathbf{u}} \operatorname{vec}\left[\frac{d}{d\mathbf{u}} f(\mathbf{x}_k, \overline{\mathbf{u}})\right],
\end{aligned} \tag{4.4.10}$$

where $\otimes$ is the Kronecker product of two matrices and vec denotes the vectorisation operator. The vectorisation operator vec transform a matrix $A \in \mathbb{R}^{n_1 \times n_2}$ into a vector

$\mathbf{a} \in \mathbb{R}^{n_1 n_2}$ by stacking the n_2 columns of A on top of each other. Using the Kronecker and vec operations we can write higher order derivatives of f as

$$\begin{aligned} \mathrm{D}^k f(\mathbf{x}_k, \overline{\mathbf{u}}) &= \frac{d}{d\mathbf{u}} \operatorname{vec}\left[\mathrm{D}^{(k-1)} f(\mathbf{x}_k, \overline{\mathbf{u}})\right], \\ \mathrm{d}^k f(\mathbf{x}_k, \overline{\mathbf{u}}; \mathbf{z}) &= \left(I_n \otimes (\mathbf{z}^T)^{\otimes k}\right) D^k f(\mathbf{x}_k, \overline{\mathbf{u}}). \end{aligned} \tag{4.4.11}$$

Equality (4.4.11) follows originally from a reasoning in [23, Section 5.9], here we use it in the notation closely related to [24]. The only difference to in our notation is that we employ the vec operator to avoid applying Kronecker products to derivative operators. Utilizing this notation we can write the expansion

$$f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v}) = f(\mathbf{x}_k, \overline{\mathbf{u}}) + df(\mathbf{x}_k, \overline{\mathbf{u}}; \Sigma\mathbf{v}) + \frac{1}{2} d^2 f(\mathbf{x}_k, \overline{\mathbf{u}}; \Sigma\mathbf{v}) + r(\Sigma\mathbf{v}), \tag{4.4.12}$$

where the remainder term $r : \mathbb{R}^m \to \mathbb{R}^n$ satisfies $\lim_{\mathbf{z}\to 0} \frac{r(\mathbf{z})}{\|\mathbf{z}\|^2} = 0$. Furthermore we have an explicit representation of the remainder term in the form of

$$\begin{aligned} f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v}) = f(\mathbf{x}_k, \overline{\mathbf{u}}) + df(\mathbf{x}_k, \overline{\mathbf{u}}; \Sigma\mathbf{v}) + \frac{1}{2} d^2 f(\mathbf{x}_k, \overline{\mathbf{u}}; \Sigma\mathbf{v}) \\ + \frac{1}{6} d^3 f(\mathbf{x}_k, \overline{\mathbf{u}} + \theta\Sigma\mathbf{v}; \Sigma\mathbf{v}), \end{aligned} \tag{4.4.13}$$

for a $\theta \in (0, 1)$. In the following we will construct estimates of $\mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma v)\right]$ in terms of $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_m)$. Applying this expansion to $\mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right]$ we find

$$\begin{aligned} \mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma v)\right] &= \mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}})\right] + \mathbb{E}\left[\mathrm{d}\, f(\mathbf{x}_k, \overline{\mathbf{u}}; \Sigma\mathbf{v})\right] + \frac{1}{2}\mathbb{E}\left[\mathrm{d}^2 f(\mathbf{x}_k, \overline{\mathbf{u}}; \Sigma\mathbf{v})\right] \\ &\quad + \frac{1}{6}\mathbb{E}\left[\mathrm{d}^3 f(\mathbf{x}_k, \overline{\mathbf{u}} + \theta\Sigma\mathbf{v}; \Sigma\mathbf{v})\right] \\ &= f(\mathbf{x}_k, \overline{\mathbf{u}}) + \mathrm{d}\, f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbb{E}\left[\mathbf{v}\right] + \frac{1}{2}\mathbb{E}\left[\mathbf{v}^T\Sigma^T \frac{d^2}{d\mathbf{u}^2} f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right] \\ &\quad + \frac{1}{6}\mathbb{E}\left[(I \otimes (\mathbf{v}^T\Sigma^T)^{\otimes 3}) D^3 f(\mathbf{x}_k, \overline{\mathbf{u}} + \theta\Sigma\mathbf{v}; \mathbf{v})\right] \end{aligned} \tag{4.4.14}$$

From Equation (4.4.14) we will now derive an expansion in terms of powers of

$(\sigma_1, \ldots, \sigma_m)$. To show that we can extract powers of Σ from the Kronecker products and from the expectation operators we use the fact that $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_m)$ is a diagonal matrix and thus

$$\begin{aligned}(\mathbf{v}^T\Sigma^T)^{\otimes 3} &= \left[(\Sigma\mathbf{v})^{\otimes 3}\right]^T \\ &= \left(\sigma_1^3 v_1^3, \sigma_2\sigma_1^2 v_2 v_1^2, \ldots, \sigma_m^3 v_m^3\right) \\ &= (\sigma_1, \ldots, \sigma_m)^{\otimes 3} \odot (\mathbf{v}^T)^{\otimes 3},\end{aligned}$$

where $\odot$ denotes the Hadamard Product [59, Section 3.6]. As finite dimensional norms are equivalently we may choose constants $c_1, \ldots, c_m$, such that $\sigma_i \leq c_i \|(\sigma_1, \ldots, \sigma_m)\|_2 =: \overline{\sigma}$. In this sense the variance components $(\sigma_1, \ldots, \sigma_m)$ decay at comparable rates, such that we may write $|\sigma_i\sigma_j\sigma_k| \leq C\overline{\sigma}^3$ for all indices $i, j, k \in \{1, \ldots, m\}$. As such we continue Equation (4.4.14) with

$$\begin{aligned}\mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma v)\right] &= f(\mathbf{x}_k, \overline{\mathbf{u}}) + \mathrm{d}\, f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma \underbrace{\mathbb{E}\left[\mathbf{v}\right]}_{=0} + \frac{1}{2}\mathbb{E}\left[\mathbf{v}^T\Sigma^T \frac{d^2}{d\mathbf{u}^2} f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right] \\ &\quad + \frac{1}{6}\mathbb{E}\left[\mathcal{O}(\overline{\sigma}^3)(I \otimes (\mathbf{v}^T)^{\otimes 3}) D^3 f(\mathbf{x}_k, \overline{\mathbf{u}} + \theta\Sigma\mathbf{v}; \mathbf{v})\right] \\ &= f(\mathbf{x}_k, \overline{\mathbf{u}}) + \frac{1}{2}\mathbb{E}\left[\mathbf{v}^T\Sigma^T \frac{d^2}{d\mathbf{u}^2} f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right] + \mathcal{O}(\overline{\sigma}^3).\end{aligned} \tag{4.4.15}$$

We now also apply expansion (4.4.13) to the variance $\operatorname{Var}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right]$. As f is a vector valued function its variance will be the covariance matrix

$$\operatorname{Var}\left[f(\mathbf{x}_k, \mathbf{v})\right] := \Big(\operatorname{Cov}\left[f_i(\mathbf{x}_k, \mathbf{v}), f_j(\mathbf{x}_k, \mathbf{v})\right]\Big)_{i,j=1,\ldots,m}. \tag{4.4.16}$$

In the following equation we use a shortened notation in the sense that we understand squares of vectors or vector-valued functions as the outer product. As an example we

can write the variance above as

$$\begin{aligned}\operatorname{Var}\left[f(\mathbf{x}_k,\mathbf{v})\right] &= \Big(\operatorname{Cov}\left[f_i(\mathbf{x}_k,\mathbf{v}),f_j(\mathbf{x}_k,\mathbf{v})\right]\Big)_{i,j=1,\dots,m}\\ &= \mathbb{E}\left[\left(f(\mathbf{x}_k,\mathbf{v})-E\left[f(\mathbf{x}_k,\mathbf{v})\right]\right)\left(f(\mathbf{x}_k,\mathbf{v})-E\left[f(\mathbf{x}_k,\mathbf{v})\right]\right)^T\right]\\ &= \mathbb{E}\left[\left(f(\mathbf{x}_k,\mathbf{v})-E\left[f(\mathbf{x}_k,\mathbf{v})\right]\right)^2\right].\end{aligned}$$

We take advantage of this notation and apply expansion (4.4.13) to $\operatorname{Var}\left[f(\mathbf{x}_k,\overline{\mathbf{u}}+\Sigma\mathbf{v})\right]$ to find

$$\begin{aligned}\operatorname{Var}\left[f(\mathbf{x}_k,\overline{\mathbf{u}}+\Sigma\mathbf{v})\right] &= \mathbb{E}\left[\left(f(\mathbf{x}_k,\overline{\mathbf{u}}+\Sigma\mathbf{v})-\mathbb{E}\left[f(\mathbf{x}_k,\overline{\mathbf{u}}+\Sigma\mathbf{v})\right]\right)^2\right]\\ &= \mathbb{E}\left[f(\mathbf{x}_k,\overline{\mathbf{u}}+\Sigma\mathbf{v})^2\right]-\mathbb{E}\left[f(\mathbf{x}_k,\overline{\mathbf{u}}+\Sigma\mathbf{v})\right]^2\\ &= \mathbb{E}\left[\Big(f(\mathbf{x}_k,\overline{\mathbf{u}})+\mathrm{d}\,f(\mathbf{x}_k,\overline{\mathbf{u}};\Sigma\mathbf{v})+\frac{1}{2}\,\mathrm{d}^2\,f(\mathbf{x}_k,\overline{\mathbf{u}};\Sigma\mathbf{v})\right.\\ &\qquad\left.+\frac{1}{6}\,\mathrm{d}^3\,f(\mathbf{x}_k,\overline{\mathbf{u}}+\theta\Sigma\mathbf{v};\Sigma\mathbf{v})\Big)^2\right]-\mathbb{E}\left[f(\mathbf{x}_k,\overline{\mathbf{u}}+\Sigma\mathbf{v})\right]^2\end{aligned}\tag{4.4.17}$$

Using the diagonal structure of Σ we can extract powers of $\overline{\sigma}$ from the expectation

operators in Equation (4.4.18). This leads to

$$
\begin{aligned}
\mathrm{Var}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right] &= \mathbb{E}\left[\left(f(\mathbf{x}_k, \overline{\mathbf{u}}) + \mathrm{d}\, f(\mathbf{x}_k, \overline{\mathbf{u}}; \Sigma\mathbf{v}) + \frac{1}{2}\,\mathrm{d}^2 f(\mathbf{x}_k, \overline{\mathbf{u}}; \Sigma\mathbf{v})\right.\right. \\
&\qquad \left.\left. + \frac{1}{6}\,\mathrm{d}^3 f(\mathbf{x}_k, \overline{\mathbf{u}} + \theta\Sigma\mathbf{v}; \Sigma\mathbf{v})\right)^2\right] - \mathbb{E}\left[\nabla_{\mathbf{u}} f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right]^2 \\
&= f(\mathbf{x}_k, \overline{\mathbf{u}})^2 + \frac{1}{2}\mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}})\left(\mathbf{v}^T\Sigma\frac{d^2}{d\mathbf{u}^2}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)^2\right] \\
&\qquad + \mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}})\left(\frac{d}{d\mathbf{u}}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)^2\right] \\
&\qquad + \mathbb{E}\left[\left(\frac{d}{d\mathbf{u}}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)\left(\frac{d}{d\mathbf{u}}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)^T\right] \\
&\qquad - \mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right]^2 + \mathcal{O}(\overline{\sigma}^3) \\
&= f(\mathbf{x}_k, \overline{\mathbf{u}})^2 + \frac{1}{2}\mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}})\left(\mathbf{v}^T\Sigma\frac{d^2}{d\mathbf{u}^2}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)^T\right] \\
&\qquad + \mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}})\left(\frac{d}{d\mathbf{u}}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)^T\right] \\
&\qquad + \mathbb{E}\left[\left(\frac{d}{d\mathbf{u}}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)\left(\frac{d}{d\mathbf{u}}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)^T\right] \\
&\qquad - f(\mathbf{x}_k, \overline{\mathbf{u}})^2 - f(\mathbf{x}_k, \overline{\mathbf{u}})\mathbb{E}\left[\frac{1}{2}\mathbf{v}^T\Sigma\frac{d^2}{d\mathbf{u}^2}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right]^T + \mathcal{O}(\overline{\sigma}^3) \\
&= \mathbb{E}\left[\left(\frac{d}{d\mathbf{u}}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)\left(\frac{d}{d\mathbf{u}}f(\mathbf{x}_k, \overline{\mathbf{u}})\Sigma\mathbf{v}\right)^T\right] + \mathcal{O}(\overline{\sigma}^3).
\end{aligned}
\tag{4.4.18}
$$

Using this approximation of $\mathrm{Var}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right]$ we proceed to compute the approxi-

mate bound

$$
\begin{aligned}
\operatorname{Var}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right]_{i,j} &= \operatorname{Cov}\left[f_i(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v}), f_j(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right] \\
&= \mathbb{E}\left[\frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}})^T \Sigma\mathbf{v}\mathbf{v}^T\Sigma^T \frac{d}{d\mathbf{u}} f_j(\mathbf{x}_k, \overline{\mathbf{u}})\right] + \mathcal{O}(\overline{\sigma}^3) \\
&= \frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}})^T \mathbb{E}\left[\Sigma\mathbf{v}\mathbf{v}^T\Sigma^T\right] \frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}}) + \mathcal{O}(\overline{\sigma}^3) \\
&= \frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}})^T \operatorname{diag}(\sigma_1^2, \ldots, \sigma_m^2) \frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}}) + \mathcal{O}(\overline{\sigma}^3) \\
&= \sum_{l=1}^{m} \sigma_l^2 \partial_{u_l} f_i(\mathbf{x}_k, \overline{\mathbf{u}}) \partial_{u_l} f_j(\mathbf{x}_k, \overline{\mathbf{u}}) + \mathcal{O}(\overline{\sigma}^3) \\
&= \left(\Sigma \frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}})\right)^T \left(\Sigma \frac{d}{d\mathbf{u}} f_j(\mathbf{x}_k, \overline{\mathbf{u}})\right) + \mathcal{O}(\overline{\sigma}^3) \\
&\leq \|\Sigma\|_2^2 \left\| \frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}}) \right\|_2 \left\| \frac{d}{d\mathbf{u}} f_j(\mathbf{x}_k, \overline{\mathbf{u}}) \right\|_2 + \mathcal{O}(\overline{\sigma}^3).
\end{aligned}
\tag{4.4.19}
$$

We can thus estimate the entries of the covariance matrix $\Sigma_{\mathbf{y}} = \operatorname{Var}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right]$ using the constraint Jacobian $\frac{d}{d\mathbf{u}} f(\mathbf{x}_k, \overline{\mathbf{u}})$.

To verify this approximation numerically we let select an initial state for our vehicle model

$$
\mathbf{x}_0 = (x_0, y_0, \psi_0, v_{x,0}, v_{y,0}, \omega_0)^T = (0, 0, 0, 10, 0, 0.1)^T.
$$

Starting from $\mathbf{x}_0$ we simulate state steps by sampling actions according to the action distribution

$$
\mathbf{u} \sim \pi(\mathbf{x}_0) = \mathcal{N}\left(\left(5.0, 5.0\frac{\pi}{180}\right)^T, \operatorname{diag}\left(\sigma_1, \sigma_2\right)\right),
$$

where we use identical variance values in both control dimensions $\sigma_1 = \sigma_2 = \tilde{\sigma}$. From this action distribution we can now estimate the covariance matrix of state variables

$$
\Sigma_{\mathbf{y}} = \operatorname{Cov}_{\mathbf{u}\sim\pi(\mathbf{x}_0)}\left[f(\mathbf{x}_0, \mathbf{u})\right] \tag{4.4.20}
$$

from Monte-Carlo samples. Given a variance level $\tilde{\sigma}$ we compute 10^7 samples of

$$f(\mathbf{x}_0, \mathbf{u}), \qquad \mathbf{u} \sim \pi(\mathbf{x}_0),$$

which we denote by $f^{(1)}, f^{(2)}, \ldots, f^{(10^7)} \in \mathbb{R}^n$. This lets us write a matrix of all sampled variables

$$A = \begin{pmatrix} f_1^{(1)} & f_2^{(1)} & \ldots & f_n^{(1)} \\ & & \vdots & \\ f_1^{(10^7)} & f_2^{(10^7)} & \ldots & f_n^{(10^7)} \end{pmatrix} = \left[\begin{array}{c|c|c|c} \mathbf{a}^{(1)} & \mathbf{a}^{(2)} & \ldots & \mathbf{a}^{(n)} \end{array} \right],$$

where the column vectors $\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(n)}$ of A, are the sample vectors for each state. From these samples, we can construct the covariance estimate matrix

$$\hat{\Sigma}_{\mathbf{y}} := \left(\sum_{k=1}^{10^7} \frac{1}{10^7 - 1} \left(\mathbf{a}_k^{(i)} - \overline{\mathbf{a}^{(i)}} \right) \left(\mathbf{a}_k^{(j)} - \overline{\mathbf{a}^{(j)}} \right) \right)_{i,j=1,\ldots,n}. \tag{4.4.21}$$

From Equation (4.4.19), we obtain the alternative estimate of $\Sigma_{\mathbf{y}}$ in the form of

$$\tilde{\Sigma}_{\mathbf{y}} := \left(\Sigma \frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}}) \right)^T \left(\Sigma \frac{d}{d\mathbf{u}} f_j(\mathbf{x}_k, \overline{\mathbf{u}}) \right). \tag{4.4.22}$$

We now compare the estimate $\tilde{\Sigma}_{\mathbf{y}}$ to the reference estimate $\hat{\Sigma}_{\mathbf{y}}$ for difference variance levels $\tilde{\sigma}$. Figure 4.24 shows the difference between the two estimates $\tilde{\Sigma}_{\mathbf{y}}$ and $\hat{\Sigma}_{\mathbf{y}}$.

We observe that for higher variance levels, such as $\tilde{\sigma} \in (10^{-2}, 1)$, the estimate $\|\tilde{\Sigma}_{\mathbf{y}} - \hat{\Sigma}_{\mathbf{y}}\|_F$ converges at least as fast as $\mathcal{O}(\tilde{\sigma}^3)$. Symmetry effects and $\mathbb{E}\left[\mathbf{v}^{\otimes 3}\right] = 0$ for $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, I_m)$ may cause this faster convergence. The slower convergence at lower variance levels, $\tilde{\sigma} < 10^{-2}$, is likely due to an overall approximation error in the reference estimate $\hat{\Sigma}_{\mathbf{y}}$.

We now have a method to obtain estimates of $\Sigma_{\mathbf{y}}$ without requiring action samples or higher than first order derivatives of the dynamics function f. While Equa-

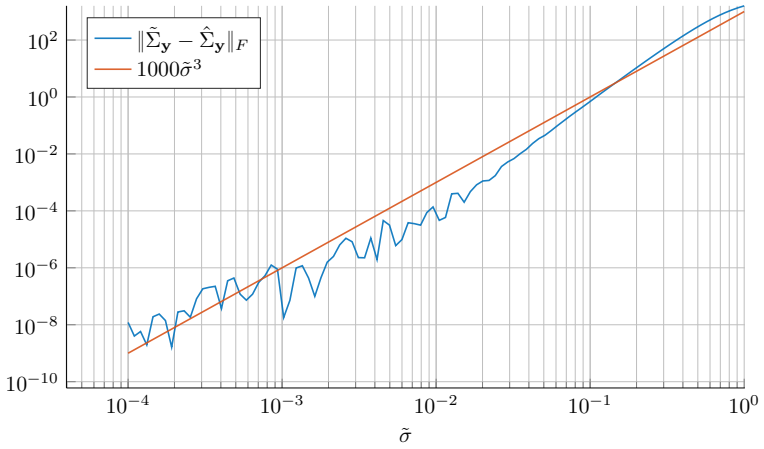


Figure 4.24: Covariance approximation errors at difference $\tilde{\sigma}$ levels

tion (4.4.14) provides us with the direct estimate of the mean action in the form of

$$\mathbb{E}\left[f(\mathbf{x}_k, \overline{\mathbf{u}} + \Sigma\mathbf{v})\right] = f(\mathbf{x}_k, \overline{\mathbf{u}}) + \frac{1}{2}\mathbb{E}\left[\mathbf{v}^T\Sigma^T \frac{d}{d\mathbf{u}} f(\mathbf{x}, \overline{\mathbf{u}})\Sigma\mathbf{v}\right] + \mathcal{O}(\overline{\sigma}^3), \tag{4.4.23}$$

we seek to obtain an estimate that does not depend on the variance level $\overline{\sigma}$. To this end, we use $\tilde{\Sigma}_{\mathbf{y}}$ as a variance estimate and, based on an accuracy level, compute the required number of Monte-Carlo samples to estimate the mean action $\mathbb{E}_{\mathbf{u}\sim\pi(\mathbf{x}_k)}\left[f(\mathbf{x}_k, \mathbf{u})\right]$.

Given a current state $\mathbf{x}_k$, tolerance $\Delta > 0$, and an accuracy parameter $\rho \in (0, 1)$, we construct the covariance matrix estimate $\tilde{\Sigma}_{\mathbf{y}}$ and compute the number of required samples N, such that with probability $p \geq \rho$ the step sample mean $\hat{\mathbb{E}}_{\mathbf{u}\sim\pi(\mathbf{x}_k)}\left[f(\mathbf{x}_k, \mathbf{u})\right]$ is contained in $B_{\Delta,\infty}\left(\mathbb{E}\left[f(\mathbf{x}_k, \mathbf{u})\right]\right)$. This procedure gives rise to the following strategy extraction algorithm. Using Algorithm 14, we can use agents to compute trajectories, which we may either use directly or use as a starting point for a collocation solver. We will discuss the application of Algorithm 14 as a starting point for collocation solvers in detail in Section 4.5.

We now examine the trajectory reconstructed by Algorithm 14. As an example, we apply the model trained in Section 4.3.6 to the "Double S"-track (see Figure 4.6. In Figures 4.25, and 4.26, we see the controls and trajectory we computed following

Algorithm 14 Strategy extraction

Require: $\mathbf{x}_0 \in \mathcal{X}, \pi : \mathcal{X} \to M^1(\mathcal{A}), \Delta > 0, \rho \in (0,1), N_{\max} \geq 1$.

$t \leftarrow 0$.
$X \leftarrow \emptyset$.
$k \leftarrow 0$.
while $\mathbf{x}_k \in \mathcal{X}$ and $t \leq T_{\max}$ **do**
    $\overline{\mathbf{u}} \leftarrow \mathbb{E}\left[\pi(\mathbf{x}_k)\right]$.
    $\hat{\Sigma}_{\mathbf{y}} \leftarrow \left(\Sigma \frac{d}{d\mathbf{u}} f_i(\mathbf{x}_k, \overline{\mathbf{u}})\right)^T \left(\Sigma \frac{d}{d\mathbf{u}} f_j(\mathbf{x}_k, \overline{\mathbf{u}})\right)$.
    Select $N \geq 1$ such that $\rho^{1/m} \leq 2\Phi\left(\frac{\Delta\sqrt{N}}{\|\hat{\Sigma}_{\mathbf{y}}\|}\right) - 1$
    Let $N \leftarrow \min(N_{\max}, N)$
    Sample $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(N)}$ iid with $\mathbf{u}^{(1)} \sim \pi(\mathbf{x}_k)$.
    Compute $f^{(1)}, \ldots, f^{(N)}$ using $f^{(i)} = f(\mathbf{x}_k, \mathbf{u}^{(i)})$.
    Estimate $\hat{\mathbb{E}}\left[f(\mathbf{x}_k, \mathbf{u})\right] \leftarrow \frac{1}{N}\sum_{i=1}^{N} f^{(i)}$.
    Solve $\mathbf{u}_k \leftarrow \operatorname{argmin}_{\mathbf{u}\in\mathcal{A}} \|f(\mathbf{x}_k, \mathbf{u}) - \hat{\mathbb{E}}\left[f(\mathbf{x}_k, \mathbf{u})\right]\|$
    $X \leftarrow X \cup \{(\mathbf{x}_k, \mathbf{u}_k)\}$.
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \delta_t f(\mathbf{x}_k, \mathbf{u}_k)$.
    $t \leftarrow t + \delta_t$
    $k \leftarrow k + 1$
end while
return X.

Algorithm 14 with parameters $N_{\max} = 10^6, \rho = 0.9, \Delta = 0.01, \delta_t = 0.01$. From the control variables in Figure 4.25, we observe high frequency oscillations in the steering input. We further see that the rear wheel acceleration input closely matches the maximal value of 20ms^{-2}, except for short periods. Near the points $t = 8.5, t = 25.2, t = 26.8$ we find steep decreases in the acceleration input.

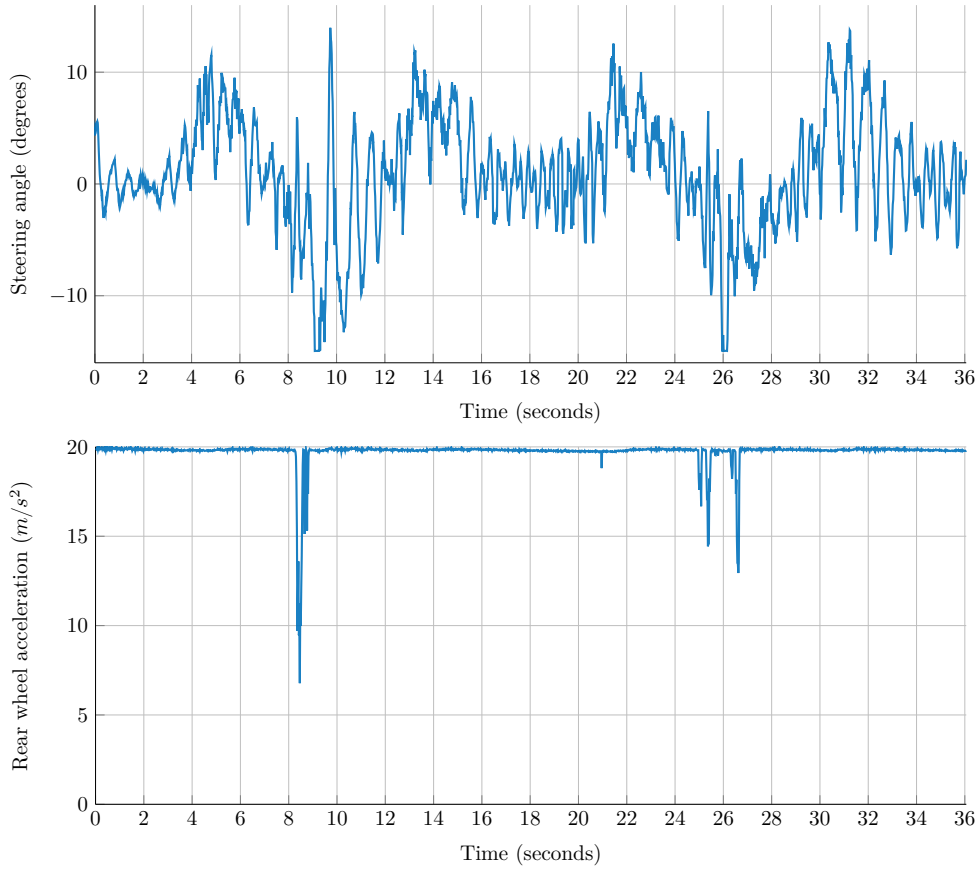


Figure 4.25: Extracted control strategy using Algorithm 14

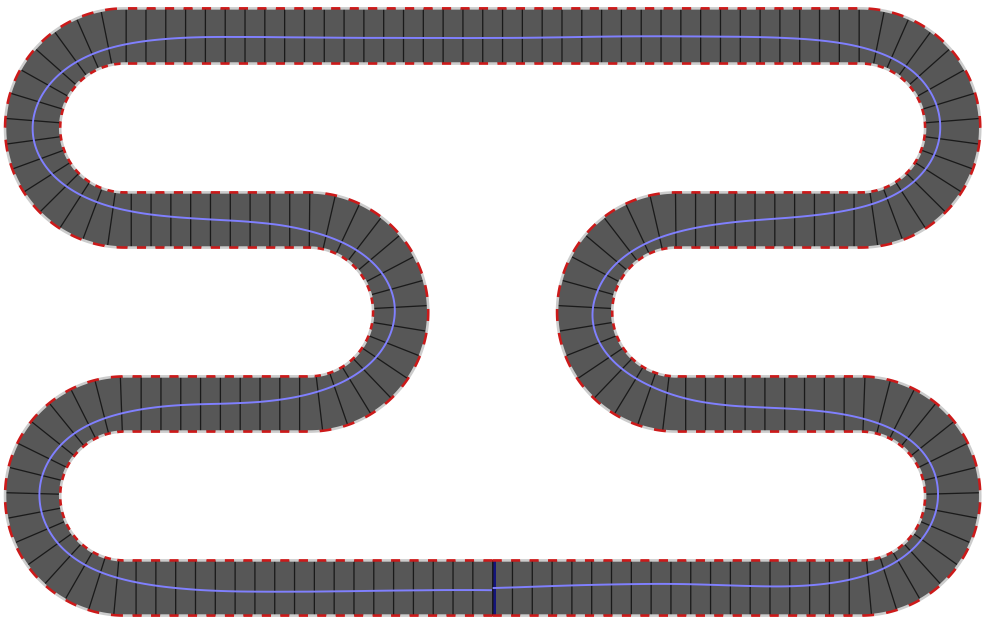
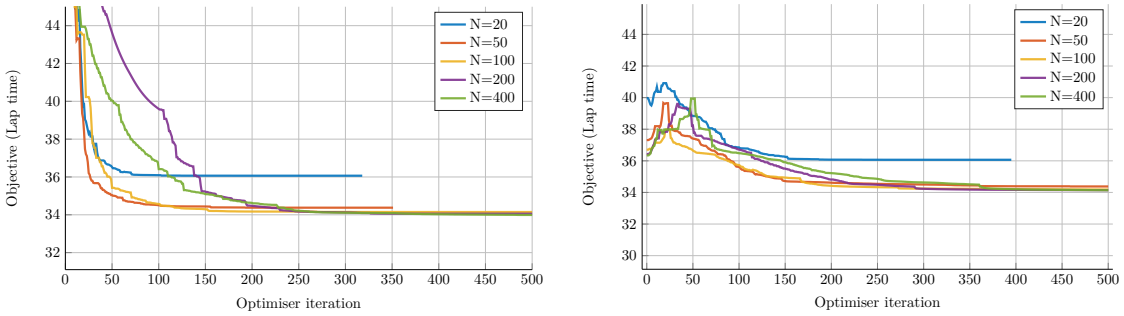


Figure 4.26: Extracted trajectory using Algorithm 14

4.5 Initialisation improvements

Algorithm 14 allows us the evaluate the expected trajectory an agent takes around a given racetrack. We will now apply this extraction algorithm to the pre-trained agents from Section (4.3.6), to warm start a collocation algorithm with a high quality initialisation. In this context, we interpret an initialisation to be of high quality if it achieves low constraint violations and near optimal objective values simultaneously.

We recall the optimisation behaviour, when applying the `trust-const` algorithm to the minimal lap time problem (3.4.16) shown in Figure 3.5. Figure 3.5 shows a significant reduction in objective value during the initial part of the simulation. We expect to accelerate this initial part of the optimisation process by utilising a trajectory generated using Algorithm 14. We interpolate the trajectory generated by Algorithm 14 using linear splines onto the node positions used by our optimiser. Figure 4.27 shows a comparison of the achieved objective reduction by using Algorithm 14. Comparing Figures 4.27a and 4.27b, we see that, while the initial lap



(a) Objective reduction on track 4.5b without initialisation

(b) Objective reduction on track 4.5b with initialisation

Figure 4.27: Comparison of the objective reduction achieved by `trust-constr` on the "S Double" test track 4.5b

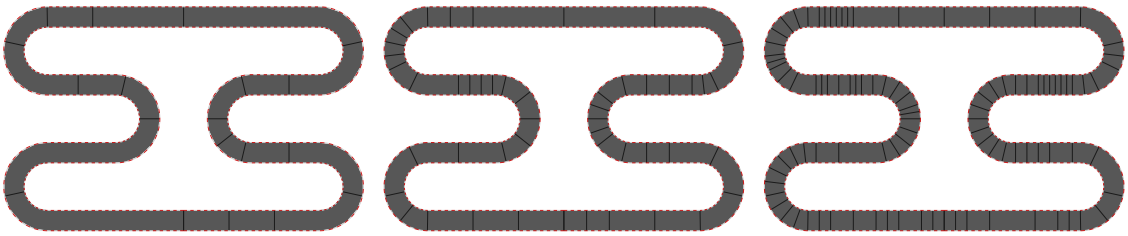
times are lower, the problems converge to similar values. The transient behaviour in Figure 4.27b shows a rise in lap times during the initial stages of the optimisation.

Let us again consider the control variables shown in Figure 4.26, where we see steep dips in the acceleration control, and oscillatory behaviour of the steering control. A

linear projection of these variables onto an equidistant grid will not be able to capture this behaviour accurately, unless a large number of collocation nodes is used. As an alternative, we propose to use the agent-generated trajectory shown in Figure 4.26 to construct a problem specific grid. The grid construction starts by describing the entire domain using three equidistant nodes and two resulting trapezoidal segments. We then refine the mesh until a desired number of nodes is achieved. To refine the mesh, we approximate the constraint violation incurred on each segment using the midpoint quadrature rule

$$c_i := (s_i - s_{i-1}) \left\| \tilde{\mathbf{x}}_i - \tilde{\mathbf{x}}_{i-1} - \frac{s_i - s_{i-1}}{2} \left(\tilde{\mathbf{f}}(\tilde{\mathbf{x}}_i, \tilde{\mathbf{u}}_i) \right) \right\|. \tag{4.5.1}$$

We use the notation from Equation (3.4.13), to indicate that we are operating on the track distance domain. We then select the segment corresponding to the highest constraint violation c_i and refine it by splitting the segment in half. Figure 4.28 shows a sequence of meshes build using this refinement process.



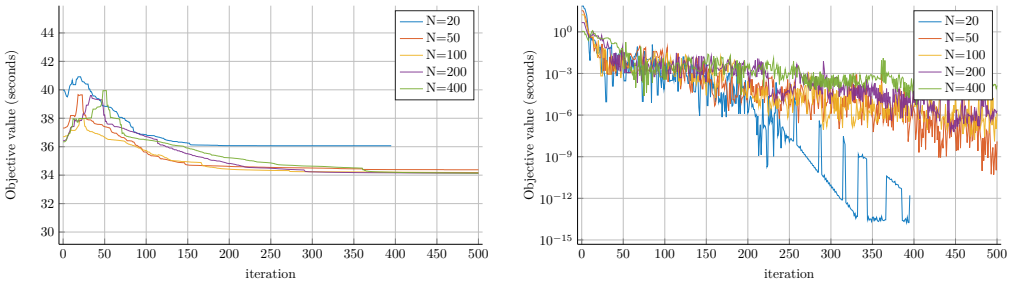
(a) Mesh with $N = 20$ nodes (b) Mesh with $N = 50$ nodes (c) Mesh with $N = 100$ nodes

Figure 4.28: Iterative mesh refinement on "Double-S"-Track

Table 4.8: Lap time values using `trust-constr`

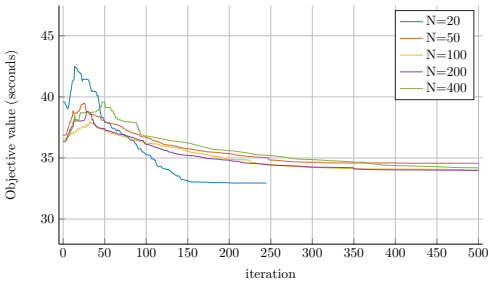
Mesh \ Iteration	50	100	200	300	400	500
Equidistant ($N = 100$)	36.650	35.734	34.420	34.241	34.209	34.184
Refined ($N = 100$)	37.217	36.221	34.898	34.653	34.095	33.985
Equidistant ($N = 200$)	38.232	36.712	34.815	34.231	34.164	34.141
Refined ($N = 200$)	37.399	36.109	34.811	34.259	34.028	33.987

We see in Figure 4.29 and Table 4.8, that by constructing the collocation segments

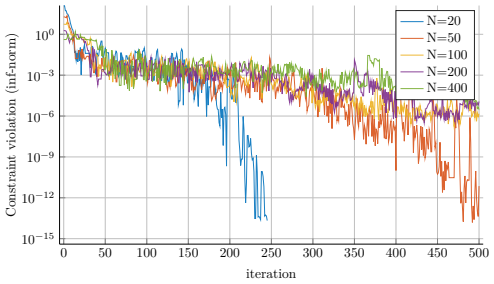


(a) Lap time reduction on equidistant mesh

(b) Constraint violation on equidistant mesh



(c) Lap time reduction on refined mesh



(d) Constraint violation on refined mesh

Figure 4.29: Optimisation results using `trust-constr` on the "Double-S" track

iteratively we are able to achieve lower overall lap times, given a sufficient number of optimisation iterations. We see from Table 4.8 that solutions constructed with $N = 100$ nodes on the refined mesh achieve lower overall lap times than solutions at a finer equidistant mesh with $N = 200$ nodes.

4.6 Software aspects

Throughout this project, we encountered a series of computationally challenging problems. Solving these problems required specialised software, as a consequence we developed a series of software projects. We have seen one example of such a software project in Section 2.3.2, where we presented a piece of specialised software which allowed us to trace a racetrack from image data and transform it into a format we can use for further simulation and analysis. The results in Section 3 were achieved by developing a Python3 software package for collocation optimisation called `colloptpy`. To perform reinforcement learning we needed an implementation of the racing environment

described in Section 4. The reinforcement learning environment we developed uses the track model from Section 2.3.1 and provides access to the described observations in a manner that is compatible with the `gym` framework developed by OpenAI [16]. In this section, we present an overview of individual packages by discussing their features and aspects of their software infrastructure.

4.6.1 Colloptpy

The `colloptpy` package we developed enables the user to solve collocation problems, such as the problems we discussed in Chapter 3. The `colloptpy` package provides users with a flexible infrastructure to define problem dynamics function

$$\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n, \qquad \frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)).$$

Users define the vectorised application of the dynamics function $\mathbf{f}$ using `pytorch` expressions. As sequential operations in python can reduce the overall computationally throughput significantly, we define

$$\tilde{\mathbf{f}} : \mathbb{R}^{k\times n} \times \mathbb{R}^{k\times m} \to \mathbb{R}^{k\times n}, \qquad \tilde{\mathbf{f}} = \begin{pmatrix} \mathbf{f}(\mathbf{x}_1, \mathbf{u}_1) \\ \vdots \\ \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) \end{pmatrix}$$

where k is an arbitrary vectorisation length. By relying `pytorch` and vectorised operations, we benefit from the automatic differentiation capabilities of `pytorch` and its associated functional derivative library `functorch`. The use of `pytorch` also enabled us to support graphics processing unit (GPU) accelerated derivative evaluation, resulting in the performance gains we discussed in Section 3.6.1. The `colloptpy` package is available on the python package index site [44]. Figure 4.30 shows a class model diagram for `colloptpy`. In Figure 4.30, we see that a collocation problem

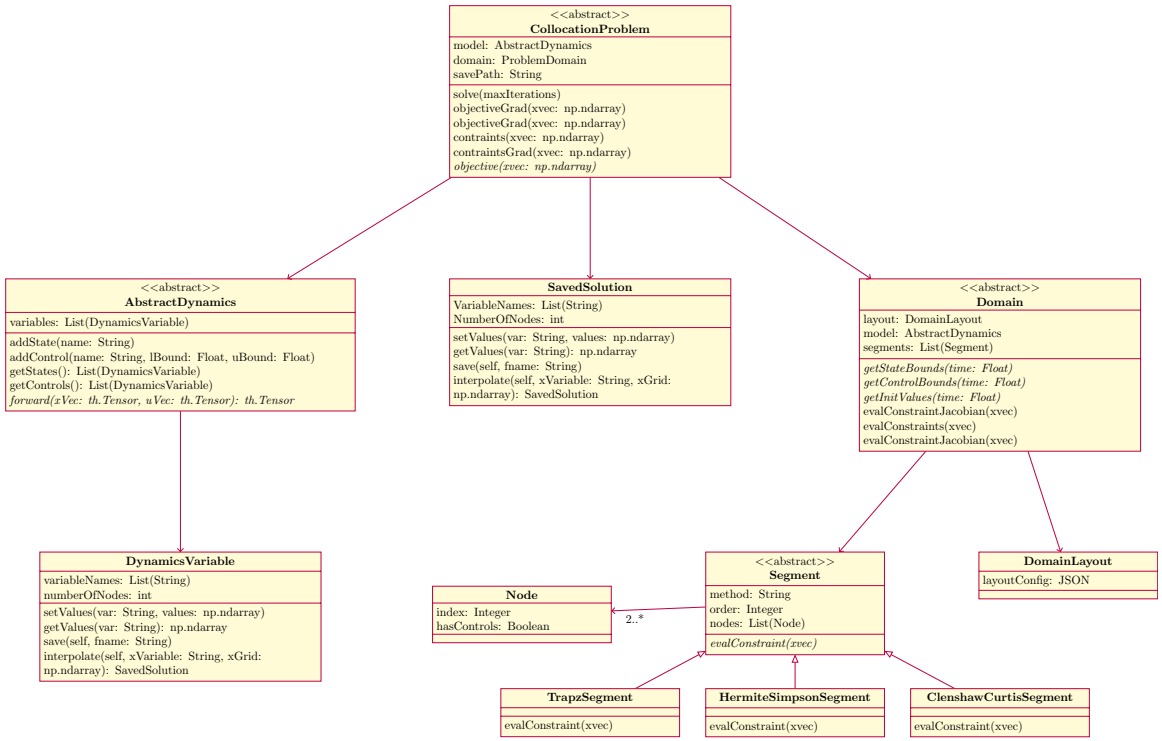


Figure 4.30: Class Diagram of the `colloptpy` optimisation package

consists of three main components, the dynamics, the domain, and a method for saving and loading solutions. Further we see that dynamics and domains are abstract classes in `colloptpy` and, as such, require subclasses to be instantiated. This structure ensures that the problem specific methods, such as the dynamics application method *forward* and the domain specific functions *getStateBounds*, *getControlBounds*, and *getInitValues* for setting variable bounds and initial guesses, are implemented. Each domain object contains a series of segments. Each segment may differ in the quadrature method used and the corresponding number of collocation nodes required for each segment. The domain layout defines which segments and quadrature orders should be used.

4.6.2 Track model software

We require a software package that allows us to represent our track model from Sections 2.3.1 and enable the track designer from Section 2.3.2. Our track model

package is called `PyRacetrack2D` [45] and supports creation, as well as modification of tracks. Tracks can be saved to and loaded from disk using a Javascript object notation (JSON) representation of a race track. Once a track is instantiated we can access the track geometry in the form of so called track-point objects. A track-point represents a location on the track's centre line and contains the global position in (x, y) coordinates, the track orientation χ, and the local curvature K. We follow Algorithm 15 to extract a track-point using a track-distance $s \in [0, L]$.

Algorithm 15 Track point extraction

Require: $s \in [0, L], \{l_1, \ldots, l_N\}, (x_0, y_0), \omega_0$. ▷ Element lengths $l_1, \ldots, l_N$.
$i \leftarrow 1$.
$\omega \leftarrow \omega_0$
$(x, y) \leftarrow (x_0, y_0)$
while $s > l_i$ **do**
    $s \leftarrow s - l_i$
    $i \leftarrow i + 1$
    assign the end point and orientation of element i to $(x, y), \omega$
end while
assign point and orientation at distance s of element i to $(x, y), \omega$
if Element i is curve piece **then**
    assign the curvature to K
else
    $K \leftarrow 0$ ▷ Straight pieces have zero curvature
end if
return $(x, y), \omega$

In Section 4.3.1, we required the ability to compute ray intersections with the track boundary in an efficient manner. To compute these ray intersections, `PyRacetrack2D` discretises the track boundary with straight line segments and computes the intersections using the `numba` just in time (JIT) compilation engine [56], for improved computational performance. In Section 4.3.2, we discussed the benefits of enforcing the track constraints geometrically. The track model supports this geometric boundary check using vectorised `numpy` operations. We can see a class model for `PyRacetrack2D` in Figure 4.31.

In Figure 4.31, we see that a track consists of sequence of track elements. Each of

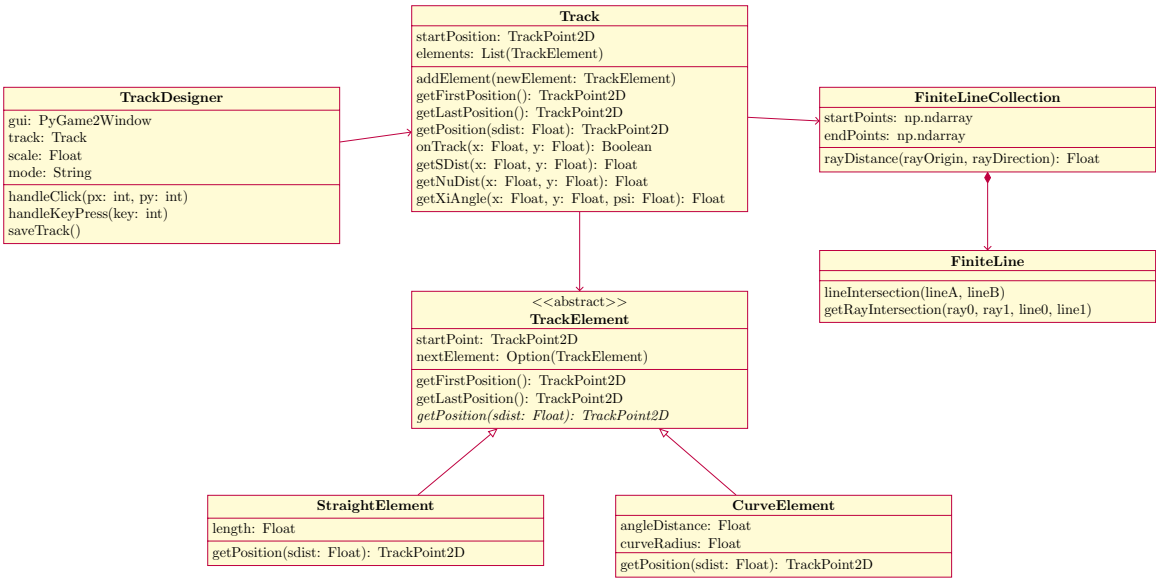


Figure 4.31: Class model of the `PyRacetrack2D` package

these track elements is either a straight line or a curve. We also see in Figure 4.31, that instances of the track class can evaluate track positions by following Algorithm 15. We find the classes *FiniteLine* and *FiniteLineCollection*, which handle the ray intersection computation with the approximated track boundary. On the left of the track and track element classes, we find the track designer, which enables the graphical interface that we have previously seen in section 2.3.2.

4.6.3 RaceGame2D

The reinforcement learning algorithms we discussed in Section 4.3.3 relied on pixel data, that shows the vehicle in relation to the race track. To handle this rendering we develop the `RaceGame2D` [46] package. Figure 4.32 shows the components of `RaceGame2D` and we can see that the display handling is split amongst three display instances, the *VirtualDisplay*, the *PyGameDisplay*, and the *RacecarDisplay*. The virtual display handles positioning elements on screen by transforming real world coordinate data into screen coordinate data, while the *PyGameDisplay* handles the rendering of graphical elements on the computer screen. Lastly, the *RacecarDisplay*

handles communication between the *VirtualDisplay* and the *PyGameDisplay*. For the

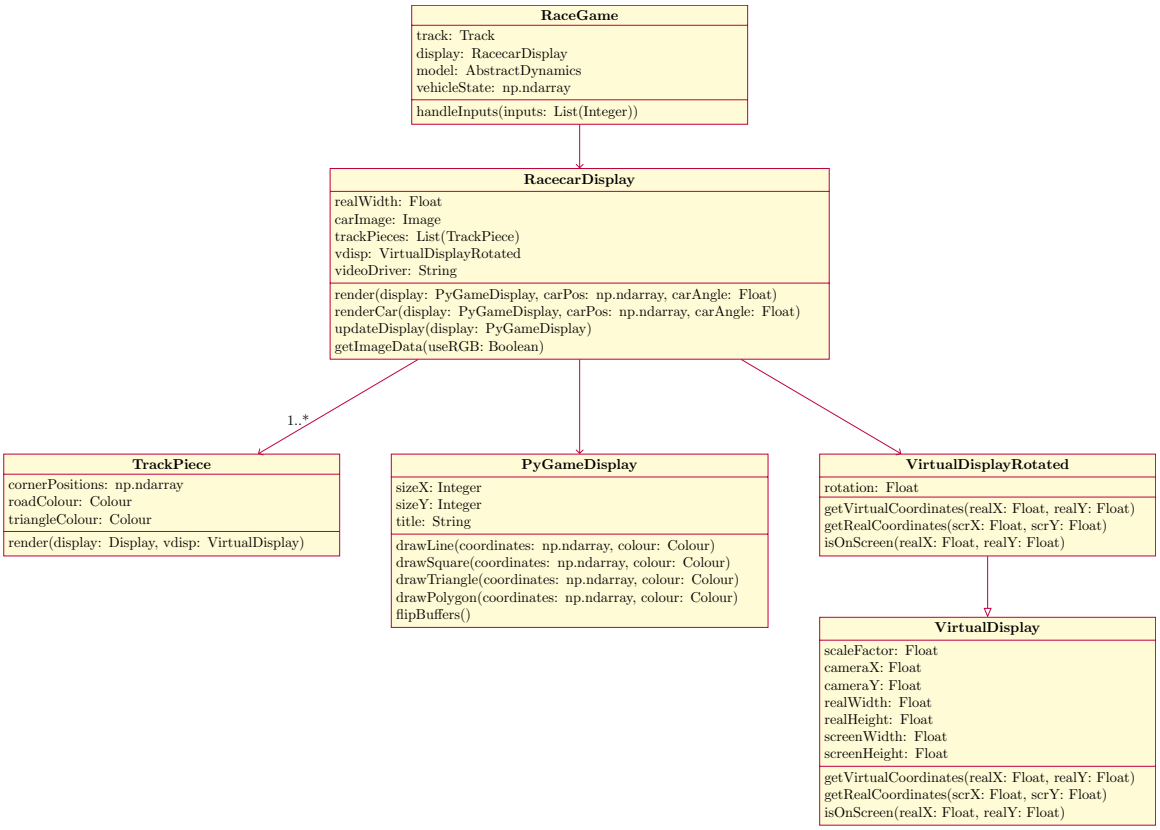


Figure 4.32: Class model of the `RaceGame2D` package

application in reinforcement learning environments, we want to highlight the variable "displayDriver" within the *RacecarDisplay* class. This variable is set as an environment variable in the main process and allows us to avoid outputting to the screen. This reduces the resource requirements of running multiple *RacecarDisplay* instances simultaneously and even enables us to use the display classes without an active display server.

4.6.4 RaceGym2D

We now build upon the `RaceGame2D` package and construct the different learning environments described in Sections 4.2 and 4.3 in the form of `gym` environments.

The `gym` framework, developed by OpenAI [16], provides a standardised method of

implementing reinforcement learning environments. Each `gym` environment need to have the four components: (i) a method called *step*, which handles actions and return rewards and observations, (ii) a method called *reset* to return the environment to an initial state, (iii) an *observation space*, and (iv) an *action space*. The `RaceGym2D` [47] package implements four different `gym` environments, each corresponding to one of the four environments we developed in Sections 4.2 and 4.3.

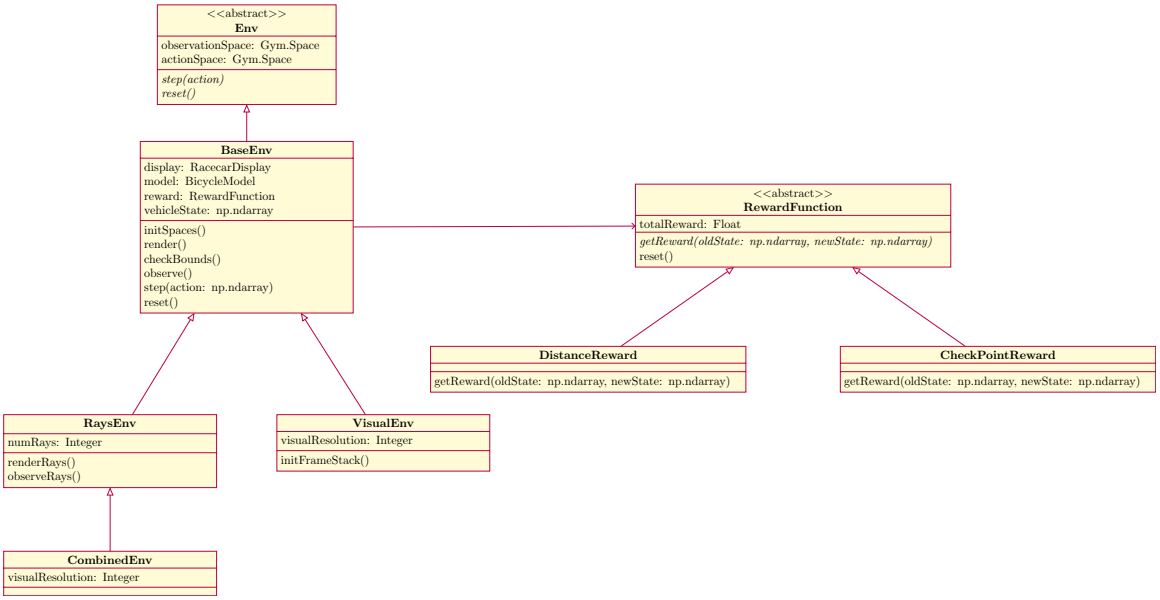


Figure 4.33: Class model of the `RaceGym2D` package

Chapter 5

Conclusion

5.1 Summary

In the first two chapters (Chapter 1 and Chapter 2), of this work we introduced the relevant optimal control theory and derived the classical bicycle model including tyre and track dynamics. At the end of Chapter 2, we were able to formulate the dynamics for our optimal control problem and we had also developed a specialised software, which lets trace and design racing circuits.

In Chapter 3, we discussed the theory of direct collocation methods and algorithms such as `trust-constr` we use to solve them. We discovered that in our lap time optimisation problem, a large amount of computational effort is spend on evaluating gradients. To reduce the computational costs we used the automatic differentiation capabilities of `functorch` to accelerate our collocation solver. By exploiting the structure of our collocation problem, we were also able to derive approximate gradients to further accelerate the simulation. Lastly, in Chapter 3, we proposed a mesh refinement algorithm based on L^1 polynomial approximation. Based on work from [64] we showed how one can used exact Clenshaw–Curtis quadrature away from singular-

ities, combined with iterative mesh refinement to efficiently compute L^1 polynomial approximation.

In Chapter 4, we first introduced the theory of reinforcement learning algorithms. Based on this, we developed a series of environments to train reinforcement learning agent on our lap time minimisation problem. The three primary feature sets we used to solve our problem were: 1. state variable information, 2. distance measurements to the track boundary, and 3. pixel data information. We discussed how each feature set is able to improve upon the previous one. By combining all three feature sets, we were able to train agents that perform effectively on representations of Formula One tracks. Using variance estimates we were able to extract the strategies of pre-trained agents and accelerate collocation based methods.

5.2 Future work

In this work, we have seen the application of reinforcement learning methods to classical two dimensional vehicle models. Learning on visual data was also based on a two dimensional top down view of the environment. Extensions to higher fidelity, three dimensional, vehicle models would be extremely interesting from a practical perspective. The networks we trained in this work are relatively low dimensional and have not been trained extensively past the point where they achieve a general level of competence on a given track. While we showed that PPO is able to perform reasonably well the current environments, we would like to explore the performance of different algorithms, with increased training resources on more accurate physical models.

Reinforcement learning, compared to supervised learning, lets us interact freely with a given environment. The answers provided by reinforcement learning could prove invaluable to self driving vehicle tasks as they could allow us to perform well in

edge case scenarios. Lap time optimisation is valuable in this regard as it is designed to push a vehicle towards the physical limit.

Future research opportunities also arise from the direct combination of reinforcement learning agents within optimal control problems. We could use an agents strategy to inform the mesh construction for collocation problems or back tracing in case the optimiser is not progressing.

Bibliography

[1] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Transactions on Mathematical Software*, 45(1):2:1–2:26, 2019.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] B. Armstrong-Helouvry. Stick-slip arising from stribeck friction. In *Proceedings., IEEE International Conference on Robotics and Automation*, pages 1377–1382. IEEE, 1990.

[4] V. I. Arnol'd. *Mathematical methods of classical mechanics*, volume 60. Springer Science & Business Media, 2013.

[5] I. Babuška and M. Suri. The p-and hp versions of the finite element method, an overview. *Computer Methods in Applied Mechanics and Engineering*, 80(1):5–26, 1990. Publisher: Elsevier.

[6] R. E. Bellman and S. E. Dreyfus. Applied dynamic programming. *Ledelse og Erhvervsøkonomi*, pages 363–371, 1962. Publisher: Princeton University Press.

[7] D. A. Benson, G. T. Huntington, T. P. Thorvaldsen, and A. V. Rao. Direct trajectory optimization and costate estimation via an orthogonal collocation method. *Journal of Guidance, Control, and Dynamics*, 29(6):1435–1440, 2006.

[8] E. Bertolazzi, F. Biral, and M. Da Lio. Symbolic–numeric indirect method for solving optimal control problems for large multibody systems. *Multibody System Dynamics*, 13(2):233–252, 2005. Publisher: Springer.

[9] J. T. Betts. *Practical methods for optimal control and estimation using nonlinear programming*. SIAM, 2010.

[10] J. T. Betts and W. P. Huffman. Mesh refinement in direct transcription methods for optimal control. *Optimal Control Applications and Methods*, 19(1):1–21, 1998. Publisher: Wiley Online Library.

[11] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, 2022.

[12] M. Blundell and D. Harty. *Multibody systems approach to vehicle dynamics*. Elsevier, 2004.

[13] H. G. Bock and K.-J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. *IFAC Proceedings Volumes*, 17(2):1603–1608, 1984. Publisher: Elsevier.

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

[15] D. Brayshaw and M. Harrison. A quasi steady state approach to race car lap simulation in order to understand the effects of racing line and centre of gravity location. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 219(6):725–739, 2005. Publisher: Sage Publications Sage UK: London, England.

[16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[17] A. Bryson. Optimal control-1950 to 1985. *IEEE Control Systems Magazine*, 16(3):26–33, 1996.

[18] R. Bulirsch, J. Stoer, and J. Stoer. *Introduction to numerical analysis*. Springer, 1991.

[19] P. Cai, X. Mei, L. Tai, Y. Sun, and M. Liu. High-speed autonomous drifting with deep reinforcement learning. *IEEE Robotics and Automation Letters*, 5(2):1247–1254, 2020. Publisher: IEEE.

[20] C. Canudas de Wit, H. Olsson, K. J. Astrom, and P. Lischinsky. A new model for control of systems with friction. *IEEE Transactions on Automatic Control*, 40(3):419–425, 1995. Publisher: IEEE.

[21] C. Canudas-de Wit, P. Tsiotras, E. Velenis, M. Basset, and G. Gissinger. Dynamic friction models for road/tire longitudinal interaction. *Vehicle System Dynamics*, 39(3):189–226, 2003. eprint: https://www.tandfonline.com/doi/pdf/10.1076/vesd.39.3.189.14152.

[22] D. Casanova. On minimum time vehicle manoeuvring: The theoretical optimal lap, 2000. Publisher: Cranfield University.

[23] J. E. Chacón and T. Duong. *Multivariate kernel smoothing and its applications*. Chapman and Hall/CRC, 2018.

[24] J. E. Chacón and T. Duong. Higher order differential analysis with vectorized derivatives. *arXiv preprint arXiv:2011.01833*, 2020.

[25] S.-B. Choi and J. K. Hedrick. Robust throttle control of automotive engines: Theory and experiment. *Journal of Dynamic Systems, Measurement, and Control*, 118(1):92–98, 1996-03. eprint: https://asmedigitalcollection.asme.org/dynamicsystems/article-pdf/118/1/92/5595525/92_1.pdf.

[26] D. S. Clark. Short proof of a discrete gronwall inequality. *Discrete applied mathematics*, 16(3):279–281, 1987. Publisher: Elsevier.

[27] W. Commons. Silverstone circuit, 13/12/2020.

[28] V. Cossalter, M. D. Lio, R. Lot, and L. Fabbri. A general method for the evaluation of vehicle manoeuvrability with special emphasis on motorcycles. *Vehicle System Dynamics*, 31(2):113–135, 1999. eprint: https://www.tandfonline.com/doi/pdf/10.1076/vesd.31.2.113.2094.

[29] C. L. Darby, W. W. Hager, and A. V. Rao. An hp-adaptive pseudospectral method for solving optimal control problems. *Optimal Control Applications and Methods*, 32(4):476–502, 2011. Publisher: Wiley Online Library.

[30] P. Dayan. Improving generalization for temporal difference learning: The successor representation. *Neural Computation*, 5(4):613–624, 1993.

[31] C. Dorao and H. Jakobsen. hp-adaptive least squares spectral element method for population balance equations. *Applied Numerical Mathematics*, 58(5):563–576, 2008. Publisher: Elsevier.

[32] H. Dugoff, P. S. Fancher, and L. Segel. Tire performance characteristics affecting vehicle response to steering and braking control inputs, 1969.

[33] G. Elnagar, M. A. Kazemi, and M. Razzaghi. The pseudospectral legendre method for discretizing optimal control problems. *IEEE Transactions on Automatic Control*, 40(10):1793–1796, 1995. Publisher: IEEE.

[34] G. N. Elnagar and M. Razzaghi. A collocation-type method for linear quadratic optimal control problems. *Optimal Control Applications and Methods*, 18(3):227–235, 1997. Publisher: Wiley Online Library.

[35] F. G. Frobenius. Über lineare substitutionen und bilineare formen. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1878(84):1–63, 1878. Publisher: De Gruyter.

[36] F. Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P. Dürr. Super-human performance in gran turismo sport using deep reinforcement learning. *IEEE Robotics and Automation Letters*, 6(3):4257–4264, 2021. Publisher: IEEE.

[37] D. Garg, M. Patterson, W. W. Hager, A. V. Rao, D. A. Benson, and G. T. Huntington. A unified framework for the numerical solution of optimal control problems using pseudospectral methods. *Automatica*, 46(11):1843–1851, 2010. Publisher: Elsevier.

[38] D. Garg, M. A. Patterson, C. Francolin, C. L. Darby, G. T. Huntington, W. W. Hager, and A. V. Rao. Direct trajectory optimization and costate estimation of finite-horizon and infinite-horizon optimal control problems using a radau pseudospectral method. *Computational Optimization and Applications*, 49(2):335–358, 2011. Publisher: Springer.

[39] T. H. Gronwall. Note on the derivatives with respect to a parameter of the solutions of a system of differential equations. *Annals of Mathematics*, pages 292–296, 1919. Publisher: JSTOR.

[40] R. Guntur and S. Sankar. A friction circle concept for dugoff's tyre friction model. *International Journal of Vehicle Design*, 1(4):373–377, 1980. Publisher: Inderscience Publishers.

[41] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and others. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[42] J. Hendrikx, T. Meijlink, and R. Kriens. Application of optimal control theory to inverse simulation of car handling. *Vehicle System Dynamics*, 26(6):449–461, 1996. Publisher: Taylor & Francis.

[43] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines, 2018. Publication Title: GitHub repository.

[44] C. M. Hoeppke. "colloptpy, a python based direct collocation solver", 2022.

[45] C. M. Hoeppke. "pyracetrack, a python based implementation of a 2d racetrack model", 2022.

[46] C. M. Hoeppke. "racegame2d, python implementation of a 2d racing game", 2022.

[47] C. M. Hoeppke. "racegym2d, gym environments for lap time reinforcement learning", 2022.

[48] R. Z. Horace He. functorch: JAX-like composable function transforms for PyTorch, 2021.

[49] A. S. a. Jack Macki. *Introduction to Optimal Control Theory*. Undergraduate Texts in Mathematics. Springer, 1 edition, 1982.

[50] S. Jain and P. Tsiotras. Trajectory optimization using multiresolution techniques. *Journal of Guidance Control and Dynamics - J GUID CONTROL DYNAM*, 31, 2008.

[51] J. Kabzan, M. I. Valls, V. J. Reijgwart, H. F. Hendrikx, C. Ehmke, M. Prajapat, A. Bühler, N. Gosala, M. Gupta, R. Sivanesan, and others. Amz driverless: The full autonomous racing system. *Journal of Field Robotics*, 37(7):1267–1294, 2020. Publisher: Wiley Online Library.

[52] S. Kang and J. B. Cheek. *Numerical solution of differential equations*. Waterways Experiment Station, 1972.

[53] J. Katz. Aerodynamics of race cars. *Annu. Rev. Fluid Mech.*, 38:27–63, 2006. Publisher: Annual Reviews.

[54] D. P. Kelly. Lap time simulation with transient vehicle and tyre dynamics, 2008.

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

[56] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A LLVM-based python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15. Association for Computing Machinery, 2015. event-place: Austin, Texas.

[57] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. Publisher: Springer Science and Business Media LLC.

[58] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[59] J. R. Magnus and H. Neudecker. *Matrix differential calculus with applications in statistics and econometrics*. John Wiley & Sons, 2019.

[60] R. Meise and D. Vogt. *Introduction to functional analysis*. Clarendon Press, 1997.

[61] W. Milliken and D. Milliken. Race car dynamics. *SAE, Warrendale*, 1995.

[62] W. E. Milne and W. Milne. *Numerical solution of differential equations*, volume 19. Wiley New York, 1953.

[63] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[64] Y. Nakatsukasa and A. Townsend. Error localization of best l1 polynomial approximants, 2019. eprint: 1902.02664.

[65] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[66] T. Novi, A. Liniger, R. Capitani, and C. Annicchiarico. Real-time control for at-limit handling driving on a predefined path. *Vehicle system dynamics*, 2019. Publisher: Taylor & Francis.

[67] C. J. Ostafew, A. P. Schoellig, and T. D. Barfoot. Robust constrained learning-based NMPC enabling reliable mobile robot path tracking. *The International Journal of Robotics Research*, 35(13):1547–1563, 2016. Publisher: SAGE Publications Sage UK: London, England.

[68] H. B. Pacejka. Modelling of the pneumatic tyre and its impact on vehicle dynamic behaviour, vehicle research laboratory. *Delft University of Technology, The Netherlands*, 1989.

[69] H. B. Pacejka, editor. *Tire and vehicle dynamics*. Butterworth-Heinemann, Oxford, third edition edition, 2012.

[70] H. B. Pacejka and E. Bakker. The magic formula tyre model. *Vehicle System Dynamics*, 21:1–18, 1992. Publisher: Taylor & Francis.

[71] H. B. Pacejka and R. S. Sharp. Shear force development by pneumatic tyres in steady state conditions: a review of modelling aspects. *Vehicle system dynamics*, 20(3):121–175, 1991. Publisher: Taylor & Francis.

[72] Y. Pan, C.-A. Cheng, K. Saigol, K. Lee, X. Yan, E. Theodorou, and B. Boots. Agile autonomous driving using end-to-end deep imitation learning. *arXiv preprint arXiv:1709.07174*, 2017.

[73] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. 2017.

[74] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[75] G. Perantoni and D. J. Limebeer. Optimal control for a formula one car with variable parameters. *Vehicle System Dynamics*, 52(5):653–678, 2014. Publisher: Taylor & Francis.

[76] W. Pittenger. Bahrain international circuit–grand prix layout, 2021.

[77] W. Pittenger. Barcelona circuit–grand prix layout „catalunya (2004-2006)“, 2022.

[78] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network, 1989.

[79] R. Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[80] U. Rosolia and F. Borrelli. Learning how to autonomously race a car: a predictive control approach. *IEEE Transactions on Control Systems Technology*, 28(6):2713–2719, 2019. Publisher: IEEE.

[81] I. M. Ross and F. Fahroo. Convergence of pseudospectral discretizations of optimal control problems. In *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No. 01CH37228)*, volume 4, pages 3175–3177. IEEE, 2001.

[82] H. Scherenberg. Mercedes-benz racing cars - design and experience. *SAE Transactions*, pages 414–420, 1958. Publisher: JSTOR.

[83] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[84] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[85] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[86] P. Sermanet, S. Chintala, and Y. LeCun. Convolutional neural networks applied to house numbers digit classification, 2012. eprint: 1204.3968.

[87] P. Shinners. PyGame, 2011.

[88] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *International Conference on Machine Learning*, pages 387–395. PMLR, 2014.

[89] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988. Publisher: Springer.

[90] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[91] T. Tao. *An introduction to measure theory*, volume 126. American Mathematical Society Providence, 2011.

[92] D. Tavernini, M. Massaro, E. Velenis, D. I. Katzourakis, and R. Lot. Minimum time cornering: the effect of road surface and car transmission layout. *Vehicle System Dynamics*, 51(10):1533–1547, 2013. Publisher: Taylor & Francis.

[93] B. G. Teubner. *Vorlesungen über Variationsrechnung*, volume 22-23. 11 2021.

[94] R. Verschueren, S. De Bruyne, M. Zanon, J. V. Frasch, and M. Diehl. Towards time-optimal race car driving using nonlinear MPC in real-time. In *53rd IEEE conference on decision and control*, pages 2505–2510. IEEE, 2014.

[95] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 17:261–272, 2020.

[96] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992. Publisher: Springer.

[97] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992. Publisher: Springer.

[98] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2005. Publisher: Springer Science and Business Media LLC.

[99] Y. Zhao and P. Tsiotras. Density functions for mesh refinement in numerical optimal control. *Journal of Guidance, Control, and Dynamics*, 34, 2011.

Appendix A

Training configurations

Table A.1: Baseline training configurations

	Value		
Parameter	Figure 4.4a	Figure 4.6	Figure 4.7
Initial learning rate α_0	$\{2^{-9}, \ldots, 2^{-15}\}$	$\{2^{-9}, \ldots, 2^{-15}\}$	$\{2^{-9}, \ldots, 2^{-15}\}$
Training steps	4000000	4000000	4000000
Save frequency	20000	20000	20000
Network architecture	4×128	4×128	4×128
Rollout length	512	512	512
Training batch size	256	256	256
Training epochs	1	1	1
Discount factor γ	0.99	0.99	0.99
GAE factor λ	0.95	0.95	0.95
Environment type	"base"	"base"	"base"
Track	**"L-Shape"**	**"S-Double"**	**"L"** (4.7a), **"Double-S"** (4.7b)
Algorithm	"PPO"	"PPO"	"PPO"
$N_{\text{environments}}$	24	24	24
Time step δ_t	0.1	0.1	0.1
Maximal time $T_{\max}$	300.0	300.0	300.0
Checkpoint rewards	false	false	**true**
Show absolute states	true	true	true
Use discrete actions	false	false	false
Model name	"Bicycle"	"Bicycle"	"Bicycle"
Allow multiple laps	false	false	false
Time penalty	0.0	0.0	0.0
Learning rate schedule	"Harmonic"	"Harmonic"	"Harmonic"
Learning rate decay β	10^{-6}	10^{-6}	10^{-6}
Show track variables	true	true	true

Table A.2: Ray casting environment configuration for Figures 4.12a, 4.12b

Parameter	Value
Initial learning rate α_0	2^{-13}
Training steps	4000000
Save frequency	20000
Network architecture	$(128, 128, 128, 128)$
Rollout length	512
Training batch size	256
Training epochs	1
Discount factor γ	0.99
Environment type	**"rays"**
Training track	"L" track (Figure 4.12a), "Double-S" Track (Figure 4.12b)
Algortihm	"PPO"
$N_{\text{environments}}$	24
N_{rays}	$\{16, 32, 64, 128\}$
Time step δ_t	0.1
Maximal time $T_{\max}$	300.0
Checkpoint distance	10.0
Checkpoint rewards	true
Show absolute states	true
Use discrete actions	false
Physical model	Bicycle
Allow multiple laps	false
Time penalty	0.0
Learning rate schedule	"Harmonic"
Learning rate decay β	10^{-6}
Show track variables	true

Table A.3: Visual environment training configurations

Parameter	Values for Figure 4.16	Values for Figures 4.19 and 4.20	Values for Figure 4.21
Initial learning rate α_0	$\{5e-4, \ldots, 1e-5\}$	$\{5e-4, \ldots, 1e-5\}$	10^{-5}
Training steps	4000000	4000000	4000000
Save frequency	20000	20000	20000
Rollout length	512	512	512
Training batch size	256	256	256
Training epochs	1	1	1
Discount factor γ	0.99	0.99	0.99
Environment type	**"visual"**	**"visual"**	**"visual"**
Training track	Bahrain 4.2a	Bahrain 4.2a	Bahrain 4.2a
Algortihm	"PPO"	"PPO"	"PPO"
$N_{\text{environments}}$	24	24	24
N_{frames}	1	4	$\{1, \ldots, 8\}$
Image height	$\{60, 90, 120\}$	60	60
Visual layers	2	2	2
Kernel size k_1	5×5	5×5	5×5
Kernel size k_2	5×5	5×5	5×5
Time step δ_t	0.1	0.1	0.1
Maximal time $T_{\max}$	300.0	300.0	300.0
Checkpoint distance	10.0	10.0	10.0
Checkpoint rewards	true	true	true
Show absolute states	true	true	true
Use discrete actions	false	false	false
Physical model	Bicycle	Bicycle	Bicycle
Allow multiple laps	true	true	true
Time penalty	0.0	0.0	0.0
Learning rate schedule	"Harmonic"	"Harmonic"	"Harmonic"
Learning rate decay β	10^{-6}	10^{-6}	10^{-6}
Show track variables	true	true	true

Table A.4: Combined environment training configuration

Parameter	Values used to create Figure 4.23
Initial learning rate α_0	3×10^{-4}
Training steps	25×10^6
Save frequency	50_000
Rollout length	512
Training batch size	256
Training epochs	1
Discount factor γ	0.99
Environment type	**"combined"**
Training track	Bahrain, Baku, Imolah, Jedda, Melbourne, Miami, Montreal, Silverstone
Algortihm	"PPO"
$N_{\text{environments}}$	24
N_{rays}	16
N_{frames}	1
Image height	60
Visual layers	3
Kernel size k_1	5×5
Kernel size k_2	5×5
Kernel size k_3	5×5
Time step δ_t	0.1
Maximal time $T_{\max}$	300.0
Checkpoint distance	10.0
Checkpoint rewards	true
Show absolute states	true
Use discrete actions	false
Physical model	Bicycle
Allow multiple laps	true
Time penalty	0.0
Learning rate schedule	"Harmonic"
Learning rate decay β	10^{-6}
Show track variables	true