Universiteit Leiden
Opleiding Informatica

Autonomous Simulated Car Racing

through

Apprenticeship Learning

Name: Derk Mus

Date: 24/08/2015

1st supervisor: Dr. W.A. Kosters
2nd supervisor: Dr. L.P.J. Groenewegen

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Autonomous Simulated Car Racing through Apprenticeship Learning

Derk Mus

Abstract

Many problems in robotics have unknown, high-dimensional and highly non-linear dynamics. One of that problems is robotic car racing in a very realistic simulator. The key difficulties that arise are: i) It is difficult to build a good dynamics model. ii) It is often difficult to write down a formal specification of the task that has to be performed. iii) It is computationally expensive to find closed-loop controllers for this high-dimensional domain.

We use algorithms from the apprenticeship learning setting — the setting when expert demonstrations of the task are available — in order to solve these problems for the task of autonomous car racing for a robotic car in the Simulated Car Racing Championship. This results in a basic dynamics model that is able to capture most important aspects of the dynamics of the car. We show that an advanced inverse reinforcement learning algorithm is not robust enough to work in this complex domain. Furthermore we use a reinforcement learning algorithm with the dynamics model built and a hand-engineered spefication of a basic racing task to build a controller for the racing car.

Contents

Acknowledgements

I would like to thank Pieter Abbeel for his comments on implementing the iterative LQR algorithm, Albert Diosi for his help on using a scan matching algorithm and Sergey Levine for his useful feedback on implementing an Inverse Reinforcement Learning algorithm. Next, I would like to thank dr W.A. Kosters for his feedback, time and effort.

Chapter 1

Introduction

This work focuses on controlling a robotic car in the *Simulated Car Racing Championship*. This is an international competition where the goal is to design a controller for a racing car. This racing car will compete on a set of unknown tracks, first alone (against the clock) and then against other drivers. The championship is built on top of a realistic state-of-the-art car racing simulator, which makes controlling a car in this simulator a challenging and complex task. To learn a robotic car the task of autonomous racing we have to consider several learning strategies. Atkeson proposed general learning strategies in [1]:

- *Learning by being taught.* A teacher (or programmer) tells the robot explicitly what to do in various situations by teaching or programming rules.
- *Learning by imitation.* The robot learns by observing (human) experts doing the task or related tasks.
- *Learning by thinking or dreaming.* The robot learns the task by planning or mentally practicing the task. While doing the task, the most appropriate plan is chosen and modified to the current situation.
- *Learning by doing.* The robot learns from practicing the task.

For the task of autonomous racing we focus on a combination of imitation learning and learning by thinking, called *apprenticeship learning*. These strategies seem more appropriate than the other strategies. Explicitly telling the robot what to do (being taught or programmed) can work well in some situations, but it is much too hard to describe rules for all possible situations in complex environments. Learning by doing without any prior knowledge will probably require a huge amount of trials, before a good policy has been learned [2]. However, this strategy can be useful to optimize task performance by performing real trials (practising).

1.1 Apprenticeship learning

Apprenticeship learning [3], also called learning from demonstrations, is a form of indirect imitation learning. For imitation learning, we distinguish *direct* and *indirect* approaches [4]. With direct imitation learning, the robot does not make any assumptions about the intent of the expert demonstrating the task. It only tries to mimic the expert's behaviour when a similar situation occurs. Therefore, this approach is more useful for "reactive" tasks, where no planning for future actions is involved.

With indirect imitation learning or apprenticeship learning, the goal is to find the expert's intent or the objective the expert is trying to optimize in its demonstrations. Often it is much easier for an expert to demonstrate a task than to formulate this in a form understandable for an apprentice (how to formulate "driving well"?) [3, 5]. The expert's intention can be viewed as a formal description of the task. Then this description can be used for planning actions (learning by thinking).

For planning actions two key elements are required. Beside the goal we want to achieve, a robot requires knowledge about how actions affect the current situation (how the system evolves over time). This knowledge is called a dynamics model. With such a model, intuitively, a robot can simulate

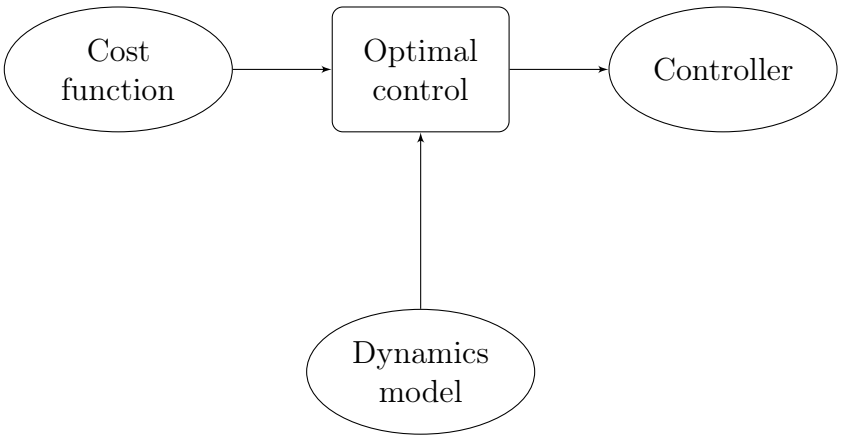


Figure 1.1: Typical optimal control or model-based reinforcement learning method.

multiple possible experiences and choose the behaviour that performs best in simulation. To learn the dynamics model, a robot can use the expert's and its own demonstrations or the physical principles could be taught (programmed) by a teacher. A combination is possible as well.

To summarize, with apprenticeship learning a robot can leverage demonstrations from an expert to learn a dynamics model and a formal task description. To describe these elements we use the formalism of Markov Decision Processes (MDPs), which are described in Chapter 2. To learn a control policy by thinking we make use of model-based reinforcement learning [6] algorithms. We note that the fields of model-based reinforcement learning and optimal control are similar. Methods in these fields typically work in the same way. As input these methods need a dynamics model of the system and a cost or reward function that describes what is good or bad behaviour (see Figure 1.1). Then the method outputs a control strategy that attempts to minimize the cost in the dynamical system.

1.2 Contributions

The main contributions of this work are: (i) modeling the dynamics of a car in the Simulated Car Racing Championship, (ii) learning a task specification for several racing tasks using apprenticeship learning algorithms and (iii) combining (i) and (ii) with existing reinforcement learning algorithms to design autonomous racing controllers. Here we focus on maximizing performance in the *real* system and not in the system modeled by (i).

1.3 Related work

There are several works that relate to components of our research. Most related is Abbeel's work using apprenticeship learning for the real-world task of acrobatic helicopter flight [7, 8]. Unlike Abbeel, we are, as well as [9], not particulary interested in learning to follow a pre-supplied or learned reference trajectory. We are interested in optimizing performance relative to a high level objective, such as racing towards the finish line. This means that we do not have a separate trajectory planner and then a controller that penalizes for deviation along that trajectory. In our approach the controller has to solve the entire problem and so we aim to obtain a controller that not blindly follows a trajectory.

Apprenticeship learning in various forms is also used to learn a four-legged robot to climb across challenging terrains [10] and for parking lot navigation [11].

If we look at the task of autonomous driving the most famous work is probably the Google driverless car [12]. This car has successfully completed thousands of miles of driving without human intervention. This car uses much of the work developed for Stanford's autonomous research vehicle Junior, Stanford's entry in the 2007 DARPA Urban Challenge [13]. This car uses a lot of sophisticated algorithms for aspects that including mapping of the environment, localization, trajectory planning, dynamical modeling

and control. The goal of the control system is to take the upcoming planned trajectory and generate the inputs (throttle/braking, steering) in order to follow this trajectory. The difference with our work is that we do not use an advanced trajectory planner, but let the controller plan a trajectory itself.

Also research into the task of autonomous racing in a real challenging environment is performed with performance only slightly less than a skilled human driver [14]. Here different aspects are used to compute a trajectory, which a controller then has to follow.

1.4 Outline

Background material on model-based reinforcement learning and MDPs is presented in Chapter 2. We describe the iterative LQR algorithm to find a control policy for an MDP in Chapter 3 and the Simulated Car Racing Championship (SCRC) in Chapter 4. The state and action spaces of the MDP we model for the SCRC can be found in Chapter 5 and the dynamics model in Chapter 7. We focus on the reward function in Chapter 8. Experiments are mentioned in Chapter 9. Some extra work about a state representation using unsupervised learning is presented in Chapter 6. Conclusions and possibilities for future work are given in Chapter 10.

Chapter 2

Background

We present a basic background on (model-based) reinforcement learning and the formalism of MDPs (for a broader introduction, see [15]). We will focus on those algorithms that we will later build upon.

2.1 States and controls

With a *state* we describe the current situation of the system in which the robot is acting. We mean the set of those elements of the system in which the agent is acting that are relevant to the system's change over time. These elements are typically *not* directly specified by the agent. For instance, for a robotic car, the state may consist of the robot's position, orientation plus the velocities and angular velocities. To denote the state of the system, we will use the notation

$$s \in \mathbb{R}^n, \tag{2.1}$$

where n is the dimension of the state space. We will use a discrete-time setting, where time is discretized in short intervals. At each timestep t the state of the system is defined by a point s_t in the n-dimensional space.

Controls (also called actions) specify those elements that the agent can

specify directly and with which the agent can act upon the system. For the car, the controls could correspond to the gas pedal, brake pedal and steering wheel. To denote controls, we will use the notation

$$u \in \mathbb{R}^m, \tag{2.2}$$

where m is the dimension of the control space. Thus, there are m controls the agent can use at each time t to act upon the system.

2.2 First-order Markov models

The state of the system in which the agent is acting evolves as a function of past states and control inputs. Starting from state s_t at time t, the next state, denoted as s_{t+1}, is specified as a function of previous states and controls:

$$s_{t+1} = f(s_t, u_t, s_{t-1}, u_{t-1}, \dots). \tag{2.3}$$

However, such general models can be quite cumbersome, because the next state could depend on a long history of controls. In the control literature, it is common to restrict to *first-order Markov models*, where the next state depends only on the current state and control:

$$s_{t+1} = f(s_t, u_t). \tag{2.4}$$

In a first-order Markov model, the state and control input contain everything needed to determine the next state of the system, where the controls and states respectively represent those quantities that the agent can and cannot directly affect. With a model, the agent does have some knowledge about how possible actions affect the state of the system. However, in some cases s_t and u_t do not give enough information to predict the next state s_{t+1} and we do need also previous states $(s_{t-1}, \dots, s_{t-k})$ and control inputs $(u_{t-1}, \dots, u_{t-k})$.

To keep a first-order Markov model we can augment the state s_t at time t with these k previous states and control inputs. The parameter k indicates the number of past states and control inputs that we need.

2.3 Stochastic models

In practice we do not have perfect knowledge of the current state and control for complex systems. Our parametrization of the state is typically incomplete: we choose some representation for the state so that most of the system's evolution can be described using a first-order Markov model, but there are almost always certain elements that affect the system evolution to some degree that we cannot observe or do not want to include in the model. In a car, for instance, this could be the fuel level, tire wear or airflow around the car. However, this is hard to measure and has a relatively negligible effect on the system. Thus, we can instead add some amount of stochasticity to account for unmodeled effects. Stochasticity can be represented by a model of the form

$$s_{t+1} = f(s_t, u_t) + \epsilon_t, \tag{2.5}$$

where ϵ_t denotes some zero-mean noise term, such as a Gaussian random variable with covariance Σ:

$$\epsilon_t = \mathcal{N}(0, \Sigma). \tag{2.6}$$

2.4 Control policies

A control policy (also referred to as a controller) is an entity that chooses control actions based on the current state. Formally, we define a policy

$$\pi : \mathbb{R}^n \mapsto \mathbb{R}^m \tag{2.7}$$

as a mapping from states to actions. When state spaces are continuous, we work with parameterized policies, where the policy function is specified by a set of parameters. A common example is a linear policy, where the controls are prescribed as a linear function of the state

$$\pi(s_t) = K\ s_t \tag{2.8}$$

with $K \in \mathbb{R}^{m \times n}$ a matrix of the policy parameters. Another common form of policies are those linear in *state features*

$$\pi(s_t) = K\ \phi(s_t) \tag{2.9}$$

where $\phi : \mathbb{R}^n \mapsto \mathbb{R}^p$ is a function mapping from states to some p-dimensional feature space and $K \in \mathbb{R}^{m \times p}$ is again a matrix of parameters.

2.5 Optimal control

To determine how to choose a policy that can control an agent in some environment, it is necessary to specify what is "good" behaviour in that environment. The typical method for doing this is with a *reward* or *cost* function, that specifies the desirability for being in a certain state and taking a certain control action in that state (some control actions could have a higher cost than others). We could consider a reward function as a description of the task that has to be learned. Reward functions are specified as a mapping from states and controls to a real number:

$$R : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}. \tag{2.10}$$

The goal of a control policy is then to maximize the *expected sum of reward* over some *time horizon* H.[1] Formally, the *value function* for a policy π and state s starting at time t is defined as the sum of expected rewards, starting at time t in state s and acting according to policy π

$$V_t^\pi(s) = \mathbb{E}\left[\sum_{t'=t}^{t+H} R(s_{t'}, \pi(s_{t'}))\middle| s_{t'+1} = f(s_{t'}, \pi(s_{t'})) + \epsilon_{t'}, s_t = s\right]. \quad (2.11)$$

The value function V_t^π satisfies a recurrence relation known as *Bellman's equation*

$$V_t^\pi(s) = R(s, \pi(s)) + \mathbb{E}[V_{t+1}^\pi(f(s, \pi(s))) + \epsilon]. \quad (2.12)$$

This equation says that the value of a state is equal to the reward of that state and the subsequent action, plus the expected value of the next state, following policy π.

The *optimal policy* $\pi^\star$ is the policy that maximizes the value function over all possible policies. The value function for this optimal policy, sometimes referred to as *Bellman's optimality equation*, is defined as follows:

$$V_t^\star(s) = \max_u \left[R(s, u) + \mathbb{E}[V_{t+1}^\star(f(s, u)) + \epsilon]\right]. \quad (2.13)$$

2.6 Markov Decision Processes

In reinforcement learning, often the formalism of *Markov Decision Processes* (MDPs) [6] is used. Intuitively, an MDP is simply a structure that contains all the elements described previously: state and control spaces, a dynamics model (also called transition probabilities), a reward function (or cost function, which corresponds to negative reward), a time horizon and a distribution over initial states. Formally, an MDP is a tuple $M = (S, U, T, R, H, D)$ where S and U are the state and control spaces, T is a set of transition probabilities given

[1] In the case of a cost function, we want to minimize the expected sum of cost over some time horizon H.

by the dynamics model, D is a distribution over initial states, H is the time horizon and R is the reward function. There are many different formulations for MDPs, such as those with an infinite (often discounted) value function instead of the finite time horizon described above.

Chapter 3

Iterative Linear-Quadratic Regulator

Linear Quadratic Regulator (LQR) control is based upon a special case of dynamics and costs (negative reward) for continuous state and action spaces. For MDPs which meet these requirements, the solution to Bellman's optimality equation can be derived analytically [16].

3.1 Linear dynamics and quadratic costs

The LQR setting assumes the dynamics of an MDP obey

$$s_{t+1} = As_t + Bu_t \tag{3.1}$$

with s_t and u_t the state and control input at time t; $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ [16]. Furthermore it assumes a cost function[1] in the form:

$$C(s, u) = s^T Q s + u^T R u \tag{3.2}$$

[1] A cost function can be considered as a negative reward function.

with $Q \in \mathbb{R}^{n\times n} \succeq 0$ and $R \in \mathbb{R}^{m\times m} \succ 0$.[2] A cost function in this form is always convex, which means that there is only one minimum. The cost function shows that the goal is to maintain at the all-zeros state, $s = 0$, while applying zero control, $u = 0$.

For this special class of MDPs, Bellman's optimality equation could be solved analytically to find the optimal value function [16]. We initialize the value function at time H as

$$V_H^\star(s) = s^T P_H s. \tag{3.3}$$

Here we can choose P_H to penalize deviations from the all-zeros state at the end. Then the value function at time $H-1$ is

$$\begin{aligned} V_{H-1}^\star(s) &= \min_u [s^T Q s + u^T R u + V_H^\star(As + Bu)] \\ &= \min_u [s^T Q s + u^T R u + (As + Bu)^T P_H (As + Bu)]. \end{aligned} \tag{3.4}$$

To find the minimum over u, we set the gradient with respect to u to 0. This yields

$$\nabla(\ldots)_u = 2Ru + 2B^T P_H (As + Bu) = 0. \tag{3.5}$$

Solving this equation gives

$$u = -(R + B^T P_H B)^{-1} B^T P_H A x. \tag{3.6}$$

Substituting this expression back into equation (3.4) gives a quadratic equation

$$V_{H-1}^\star(s) = s^T P_{H-1} s \tag{3.7}$$

[2]Here $X \succ 0$ and $X \succeq 0$ denote positive definite and positive semidefinite matrices respectively. For a square matrix X we have $X \succ 0$ if and only if for all vectors z we have $z^T X z > 0$. We have $X \succeq 0$ if and only if for all vectors z we have $z^T X z \geq 0$. Hence any state and any input different from the all-zeros state or all-zeros input yields a non-zero cost.

for

$$P_{H-1} = Q + K_{H-1}^T R K_{H-1}^T + (A + BK_{H-1})^T P_H (A + BK_{H-1})$$

and

$$K_{H-1} = -(R + B^T P_H B) - 1 B^T P_H A.$$

This shows that V_{H-1} is quadratic, just like V_H. This update is the same for all times $t = H, H-1, \ldots, 1, 0$. Thus in general, the solution to an LQR control problem is as follows:

- Initialize P_H
- For $t = H-1, \ldots, 0$:

$$\begin{aligned} K_t &= -(R + B^T P_{t+1} B) - 1 B^T P_{t+1} A \\ P_t &= Q + K_t^T R K_t^T + (A + BK_t)^T P_{t+1} (A + BK_t) \end{aligned} \tag{3.8}$$

Then the optimal policy at time t is given by

$$\pi(s) = K_t s \tag{3.9}$$

and the value function at time t is given by

$$V_t(s) = s^T P_t s. \tag{3.10}$$

This policy tries to keep the system with linear dynamics and quadratic cost at the all-zeros state, while preferring to keep the control input small. The restrictions on the dynamics and cost function may seem restrictive, but the algorithm can be extended to certain more general settings:

- The algorithm can be generalized to *time-varying* dynamics and costs:

$$s_{t+1} = A_t s_t + B_t u_t, \quad C_t(s, u) = s^T Q_t s + u^T R_t u.$$

The difference is that the dynamics and costs can now be different for each time t. The derivation is identical, so the resulting equations for K_t and P_t have the form:

$$K_t = -(R_t + B_t^T P_{t+1} B_t) - 1 B_t^T P_{t+1} A_t$$
$$P_t = Q_t + K_t^T R_t K_t^T + (A_t + B_t K_t)^T P_{t+1}(A_t + B_t K_t).$$

- LQR can be applied to affine systems

$$s_{t+1} = A_t s_t + B_t u_t + a$$

by adding a constant term to the state and applying LQR to the augmented system

$$\bar{s} = \begin{bmatrix} s \\ 1 \end{bmatrix}, \quad \bar{A} = \begin{bmatrix} A & a \\ 0 & 1 \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} B \\ 0 \end{bmatrix}, \quad \bar{Q} = \begin{bmatrix} Q & 0 \\ 0 & 0 \end{bmatrix}, \quad \bar{R} = R.$$

3.2 Nonlinear dynamics

The constraint of linear models is a large restriction. However, the algorithm can be extended (approximately) to nonlinear dynamics. A general nonlinear dynamics model has the form

$$s_{t+1} = f(s_t, u_t). \tag{3.11}$$

To apply LQR to this setting we *linearize* around a reference state-input trajectory $s_0^\star, u_0^\star, s_1^\star, u_1^\star, \ldots, s_H^\star, u_H^\star$. Linearization means finding a linear approximation to a function at a given point. We need the reference trajectory for the initial linearizations of the algorithm. We linearize around each point in the state-input trajectory to obtain linear, time-varying dynamics. In practice, a linearized model is typically a good approximation to a nonlinear dynamics model for regions "close" to the point where we linearized. To obtain the

matrices A_t and B_t for all time steps t, we linearize the dynamics around $s_t^\star$ for all t with a first-order Taylor expansion

$$s_{t+1} \approx f(s_t^\star, u_t^\star) + \frac{\partial f}{\partial s}(s_t^\star, u_t^\star)(s_t - s_t^\star) + \frac{\partial f}{\partial u}(s^{t\star, u^{t\star})(u_t - u_t^\star)}. \quad (3.12)$$

The partial derivatives can be approximated with a finite difference method. The approximate matrices are

$$A_t \approx \frac{\partial f}{\partial s}(s_t^\star, u_t^\star), \quad B_t \approx \frac{\partial f}{\partial u}(s_t^\star, u_t^\star) \quad (3.13)$$

for all t. Now instead of equation (3.12) we can write

$$s_{t+1} - s_{t+1}^\star = A(s_t - s_t^\star) + B(u_t - u_t^\star). \quad (3.14)$$

Here we assume the trajectory we linearized around is a realizable trajectory, thus $s_{t+1}^\star = f(s_t^\star, u_t^\star)$. We get a time-varying LQR format if using

$$z_t = s_t - s_t^\star, \quad v_t = u_t - u_t^\star \quad (3.15)$$

Then we have

$$z_{t+1} = A_t z_t + B_t v_t \quad (3.16)$$

and a non time-varying cost function

$$C(z, v) = z_t^T Q z_t + v_t^T R v_t. \quad (3.17)$$

This is an LQR problem that can be solved to obtain a policy $v_t = K z_t$. Because

$$v_t = u_t - u_t^\star = K(s_t - s_t^\star), \quad (3.18)$$

the optimal control input u_t for time t can be computed with

$$u_t = u_t^\star + K(s_t - s_t^\star). \quad (3.19)$$

3.3 Iterative LQR

The method described above is still limited, because it requires as initialization a realizable state-input trajectory. A generic optimal control problem has the form

$$\min_{u_0,\ldots,u_H} \sum_{t=0}^{H} C(s_t, u_t) \tag{3.20}$$

subject to $s_{t+1} = f(s_t, u_t)$ for all t. This problem can be solved by iteratively approximating it and using the fact that the LQR formulation is easy to solve. The iterative LQR algorithm is as follows [16]:

1. Pick either (a) an (arbitrary) control policy $\pi^{(0)}$ or (b) an (arbitrary) sequence of states $s_0^{(0)}, s_1^{(0)}, \ldots, s_H^{(0)}$ and control inputs $u_0^{(0)}, u_1^{(0)}, \ldots, u_H^{(0)}$. With (a) go to Step (2), with (b) go to Step (3).

2. Execute the current policy $\pi^{(i)}$ and record the resulting state-input trajectory $s_0^{(0)}, u_0^{(0)}, s_1^{(0)}, u_1^{(0)}, \ldots, s_H^{(0)}, u_H^{(0)}$.

3. Compute the approximation for the matrices A_t, B_t, Q_t and R_t around the obtained state-input trajectory by computing a first-order Taylor expansion of the dynamics model. The Q_t and R_t matrices can be approximated by two second order Taylor expansion. Here we assume a cost function $C(s_t, u_t) = g(s_t) + h(u_t)$.

4. Use the approximations from the previous step to solve for the optimal control policy $\pi^{(i+1)}$.

5. Set $i = i + 1$ and go to Step (2).

With the approximations we obtain a standard time-varying linear system $z_{t+1} = A_t z_t + B_t v_t$ with quadratic cost $z_t^T Q_t z_t + v_t^T R_t v_t$. The linearization around $(s_t^{(i)}, u_t^{(i)})$ in iteration i of the algorithm gives us

$$s_{t+1} \approx f(s_t^{(i)}, u_t^{(i)}) + \frac{\partial f}{\partial s}(s_t^{(i)}, u_t^{(i)})(s_t - s_t^{(i)}) + \frac{\partial f}{\partial u}(s^{(i)}, u^{(i)})(u_t - u_t^{(i)}).$$

Subtracting the term $s_t^{(i)}$ on both sides gives the format we want. Hence we get the standard format if we use

$$\begin{aligned} z_t &= \begin{bmatrix} s_t - s_t^{(i)} \\ 1 \end{bmatrix} \\ v_t &= u_t - u_t^{(i)} \\ A_t &= \begin{bmatrix} \frac{\partial f}{\partial s}(s_t^{(i)}, u_t^{(i)}) & f(s_t^{(i)}, u_t^{(i)}) - s_t^{(i)} \\ 0 & 1 \end{bmatrix} \\ B_t &= \begin{bmatrix} \frac{\partial f}{\partial u}(s^{(i)}, u^{(i)}) \\ 0 \end{bmatrix}. \end{aligned}$$

The derivations for Q_t and R_t are somewhat different. With the second-order Taylor expansion around $s_t^{(i)}$ to approximate $g(s_t)$, we get

$$g(s_t) \approx j_0 + (s_t - s_t^{(i)})^T j + \frac{1}{2}(s_t - s_t^{(i)})^T J (s_t - s_t^{(i)}) \tag{3.21}$$

where $j_0 = g(s_t^{(i)})$, $j = \frac{\partial g}{\partial s}(s_t^{(i)})$ and $J = \frac{\partial^2 g}{\partial s^2}(s_t^{(i)})$. This equation is not in quadratic form, but can be written as

$$z_t^T \begin{bmatrix} \frac{1}{2}J & \frac{1}{2}j \\ \frac{1}{2}j^T & \frac{1}{2}j_0 \end{bmatrix} z_t = z_t^T Q_t z_t. \tag{3.22}$$

The derivation for R_t is similar.

Often *Differential Dynamic Programming* (DDP) is used to refer to this algorithm, while strictly speaking iterative LQR involves a minor simplification of the classical DDP algorithm [16].

3.4 Practicalities

In practice, the optimal policy for the approximation might end up not staying close to the sequence of points around which the approximation was computed

by the Taylor expansion. A solution for this is a cost function that forces the resulting policy to stay close[3]

$$(1-\alpha)C(s_t,u_t)+\alpha(||s_t-s_t^{(i)}||_2^2+||u_t-u_t^{(i)}||_2^2). \tag{3.23}$$

In each iteration the value for α (starting near to 1) is decreased. So the second term will ensure that only small steps toward the optimal policy are taken.

Note that the algorithm will only work is the cost function is convex. If the approximated matrices Q_t and R_t are not positive (semi)definite α can be increased until this is the case.

3.5 Receding horizon

At convergence of iterative LQR the linearizations are around the state-input trajectory the algorithm converged to. In practice the agent could not be on this trajectory due to various reasons. A solution to this problem is to resolve the control problem at each time t for the time steps t until H (as applied in [8]). Replanning the entire trajectory is often impractical. In practice we could replan only for a smaller number of timesteps until a horizon h. This requires a cost function $V_{t+h}^{\star}$ which accounts for all remaining timesteps from $t+h$ to H. This cost function could be taken from the offline run which planned the entire trajectory.

3.6 Preventing highly oscillating control inputs

On real systems often high frequency control inputs get generated [16]. This means control inputs over time are not smooth. This is typically undesirable

[3]With $||\cdot||_2$ we denote the 2-norm, also called the Euclidean norm. In general the p-norm is defined as $||x||_p=\left(\sum_{i=1}^{n}|x_i|^p\right)^{1/p}$.

and can result in poor performance. To solve this we can reformulate the LQR problem and penalize for change in control inputs:

$$\begin{bmatrix} z_{t+1} \\ v_t \end{bmatrix} = \begin{bmatrix} A_t & B_t \\ 0 & I \end{bmatrix} \begin{bmatrix} z_t \\ v_{t-1} \end{bmatrix} + \begin{bmatrix} B_t \\ I \end{bmatrix} \Delta v$$

with Δv the change in control inputs between time $t-1$ and time t. The new cost matrices are now

$$Q'_t = \begin{bmatrix} Q_t & 0 \\ 0 & R_t \end{bmatrix}$$

and R'_t which penalizes change in controls.

3.7 Summary

We described the iterative LQR algorithm, which iteratively approximates the dynamics and cost function along a state-action trajectory to get a standard LQR format. Then the standard solution for an LQR problem is used to find an optimal policy for an MDP. Iterative LQR is a local, trajectory-based method, which means that computational effort is focused along likely trajectories and generated controllers are only valid in a local region of the state space. In this way the algorithm is not directly subject to the curse of dimensionality and is suitable for learning controllers in high-dimensional, highly nonlinear systems with continuous state and action spaces.

Chapter 4

Simulated Car Racing Championship

In this work we build a controller for the Simulated Car Racing Championship (SCRC) [17]. This is a simulated car racing competition first organized in 2008. The championship consists of several races on different tracks. The competition is build around a complex car racing game, the open-source racing game *The Open Car Racing Simulator*.

4.1 The Open Racing Car Simulator

The Open Car Racing Simulator (TORCS) is an open-source car racing simulator [18]. It provides a physics engine, full 3D visualization and several tracks and models of cars. The car dynamics is very accurately simulated and the physics engine takes into account many aspects such as traction, aerodynamics and fuel consumption.

Each car is controlled by an automated driver. At each control step (approximately every 0.02 seconds), a driver can access the current game state through its sensors. Sensors include information about the car and the track, as well as information about the distance to other cars on the track

Name	Range (unit)	Description
`angle`	$[-\pi, +\pi]$ (rad)	Angle between the car direction and the direction of the track axis.
`curLapTime`	$[0, +\infty)$ (s)	Time elapsed during current lap.
`distFromStart`	$[0, +\infty)$ (m)	Distance of the car from the start line along the track line.
`distRaced`	$[0, +\infty)$ (m)	Distance covered by the car from the beginning of the race.
`gear`	$\{-1, 0, 1, \ldots, 7\}$	Current gear: -1 is reverse; 0 is neutral.
`lastLapTime`	$[0, +\infty)$ (s)	Time to complete the last lap.
`rpm`	[2000,7000] (rpm)	Number of rotations per minute of the car engine.
`speedX`	$(-\infty,+\infty)$ (km/h)	Speed of the car along the longitudinal axis of the car.
`speedY`	$(-\infty,+\infty)$ (km/h)	Speed of the car along the transverse axis of the car.
`speedZ`	$(-\infty,+\infty)$ (km/h)	Speed of the car along the z axis of the car.
`track`	[0,200] (m)	Vector of 19 range finder sensors: each sensor returns the distance between the track edge and the car within a range of 200 meters. The configuration of the range finder sensors (the angle with respect to the car axis) can be configured before the beginning of each race. When the car is outside of the track, the returned values are not reliable.
`trackPos`	$(-\infty, +\infty)$	Distance between the car and the track axis. The value is normalized with respect to the track axis, it is 0 when the car is on the axis, -1 when the car is on the right edge of the track and $+1$ when it is on the left edge of the car. Values greater than 1 or smaller than -1 mean the car is outside of the track.
`wheelSpinVel`	$[0, +\infty]$ (rad/s)	Vector of 4 sensors representing the rotation speed of wheels.
`z`	$[-\infty, +\infty]$ (m)	Distance of the car mass center from the surface of the track along the z axis.

Table 4.1: Description of the most important sensors. Ranges are reported with their unit of measurement (where defined). For a full overview of available sensors and effectors, see [17].

Name	Range	Description
accel	[0,1]	Virtual gas pedal (0 means no gas, 1 full gas).
brake	[0,1]	Virtual gas pedal (0 means no brake, 1 full brake).
clutch	[0,1]	Virtual clutch pedal (0 means no clutch, 1 full clutch).
gear	$\{-1,0,1,\ldots,7\}$	Set gear value.
steering	$[-1,1]$	Steering value: -1 and $+1$ means respectively full right and left, that corresponds to an angle of 0.785398 rad.

Table 4.2: Description of the available effectors.

(Figure 4.1). The car can be controlled by using the gas and brake pedal, the clutch, the gear stick and the steering wheel. For the most important sensors, see Table 4.1. For the available effectors, see Table 4.2.

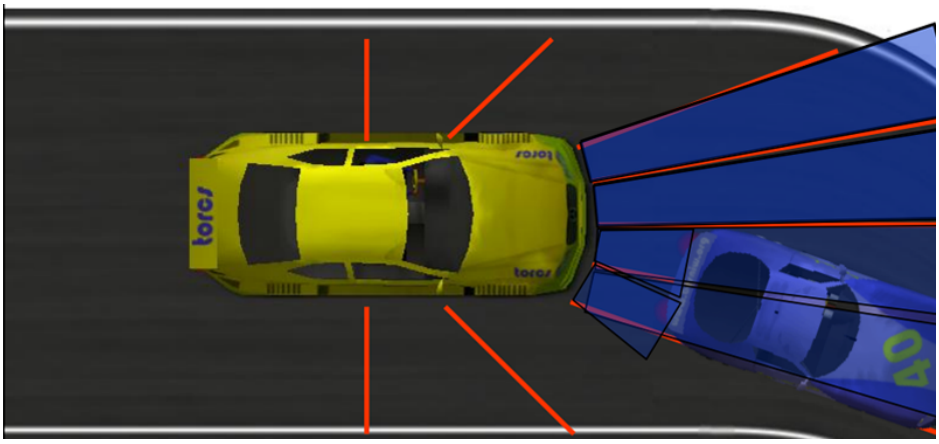


Figure 4.1: Range finder sensors return the distance to the track edge (red lines). Opponent sensors return the distance to the closest opponent in the covered area (blue). (Figure taken from [19].)

4.2 Competition organization

The competition is organized like a real Formula 1 championship. The competitions consist of different Grand Prix, each on a different racetrack. Competitors do not know the racetracks in advance. In general, the length of a racetrack is approximately between 2500 and 6000 meters. Each Grand Prix has three stages:

- During the warming-up (also called training) each driver races alone for 100,000 game ticks, which corresponds to approximately 33 minutes and 20 seconds of actual racing time. The warming-up can be used to form a model of the track.
- In the qualifying stage each driver races alone for 10,000 game ticks (approximately 3 minutes and 20 seconds). The eight drivers that cover the largest distance during the qualifying stage are qualified for the actual Grand Prix race.
- During the race the goal is to complete five laps and finish as first. At the end of the race the drivers are scored: 10 points to the first driver that completes the five laps, 8 points to the second one, 6 to the third one, 5 to the fourth one, 3 to the sixth one, 2 to the seventh one and 1 to the eighth one. In addition, the driver performing the fastest lap in the race and the driver completing the race with the smallest amount of damage receive 2 additional points each.

At the end, the team who scored most points summed over all Grand Prix wins the competition.

4.3 Competitors

In previous editions of the championship the best drivers had their parameters tuned using an evolutionary algorithm (to tune the parameters of a neural

network or a hand-coded controller). Fully hand-tuned controllers did not perform very well. Several competitors included some form of online (during the race) learning to remember previous crash situations so as to adapt their driving style in subsequent laps. Winner of the Simulated Car Racing Championship 2011 and 2012 was an advanced competitor called Mr. Racer [20, 21]. This competitor used the warming-up to generate a track model that estimates curvatures of the next part of the track. Then a controller can use this model together with expert knowledge and an evolutionary algorithm.

Imitation-based learning approaches have not been not very successful. Some competitors, including [22], tried to develop drivers by imitating human players using forms of supervised learning (a direct imitation approach). The performance of these competitors was limited and the approach has been abandoned by most of the participants. This is coherent with the published works which show that, in car racing, imitation-based learning generally fails to produce competitive controllers [23].

Chapter 5

States and Controls

To model the control problem of racing with a car on a racetrack for the Simulated Car Racing Championship we use a *Markov Decision Process* (MDP). We model the following elements to form this MDP. The MDP is a tuple $M = (S, U, T, D, H, C)$ where S and U are the state and control spaces. In this chapter we describe the elements we choose to form the state and control space.

5.1 State elements

We use the following elements to describe the state:

- $\dot{x}$ is the velocity of the car along the longitudinal axis of the car (in m/s). Thus, the velocity is expressed in the body (or robot centric) coordinate frame.
- $\dot{y}$ the velocity of the car along the transversal axis of the car (in m/s).
- $\dot{\omega}$ is the angular velocity of the car, also called yaw rate. The angular rate of the car is expressed in rad/s.

- p is the distance of the car from the start line along the track axis in meters (corresponding to `distFromStart` in Table 4.1).
- q is the normalized distance between the car and the track axis. The value is normalized with respect to the track width. It is 0 when the car is on the axis, -1 when the car is on the right edge of the track and $+1$ for the left edge (corresponding to `trackPos` in Table 4.1).
- α is the angle between the car direction and the direction of the track axis in radians (corresponding to `angle` in Table 4.1).

The body coordinate representation specifies the vehicle state using a coordinate frame in which the x and y axes are forward and sideways relative to the current orientation of the vehicle, instead of north and east. Intuitively, using this coordinate frame is the most natural way to describe velocities.

We augment the state s_t at time t with k previous states and control inputs $s_{t-1}, u_{t-1}, \ldots, s_{t-k}, u_{t-k}$ to be able to model state transitions more accurately. By including previous states and controls in the current state the Markov property (the future depends only upon the current state) is not violated. We rotate linear velocities ($\dot{x}$ and $\dot{y}$) at previous timesteps to the body frame at time t (see Chapter 7 for more details).

Note that our current state description does not include information about upcoming parts of the track and positions of opponents. Also we assume the car drives in a 2-dimensional plane and we do not include 3-dimensional elements, such as roll, pitch and velocity in the z direction. This is because these effects are negligible on most tracks and we do not have enough sensor information to measure them.

Most state elements are given directly by sensors. This is not the case for the yaw rate. The car has no gyroscope to measure the yaw rate directly and no compass from which we can derive the yaw rate. Fortunately, we can estimate the yaw rate with the use of velocity and rangefinder sensors.

5.2 Yaw rate estimation

The rangefinder sensors measure the distance to the track edge. To find the yaw rate we can estimate the rotation between two scans when we know the translation.[1] The rangefinders at time $t-1$ form the reference scan R and the rangefinders at time t form the current scan C. Now we want to find the rotation around the position of the car at time t such that the current scan matches the reference scan, see Figure 5.1. We pose this as the following optimization problem:

$$\dot{\omega} = \underset{\beta}{\operatorname{argmin}}(\operatorname{dist}(\operatorname{rot}(C, \beta), R))/\Delta t \tag{5.1}$$

Here Δt is the time elapsed between time $t-1$ and t. By dividing the angle β by Δt we can get an estimate for the yaw rate $\dot{\omega}_t$ at time t; dist is a distance function which measures the distance between the current scan C rotated by an angle β and the reference scan R.

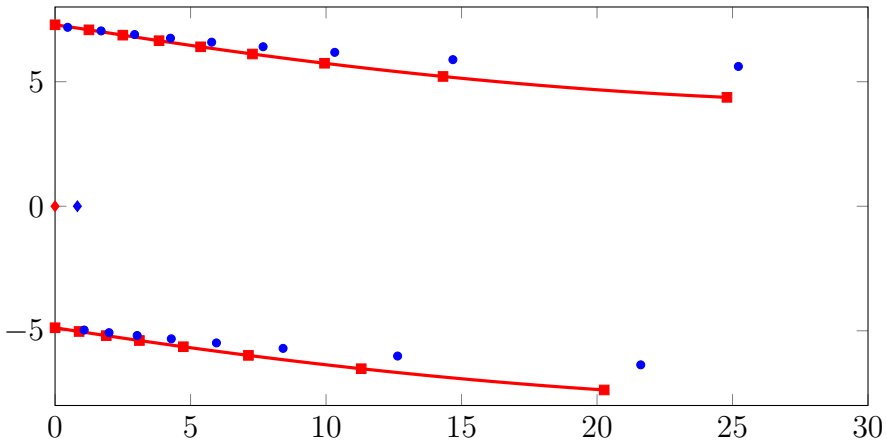
5.3 Control elements

The state is affected by the controls. We define the following controls which form the control space U:

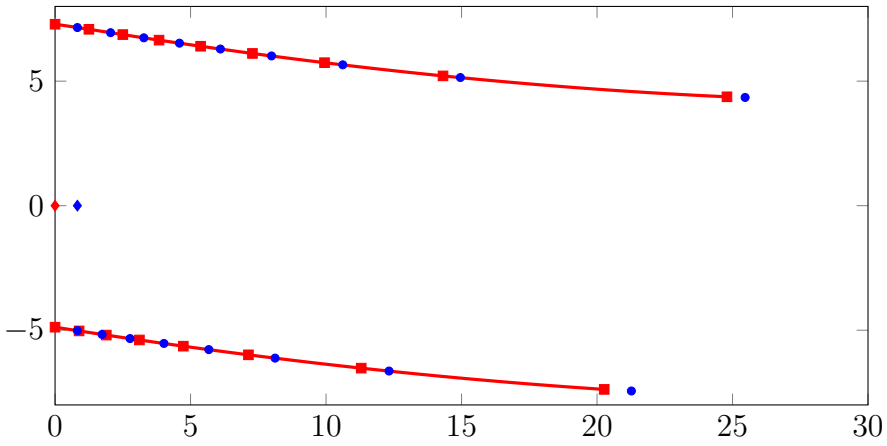
- u_1 controls the speed of the car and is a combination of the gas and the brake pedal. The range is $[-1, +1]$ with -1 meaning full brake and $+1$ full gas. An input of 0 means no braking and no gas.
- u_2 controls the steering wheel ($[-1, +1]$) and affects the heading of the vehicle in this way, with -1 meaning full right steering and $+1$ full left.

Gear and clutch are controlled by some fixed rules and are thus not controlled directly by the controller.

[1] We also tried a simplification of the Polar Scan Matching algorithm [24], which works directly in the polar coordinate system. However, this gave less accurate results.



(a)



(b)

Figure 5.1: The rotation of the blue circles (current scan C at time t) around the blue diamond (new position at time $t+1$) such that the blue points lie on the curve formed by the red squares (reference scan R at time $t-1$) estimates the rotation the car has made. The red diamond at $(0,0)$ represents the initial position of the car at time $t-1$. The new position is estimated with the linear velocities $\dot{x}$ and $\dot{y}$ at time $t-1$. So we plotted the scan C in the coordinate frame of scan R at time $t-1$ and than search for the rotation this scan C has to made to fit to the track edges already present. In (b) the result is shown after a search for the rotation angle that results in a minimal distance between the scans. Hence we have a estimate for the rotation between the two consecutive times $t-1$ and t and so we obtain an estimate for the angular velocity $\dot{\omega}$ at time t.

Chapter 6

Learning Features

In our state representation described in Chapter 5 we have no information about the upcoming part of the track. If the car is approaching a sharp left or right turn, then it has to anticipate and lower its speed. In the standard formulation for iterative LQR (Chapter 3) we assume a fixed horizon H. Then we compute a time-varying policy, meaning that we have a separate policy for each timestep t. If we have to deal with an infinite horizon or we want a more robust policy, then we want a non time-varying policy. However, to have such a policy we need a more expressive state representation. Our state also has to represent the track curvature for the upcoming part of the track. In this chapter we propose ideas on learning features corresponding to curvatures of track parts with the use of unsupervised feature learning. The goal is to have k features that are each sensitive for a specific track segment. For example, if our car approaches a sharp right turn we want one of the feature detectors to become active and another one if the car approaches a sharp left curve.

6.1 Unsupervised feature learning

The field of unsupervised feature learning [25] has become very active in recent years. The goal is to learn feature representations from unlabeled data.

When the feature representations are used to learn a new set of features one can build a deep hierarchical representation of features. When using unlabeled image data features in the first layer can correspond to edges and features in higher layers can become sensitive to objects parts. When building very deep models from very large amounts of data features in the higher layer can correspond to faces, pedestrians or cats [26].

6.2 Learning features

We use unlabeled data to learn a dictionary of features. If we have a new example of a track segment we want to encode this to a feature vector.

The first component needed is an unsupervised learning algorithm to learn a number of features. Clustering algorithms, like Mixture of Gaussians, K-means, auto-encoders and Restricted Boltzmann Machines are possible.

The second component is a feature encoding that maps each new example into a feature vector.

The unsupervised learning algorithm we choose to use is K-means because of its simplicity. It is a fast and relatively simple algorithm. Although it is a simple algorithm [27] showed that k-means methods match or outperform more sophisticated methods on various classification tasks in computer vision. The feature encoding we use is the "triangle" method from [27]. Given the centroids D obtained by K-means the feature encoding ϕ is defined by:

$$\phi_k = \max(0, mean(v) - v_k) \tag{6.1}$$

where $v_k = \left|\left|x - D^{(k)}\right|\right|_2$ is the Euclidean distance between the new example and the k-th centroid. The mean of the elements in v is computed by $mean(v)$. If the distance to a feature is "above average" then ϕ_k will become 0. This will set roughly half of the feature vector to zero and encourages some sort of competition between features.

Given features and a feature encoding we are able to map any track patch

of length r to a K-dimensional feature vector ϕ. This vector can be interpreted as a track segment representation of reduced dimension.

In Figure 6.1 the learned bases by the K-means algorithm are visualized. We see some curves to the left and some curves to the right. However, most patches go straight ahead. More research is needed to learn really useful features from track patches. Here we presented some initial directions to start with. Because there are some curves visible in the learned bases, it is a promising direction for more research.

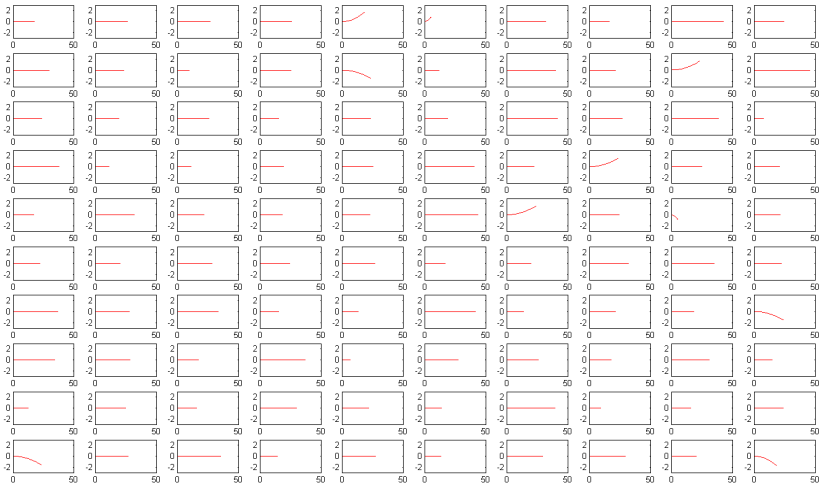


Figure 6.1: Learned bases after 50 iterations of K-means.

Chapter 7

Dynamics Model

The dynamics describe how a system evolves over time. Modeling these dynamics is often the most difficult element to specify in an MDP. We have to build a function $s_{t+1} = f(s_t, u_t)$ that can predict the next state s_{t+1} given the current state s_t and control input u_t at time t. This function can be specified using physical principles, learned from data generated on the real system or a combination of both. In this chapter we describe how to build a dynamics model from real data and some physical principles for the Simulated Car Racing Championship. First we focus on how to predict linear and angular velocities for the next state. Predicting the new position and orientation with respect to the track axis is then fairly simple, supposing we have built a map of the track.

7.1 Non-Markovian dynamics

Inspecting a state-input trajectory shows non-Markovian effects. An example is shown in Figure 7.1. We see that full left or right steering at time t does affect the yaw rate at time $t + 1$ *and* future times. Therefore, to be able to predict the next state, we need information from previous states and controls. As described in Chapter 5, state and control elements from the last k timesteps

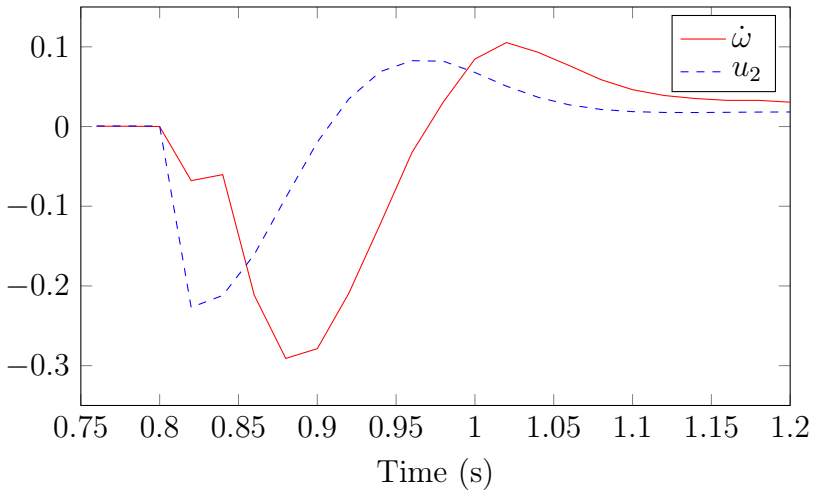


Figure 7.1: Effect of steering input u_2 (blue dashed) on yaw rate $\dot{\omega}$ (red solid).

are included into the current state.

7.2 Acceleration prediction model

By using the body coordinate frame in the state representation for the velocities certain symmetry properties are already encoded into the model, such as that the dynamics for the velocities are the same regardless of position, orientation and heading. We build a model that can predict accelerations, because forces and torques (and thus accelerations) are often a fairly simple function of state and controls [28].[1] By using the following update equation from physics we can update the velocities $(\dot{x}, \dot{y})$ in the body coordinate frame:

$$(\dot{x}, \dot{y})_{t+1} = R(\dot{\omega}_{t+1}) \cdot ((\dot{x}, \dot{y})_t + (\ddot{x}, \ddot{y})_t \, \Delta t)$$

Here, $R(\dot{\omega}_{t+1})$, is the rotation matrix that rotates the body-coordinate frame from time t to the body frame at time $t + 1$ (and is determined by the yaw

[1] Remember Newton's second law of motion: $F = m \cdot a$.

rate)[2]. This equation makes clear that the change in velocity expressed in the body coordinate frame does not correspond directly to physical accelerations, because the body coordinate velocities at times t and $t+1$ are expressed in different coordinate frames. Thus, $\dot{x}_{t+1} - \dot{x}_t$ is not the forward acceleration. To obtain the acceleration the velocity at time $t+1$ must be rotated into the body frame at time t before taking the difference. Doing this makes it easier to capture *sideslip* into the model, the effect that the heading of the car is not aligned with its direction of motion when turning.

We use the following parameterization, partially based on the helicopter model in [7], to predict the accelerations $(\ddot{x}, \ddot{y}, \ddot{\omega})$ as a function of state and control inputs:

$$\ddot{x}_t = B_1(\dot{y}_t\dot{\omega}_t) + A_x\dot{x}_t + \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}^T \begin{bmatrix} (u_a)_{t-1} \\ (u_a)_{t-2} \\ (u_a)_{t-3} \end{bmatrix} + \begin{bmatrix} C_4 \\ C_5 \\ C_6 \end{bmatrix}^T \begin{bmatrix} (u_b)_{t-1} \\ (u_b)_{t-2} \\ (u_b)_{t-3} \end{bmatrix},$$

$$\ddot{y}_t = B_2(\dot{x}_t\dot{\omega}_t) + A_y\dot{y}_t,$$

$$\ddot{\omega}_t = A_\omega\dot{\omega}_t + C_7(u_2)_t + \begin{bmatrix} B_3 \\ B_4 \\ B_5 \end{bmatrix}^T \begin{bmatrix} \dot{x}_t\dot{\omega}_t \\ \dot{x}_{t-1}\dot{\omega}_{t-1} \\ \dot{x}_{t-2}\dot{\omega}_{t-2} \end{bmatrix} + \begin{bmatrix} C_8 \\ C_9 \\ C_{10} \\ C_{11} \\ C_{12} \end{bmatrix}^T \begin{bmatrix} \Delta(u_2)_t \\ \Delta(u_2)_{t-1} \\ \Delta(u_2)_{t-2} \\ \Delta(u_2)_{t-3} \\ \Delta(u_2)_{t-4} \end{bmatrix}.$$

Here the acceleration control u_1 is separated again in a gas pedal value u_a and a brake pedal value u_b, because negative accelerations due to braking are in general much greater than positive accelerations due to pushing the gas pedal.[3] With $\Delta(u_2)_t$ we describe the difference in steering input between time t and $t-1$. The number of parameters from previous timesteps is selected

[2]We use ω_{t+1} instead of ω_t, because ω_{t+1} is the best estimate for the rotation made.

[3]The values u_a (gas) and u_b (brake) are in the range $[0, 1]$. Values for u_1 smaller than 0 result in a positive value for u_b. Values for u_1 greater than 0 result in a positive value for u_a.

using cross-validation. The free parameters A, B and C are to be learned from racing data. The parameters A model damping, B the influence of control inputs and C the influence of state features.

The model is relatively simple and has a sparse dependence on the current velocities, angular rates and inputs. In the model some simplifying assumptions are made. We do not model gear changes and clutch effects. Furthermore it assumes that the control inputs are independent from each other, u_1 does only control $\dot{x}$ and u_2 does only control $\dot{\omega}$ directly. With the complex dynamics of a car this is probably not true, but a model of this form should be able to capture the most dominant aspects.

7.3 Parameter learning

To learn the free parameters we collect a state-input trajectory from an expert driver. For each time t we compute the corresponding accelerations $(\ddot{x}, \ddot{y}, \ddot{\omega})_t$. We use linear regression to find the parameters which minimize the mean squared prediction error:

$$\frac{1}{H}\sum_{t=1}^{H}(\ddot{x}_t - \hat{x}_t)^2 \tag{7.1}$$

for the true accelerations $\ddot{x}$ and predicted accelerations $\hat{x}$, with H the number of consecutive timesteps. The same holds for the other accelerations $\ddot{y}$ and $\ddot{\omega}$.

7.4 Building a track map

To predict the new position and orientation $(p, q, \alpha)_{t+1}$ with respect to the track axis we need a map of the track. Most other controllers use a simple heuristic to estimate the curvature of the upcoming track segment. They rely on the longest of the 19 rangefinders. This leads to problems for tight curves as indicated by [20], see Figure 7.2. With a map of the track this type of curve is recognized early.

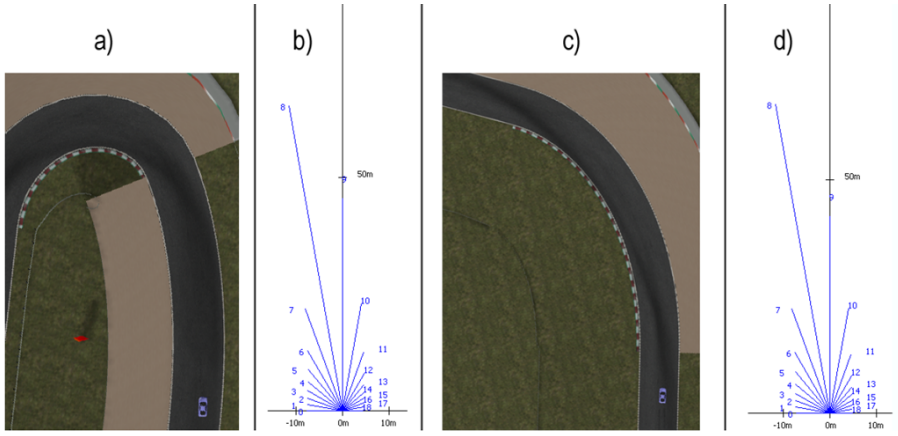


Figure 7.2: Two identical rangefinder scans for approaching two different curves in (b) and (d) for the two totally different curves (a) and (c). The sensor information is misleading in this case. (Figure taken from [20].)

We build this map from a state trajectory of one lap. With a state trajectory containing $(p, q, \alpha)_{t=1}^{H}$ with H timesteps (see Chapter 5) we can compute $H - 1$ points in a fixed coordinate frame that lie on the track axis for each time t. Then we compute the distance and rotation between these consecutive points. The resulting map is a sequence of tuples $M = (p_i, \alpha_i, d_i)_{i=1}^{H}$. Here p_i is the position of a point along the track axis (in meters), α_i is the rotation the axis makes when going to the next point on the axis and d_i is the distance to the next point. We do not define the map as a sequence of points in a fixed coordinate frame, because the points are estimates and we do not want to introduce accumulated errors into the map. In Figure 7.3 we have a reconstruction of the track from the map we build from one lap of racing data.

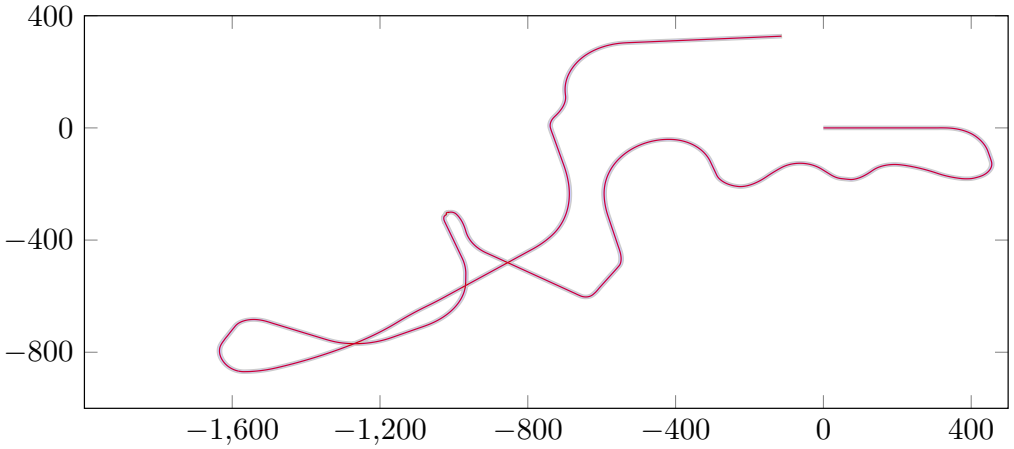


Figure 7.3: The map of the track that is build from data from one lap. The start is located at $(0, 0)$. As we can see, the map is not perfect. The end of the track is not directly connected to the start at the $(0, 0)$ point. This is probably due to inaccurate yaw rate estimations. Hence we cannot make of perfect map of the curves in the track. However, because the end of the track is nearby the start we assume the map is reasonable at distances of a few hundred meters. Further looking ahead while racing is probably not necessary. The original track in the simulator is a reconstruction of the Suzuka Circuit in Japan [29].

	Training		Test	
Acceleration	**MSE**	**Signs**	**MSE**	**Signs**
$\ddot{x}$	3.52	0.93	3.22	0.93
$\ddot{y}$	97.74	0.88	111.50	0.88
$\ddot{\omega}$	7.67	0.80	8.83	0.79

Table 7.1: Results on both the training and test set. Mean-squared error and relative number of signs predicted correctly are reported.

7.5 Predicting position and orientation using a track map

With the predicted velocities and yaw rate we have an estimate of the new position and orientation relative to the current position and orientation. With the map of the corresponding track of the map we can now compute $(p, q, \alpha)_{t+1}$, the new position and orientation *with respect to* the track axis.

7.6 Results

First we describe results of our acceleration model. We collect a demonstration from an expert driver [21] of approximately 10 minutes at a race track with a lot of variation in curves. We use one part of this demonstration (70%) to train our acceleration model and the other part (30%) as a test set to report results. We report the mean-squared error and the relative number of signs (positive or negative acceleration) predicted correctly over both the training and test set in Table 7.1.

To test also the performance up to 10 simulation steps (0.2 seconds) we report the mean-squared error for the 6 relevant state elements $\dot{x}$, $\dot{y}$, $\dot{\omega}$, p, q, α. The results are in Figure 7.4. Here we use the acceleration model and the map to predict the linear velocity, angular velocity and the position and orientation relative to the track axis. We do not only predict this for the next timestep, but over a longer period of 10 timesteps.

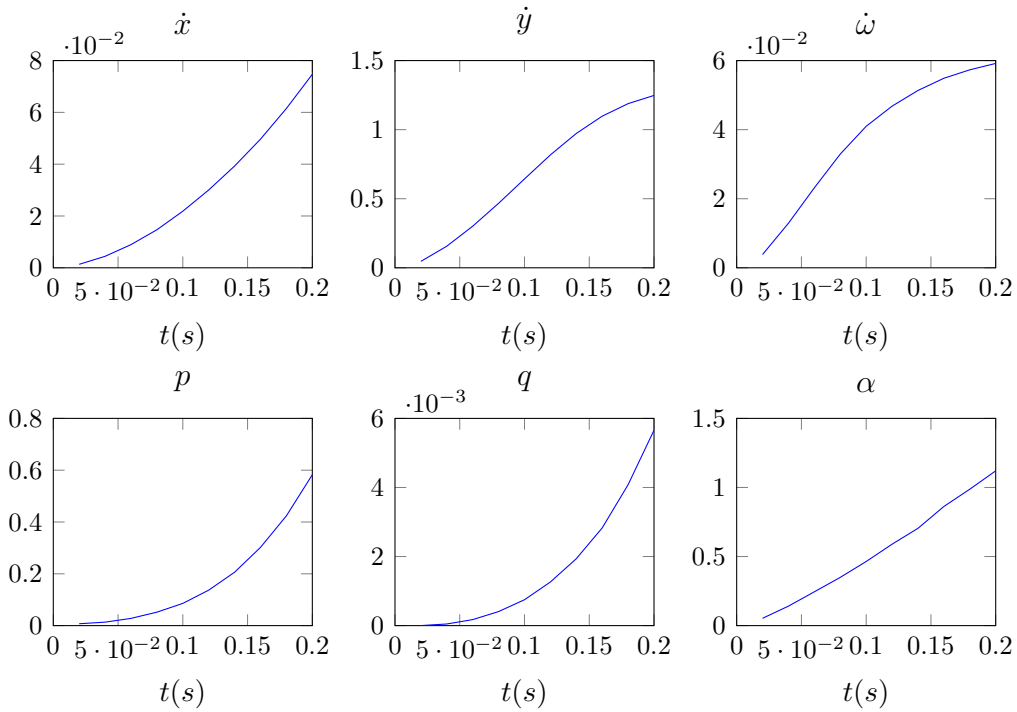


Figure 7.4: Mean-squared prediction error throughout 0.20 second simulations (10 simulation steps) for the linear velocities $\dot{x}$ and $\dot{y}$, the angular velocity $\dot{\omega}$ and the position and orientation state-elements p, r and α. The time is plotted on the horizontal axis in seconds ($t(s)$) versus the mean-squared error on the vertical axis.

Because we have no other model to compare our results with, we zoom in on a particular part of the real raced trajectory. We compare this part with the prediction of our model for this part for a period of 4 seconds. The part of the trajectory is in a curve, so also steering and yaw rate are involved. We show the prediction for this particular part in Figure 7.5.

We show the real trajectory for this particular part together with the predicted trajectory on the map of the track we build in Figure 7.6. The predicted trajectory is shown on the map with use of the position and orientation elements of the predicted state. The inputs u_1 and u_2, for accelerating/braking and steering during the predicted period, are in Figure 7.7.

7.7 Discussion

In this chapter we built a dynamics model for the Simulated Car Racing Championship. In the results for the 4-second simulation we see that in the case for the choosen part of the trajectory the angular velocity $\dot{\omega}$ is underestimated during the first 2 seconds. Hence the predicted trajectory goes off the track and the error for lateral track position q increases. During the last 2 seconds of the simulation the angular velocity $\dot{\omega}$ is overestimated. Hence we see a sharp curve in the predicted trajectory that goes back to the track axis.

Based on this simulation we can say that the trends and directions in the linear and angular velocities are predicted reasonably, although the exact magnitude is not very accurate. However, we think this is comparable with an average human driver. When you push the gas or brake pedal you expect an increase or decrease in speed respectively, while you do not know the exact magnitude of the acceleration. When you steer to the left you expect the vehicle is turning to the left and vice versa for steering to the right, but how much the vehicle will turn exactly is not known. During making the turn you see if the turn will be too short or too wide and then you correct your

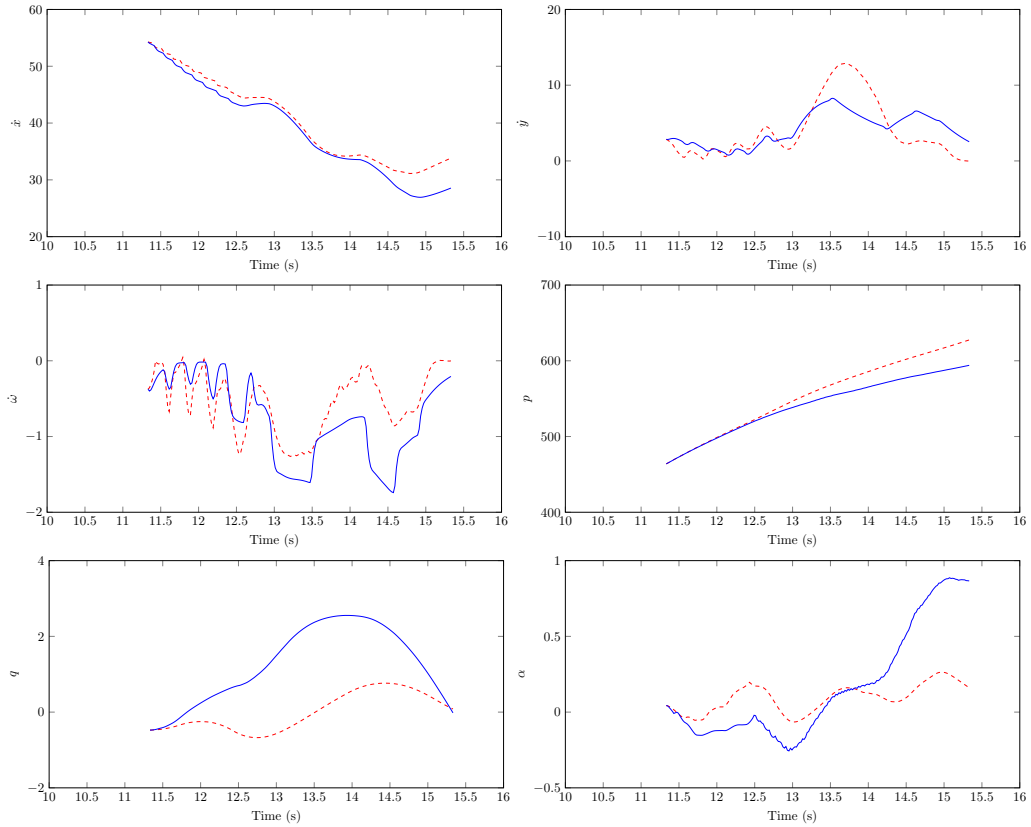


Figure 7.5: The state elements and predictions for a particular part of the trajectory (4 seconds). The predicted state trajectory started in exactly the same state as the real trajectory and has the same control inputs over a period of 4 seconds (200 simulation steps, the real time $t(s)$ is shown on the horizontal axis). The blue dotted line shows the predicted state elements over the predicted period and the dashed red line shows the real values.

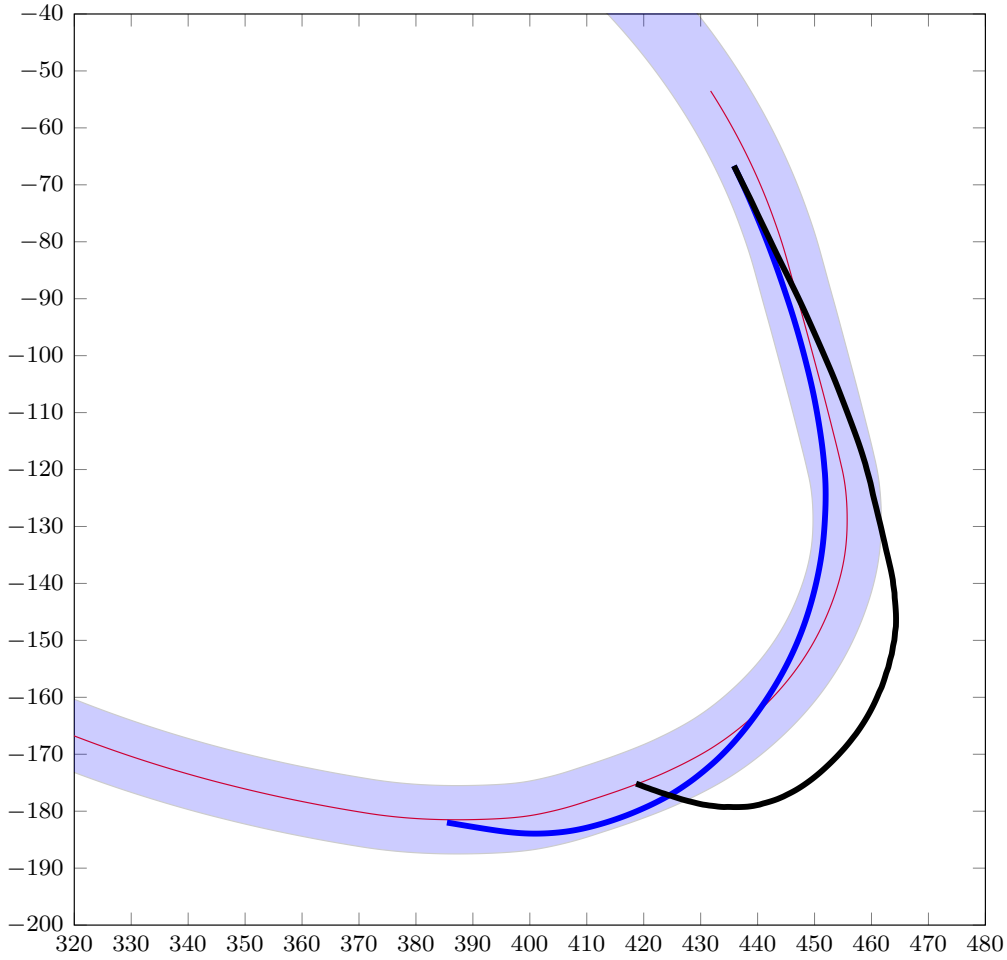


Figure 7.6: The real raced trajectory and the predicted trajectory shown on the map of the track for a period of 4 seconds. Both trajectories start from exactly the same state. The blue one is the real raced trajectory and the black one is the predicted trajectory, drawn on the map using predicted states. The red line is the track axis. The track itself is shown in purple. The predicted trajectory can go off the track, because the dynamics model does not take the track edge into account.

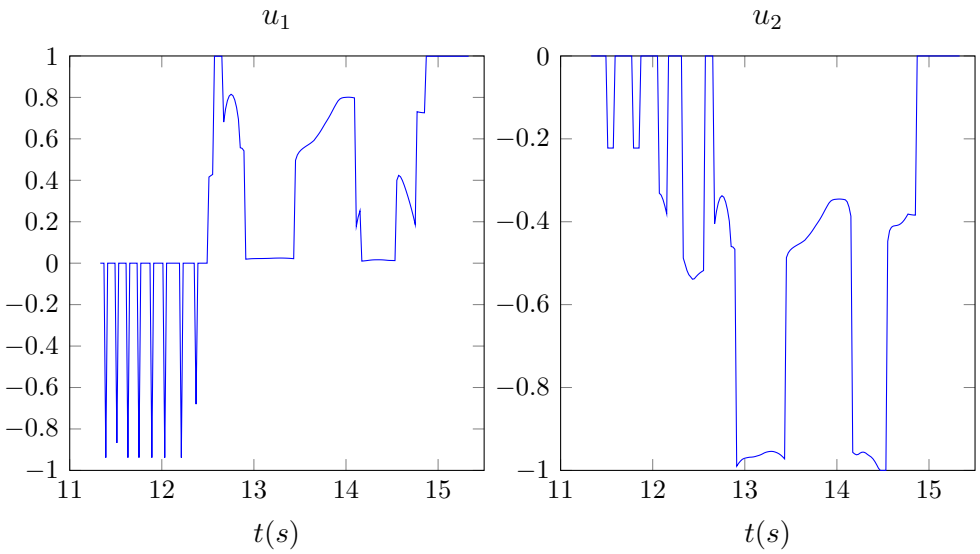


Figure 7.7: Control inputs u_1 for accelerating/braking and u_2 for steering versus time $t(s)$. During approach of the curve the brake is used. After that the gas pedal is used to accelerate again. During this period the driver only steered to the right (negative values for u_2).

steering. The ideas of using reinforcement learing with inaccurate models are exploited in [2, 15]. With the use of these algorithms for inaccurate models in reinforcement learning and practising the task during trials to refine performance it might be possible to use the dynamics model described in this chapter for a racing task in the Simulated Car Racing Championship.

Chapter 8

Reward Function

In the previous chapter we focussed on building a dynamics model. In a Markov Decision Process another important element is the reward function. The reward function is a description of the task that has to be performed. The agent's goal is to collect as much reward as possible. Specifying a reward function for complex tasks is not simple, because how to formulate a reward function for "driving well"? Intuitively, you might know what you want, but can not say it. *Inverse Reinforcement Learning* (IRL), also known as inverse optimal control (IOC), tries to recover an unknown reward function from expert demonstrations in a Markov Decision Process (MDP). This learned reward function can be used to infer the expert's goal or generalize the expert's behaviour to new situations. Work on IRL for driving tasks in a continuous simulator is reported in [30]. Actions in their simulator, similar to our work, correspond directly to gas, brake and steering control. In [30] a controller learns an aggressive driving style (cutting of other cars), an evasive one (driving fast, but keeping plenty of clearance) and a tailgater style (following other cars very closely) from expert demonstrations. We try to implement the linear variant of the algorithm from [30] to learn a reward function from demonstrations.

8.1 Inverse Reinforcement Learning

Inverse reinforcement learning (IRL) tries to recover the intention from expert demonstrations in a form understandable for computer algorithms. In reinforcement learning the reward or cost function is the most generalizable representation of the behaviour of an agent (see Figure 8.1). Therefore IRL tries to recover this reward function from the demonstrations, to generalize the expert's behaviour to new situations. An example where this is applied succesfully is learning humanoid locomotion from demonstration of human running [31]. Here naturalistic running behaviours could be used in new environments, including rough terrain.

In IRL we assume the expert is trying to collect as much reward as possible in an MDP with an unknown reward function. IRL tries to find a reward function that makes the demonstrated behaviour of the expert (near-)optimal. Once we have found such a reward function, any reinforcement learning technique can be used to find an optimal policy for the MDP which does now include a reward function.

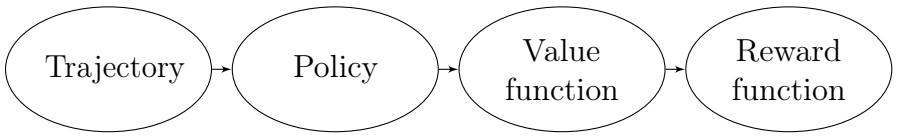


Figure 8.1: Ordering in terms of generalizability from least generalizable (trajectory) to most generalizable (reward function) [32].

In IRL two variants for the form of a reward function are used. It can be either a linear combination of provided features or a nonlinear combination of features. In the simpler but less expressive case of a linear combination, IRL algorithms try to recover the feature weights. Still there is a need for smart engineering of the reward features. In the case of a nonlinear function the reward function is represented as a Gaussian process [33]. In this case more general reward features are possible, because the Gaussian process can also find nonlinear relationships.

8.2 Algorithm

The first IRL algorithm that can deal with high-dimensional and continuous state and action spaces is presented in [30], we use the linear variant of this algorithm. We have a dynamics function $s_{t+1} = f(s_t, u_t)$ and assume an unknown reward function $r(s_t, u_t)$, where s_t is the state and u_t the action at time t. The optimal actions are given by

$$u = \max_u \sum_t r(s_t, u_t) \tag{8.1}$$

An IRL algorithm tries to find a reward function r under which the optimal actions match the expert's demonstrations. Reward features are given to the algorithm, $\mathbf{f} : (s_t, u_t) \to \mathbb{R}$. These features can be used to represent the unknown reward as

$$r(s_t, u_t) = \sum_j \theta_j \mathbf{f}_j(s_t, u_t) \tag{8.2}$$

Real demonstrations are rarely perfectly optimal. Therefore we require a model that can explain this suboptimality or "noise". In [34] such a model is presented, called the maximum entropy IRL (MaxEnt) model. With this model, the probability of a sequence of actions $u = u_1, \ldots, u_H$, for an action sequence of H steps given an initial state, is proportional to the exponential of the rewards collected along that trajectory:

$$P(u|s_0) = \frac{1}{Z} \exp\left(\sum_t r(s_t, u_t)\right) \tag{8.3}$$

where Z is the partition function that describes the total exponential of the reward for all possibilities of the action sequences u. In discrete environments Z can be written as

$$Z = \sum_{i=1}^{n} \sum_t r(x_t, u_t) \tag{8.4}$$

with n the number of possibilities for the action sequence u. However, for continuous environments we need to write Equation 8.4 with an integral over all possibilities

$$P(u|x_0) = e^{r(u)} \left[\int e^{r(\tilde{u})} d\tilde{u} \right]^{-1} \tag{8.5}$$

The reward along paths for all $\tilde{u}$ can be approximated with a second order Taylor expansion of r around u:

$$r(\tilde{u}) = r(u) + (\tilde{u} - u)^T \frac{\partial r}{\partial u} + \frac{1}{2}(\tilde{u} - u)^T \frac{\partial^2 r}{\partial u^2}(\tilde{u} - u) \tag{8.6}$$

With some derivations shown in [30] we can obtain the approximate log likelihood of an action sequence u given an initial state as

$$\ell = \frac{1}{2}\mathbf{g}^T \mathbf{H}^{-1} \mathbf{g} + \frac{1}{2} \log |-\mathbf{H}| - \frac{d_u}{2} \log 2\pi \tag{8.7}$$

Here g is the gradient of the total reward with respect to the action sequence, $\mathbf{H}$ is the Hessian and d_u is the dimensionality of the action.

Determining the reward weights θ that maximize the approximate log likelihood is possible with gradient-based optimization (for example L-BFGS[1]). This requires the gradient $\frac{\partial \ell}{\partial \theta}$. With the algorithm in [30] this gradient can be computed and the weights θ can be learned.

8.3 Features

The algorithm has to be presented with demonstrations and reward features. We use demonstrations with a length of 10 seconds (500 samples). The demonstrations are from parts everywhere on the track.

We propose a number of reward features with which we can build the reward function. The features are:

[1]Limited-memory BFGS (L-BFGS) is an advanced method for solving nonlinear optimization problems.

- Distance from track edge. Distance from the track edge is rewarded with

$$\mathbf{f_1} = 1 - q_t^2 \tag{8.8}$$

with q_t the lateral position with respect to the track axis element at time t. Most important for the car is of course to stay on the track. This feature gives most reward for driving in the middle of the track. However, to take a curve optimally an expert driver will approach the curve from the outside, then go to the inside to end the curve at the outside again. This behaviour cannot be captured with this feature.

- High-speed. Speed in longitudinal direction $\dot{x}$ is another reward feature

$$\mathbf{f_2} = \dot{x}_t \tag{8.9}$$

A high longitudinal speed is good for most parts of the track. However, when going into a curve the speed cannot be too high, because of the danger of loosing control and going off the track.

- Distance covered. This seems a very important feature. This feature is computed with

$$\mathbf{f_3} = p_t - p_{t-1} \tag{8.10}$$

with p_t the distance of the car from the start line along the track axis at time t. The goal is to race as fast as possible, thus to cover as many meters as possible along the track axis each timestep.

- Lateral speed. The lateral speed feature is

$$\mathbf{f_4} = \dot{y}_t^2 \tag{8.11}$$

where $\dot{y}$ is lateral speed. Lateral speed is something we want to avoid in most cases. A car with high lateral speed is difficult to control.

- Angle. The angle with respect to the track axis α_t is another feature

$$\mathbf{f_5} = \alpha_t^2 \tag{8.12}$$

  A too big angle means the car is facing a wrong direction, so this is something that also should be avoided.

- Steering effort. The steering feature looks like

$$\mathbf{f_6} = u_2^2 \tag{8.13}$$

  where u_2 is the steering input. Too abrupt steering can lead to loss of control of the car.

The features chosen contain no discontinuities. For each reward feature the gradient and the Hessian with respect to the state and the action are computed. These gradients and Hessians will be denoted by $\tilde{g}^{(k)}$, $\hat{g}^{(k)}$, $\tilde{H}^{(k)}$ and $\hat{H}^{(k)}$ with k the feature index.

8.4 Results

We run the linear variant of the algorithm described in [30]. An extra regularizer feature is added as also described in [30]. This is done in order to make the determinant of the negative Hessian in Equation 8.7 positive. Otherwise the log determinant is undefined. This regularizer feature is a constant 1 for all states.

However, when running the algorithm the initial weight of the regularizer feature is doubled until the negative Hessians of all example paths are positive definite. In practive, with our example demonstrations, this weight keeps being doubled until the maximum number of iterations is reached. So no feature weights are found that say something about the objective that the expert is trying to optimize.

8.5 Discussion

It turned out the IRL algorithm from [30] works well on the more simple and artificial examples in [30]. However, on more realistic data from a detailed racing simulator the algorithm seems to be not robust enough. The algorithm does not succeed in making the matrices positive semi-definite. This could have to do with the shape of the features. However, very simple features without discontinuities are tried in our experiments.

An alternative IRL algorithm that does probably have more chance to be successful is described in [31]. This is an LQR-based dynamic programming algorithm. This method tends to be more stable and robust. It is mathematically equivalent and just a more stable way of computing the same thing.

By learning a linear combination of features the reward function is still limited in its expressiveness. The linear features have to form a good basis of the reward function. Instead of learning weight for a linear combination of reward features we can use a Gaussian process as a reward function [33]. In comparison to a linear reward function this can yield a reward that is nonlinear in its features. This method can work with less expressive features. However, it comes with the bias and variance tradeoff that arrive with the increased complexity of the model. The possibilities of nonlinear reward functions are not further explored in this work.

So IRL algorithms try to recover a reward function from expert demonstrations, but still the reward features need to be engineered by hand. Future research has to find out whether it is possible to automatically extract reward features from state-action trajectories of an expert. In many applications deep learning has already replaced hand-engineerded features by features learned from data successfully [35].

Chapter 9

Experiments

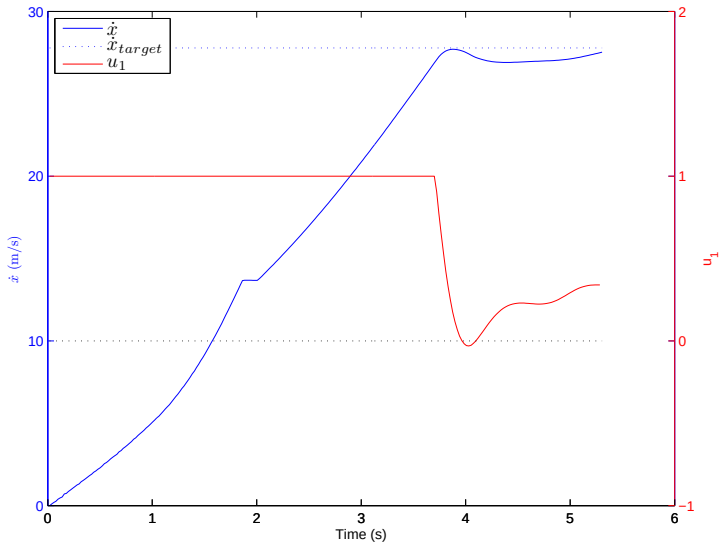
In this chapter we describe an experiment with a reinforcement learning algorithm. We apply the iterative LQR algorithm as described in Chapter 3 to find an optimal controller for the Markov Decision Process (MDP) described in the previous chapters. We use the dynamics model learned in Chapter 7. We define an experiment to test the controller. A trajectory is provided to initialize the iterative LQR algorithm.[1]

9.1 Constant speed

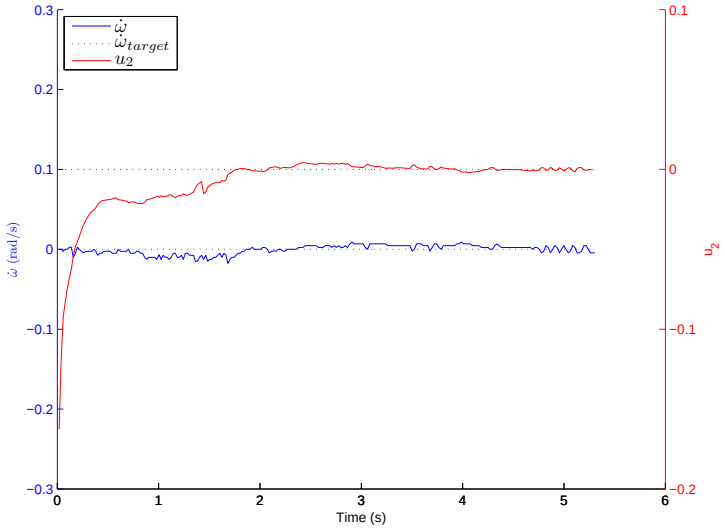
In this experiment the car has to reach a constant longitudinal speed ($\dot{x}$) of 50 km/h (13.89 m/s), while not turning. The car starts from zero speed. The duration of the experiment is 250 timesteps (approximately 5 seconds). We define a cost function that quadratically penalizes deviations from the desired longitudinal speed and yaw rate (yaw rate penalties are also multiplied by 10 because of the difference in order of magnitude). Also change in control inputs is penalized. This experiment is relative simple, but the car has to control

[1]The reference trajectory used for the first iteration of the algorithm is a trajectory raced by a driver, called *SimpleDriver*, on the same race track. This driver follows simple heuristics to stay on the track and accelerate where possible.

the gas, brake and the steering wheel. Therefore this experiment requires a dynamics model with knowledge of how to control longitudinal speed and yaw rate with the use of these controls. Results are reported in Figure 9.1. We can see that the controller is able to go to the desired speed, while not turning. Three iterations of the iterative LQR algorithm are used. The α-values are decreasing from 0.5 to 0.25 to 0.



(a)



(b)

Figure 9.1: (Best viewed in color.) Generated control inputs by the controller. In (a) we see the time on the horizontal axis and the resulting speed $\dot{x}$ on the left vertical axis and the generated control u_1 at the right vertical axis. The gas pedal is pushed to the limit until the desired speed is reached. In (b) the yaw rate $\dot{\omega}$ is on the left vertical axis and the steering control u_2 on the right vertical axis. Steering is around 0 (no steering) to keep the yaw rate at zero. The right steering in the beginning does not have much effect because of the low speed.

Chapter 10

Conclusions

In this work we focused on controlling a car in the Simulated Car Racing Championship. We used apprenticeship learning, an indirect form of imitation learning. This means that demonstrations of an expert for the task to be performed are available. These demonstrations are useful for learning the two key elements that are needed before a reinforcement learning/optimal control algorithm could be used: i) a dynamics model and ii) a reward/cost function.

The formalism of a Markov Decision Process is used to describe the task of racing in the Simulated Car Racing Championship. A state and action space is proposed and using these elements of the MDP a dynamics model is build. The model is built using data from the demonstrations and some physical principles. An attempt is made to learn a reward function from the demonstrations available and the iterative LQR algorithm is used to come up with a policy for a relatively simple task in the real racing simulator.

When looking to the results of the dynamics model we can conclude that the model is not perfect, but does a reasonable job in predicting the accelerations. The model can be inaccurate sometimes, but most important is that the signs of the accelerations are predicted correctly in most cases.

Learning a reward function from demonstrations turned out to be difficult for a task in a very realistic and complex environment. The inverse optimal

control algorithm that was succesful for other simpler environments did not succeed in our complex environment.

Also applying the iterative LQR algorithm for this environment turned out to be far from simple. To keep the matrices required for the cost function positive (semi-)definite, large values for deviation from the trajectory linearized about have to be choosen. Also bounds for the control inputs are not included in this algorithm, so generated inputs can be outside the possible bounds. However, for the relatively simple task of accelerating to a constant speed, while not turning, this was not a problem. A controller for this task was created successfully with the dynamics model learned and a manually defined cost function.

10.1 Future work

It can be too difficult to capture all the dynamics in one "global" model. However, we can use multiple "local" models. When making a new prediction for a certain state-action pair we consider only data points in the neighbourhood of this query point to build the model. One drawback of this method is that it is very expensive, because we have to build a new model for every prediction. This method is called Locally weighted linear regression [36]. Another possibility is to deal with the inaccuracies in the dynamics model and use algorithms that can handle this [2, 15]. With enough trails it is then still possible to achieve good performance. Most of these methods look at the signs of the derivatives. Here our dynamics model scored well.

For learning a reward functions from demonstrations the method tried in this work turned out to be not successful. However, an alternative method is described in [31]. This comes with some extra assumptions. A LQR based dynamic programming has to be used, which was the case in this work with the iterative LQR algorithm. In this work and in others on inverse reinforcement learning it is still the case that the features provided to the algorithm are

manually engineered. Deep learning provides a promising way to automatically learn features. In Chapter 6 a first step in this direction was taken.

Perhaps other reinforcement learning algorithms can be used. The iterative LQR algorithm is commonly used for trajectory following. It turns out it is difficult to apply the algorithm for other tasks, because then it is difficult to obtain positive-definite Q and R matrices. Another class of reinforcement learning algorithms that can deal with continuous states and actions are policy gradient methods [37].

Bibliography

[1] Christopher G. Atkeson. *DARPA Robotics Challenge: Team Steel*. 2012. URL: `http://www.cs.cmu.edu/~cga/drc/` (retrieved August 2014).

[2] Pieter Abbeel, Morgan Quigley, and Andrew Y. Ng. "Using Inaccurate Models in Reinforcement Learning". In: *Proceedings of the 23rd International Conference on Machine Learning*. 2006, pp. 1–8.

[3] Pieter Abbeel and Andrew Y. Ng. "Apprenticeship Learning via Inverse Reinforcement Learning". In: *Proceedings of the 21st International Conference on Machine Learning*. 2004, pp. 1–8.

[4] Abdeslam Boularias, Brian Ziebart, and Jan Peters. *Workshop: New Developments in Imitation Learning*. 2011. URL: `http://www.robot-learning.de/Research/ICML2011` (retrieved August 2014).

[5] Andrew Y. Ng and Stuart Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the 17th International Conference on Machine Learning*. 2000, pp. 663–670.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[7] Pieter Abbeel, Adam Coates, and Andrew Y. Ng. "Autonomous Helicopter Aerobatics through Apprenticeship Learning". In: *International Journal of Robotics Research* 29 (2010), pp. 1608–1639.

[8] Pieter Abbeel. "Apprenticeship Learning and Reinforcement Learning with Application to Robotic Control". PhD thesis. Stanford University, 2008.

[9] Yuval Tassa, Tom Erez, and William Smart. "Receding Horizon Differential Dynamic Programming". In: *Advances in Neural Information Processing Systems 20*. 2008, pp. 1465–1472.

[10] J. Zico Kolter, Pieter Abbeel, and Andrew Y. Ng. "Hierarchical Apprenticeship Learning with Application to Quadruped Locomotion". In: *Advances in Neural Information Processing Systems 20*. 2008, pp. 769–776.

[11] Pieter Abbeel, Dmitri Dolgov, Andrew Ng, and Sebastian Thrun. "Apprenticeship Learning for Motion Planning, with Application to Parking Lot Navigation". In: *Proceedings of the International Conference on Intelligent Robots and Systems*. 2008, pp. 1083–1090.

[12] Wikipedia. *Google Driverless Car*. 2012. URL: `http://en.wikipedia.org/wiki/Google_driverless_car` (retrieved May 20, 2012).

[13] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J. Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, Michael Sokolsky, Ganymed Stanek, David Stavens, Alex Teichman, Moritz Werling, and Sebastian Thrun. "Towards fully autonomous driving: systems and algorithms". In: *Intelligent Vehicles Symposium (IV), 2011 IEEE*. 2011, pp. 163–168.

[14] Joseph Funke, Paul A. Theodosis, Rami Hindiyeh, Ganymed Stanek, Krisada Kritayakirana, Chris Gerdes, Dirk Langer, Marcial Hernandez, Bernhard Müller-Bessler, and Burkhard Huhnke. "Up to the limits: Autonomous Audi TTS". In: *Intelligent Vehicles Symposium*. 2012, pp. 541–547.

[15] J. Zico Kolter. "Learning and Control with Inaccurate Models". PhD thesis. Stanford University, 2010.

[16] Pieter Abbeel. *Optimal Control for Linear Dynamical Systems and Quadratic Cost ("LQR")*. 2011. URL: `http://www.cs.berkeley.edu/~pabbeel/cs287-fa11/slides/LQR.pdf` (retrieved August 2014).

[17] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. *Simulated Car Racing Championship: Competition Software Manual*. Tech. rep. 2011.06. Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, 2011.

[18] Bernhard Wymann and Eric Espié. *The Open Racing Car Simulator*. 2012. URL: `http://torcs.sourceforge.net/` (retrieved August 2014).

[19] Daniele Loiacono. *Artificial Intelligence in Racing Games*. 2011. URL: `http://home.deib.polimi.it/loiacono/uploads/Teaching/VDP/VDP2011-Racing.pdf` (retrieved August 2014).

[20] Jan Quadflieg, Mike Preuss, Oliver Kramer, and Günter Rudolph. "Learning the Track and Planning Ahead in a Car Racing Controller". In: *Conference on Computational Intelligence and Games*. 2010, pp. 395–402.

[21] Jan Quadflieg, Mike Preuss, and Günter Rudolph. "Driving Faster Than a Human Player". In: *11th European Conference on Evolutionary Computation in Combinatorial Optimisation*. 2011, pp. 143–152.

[22] Jorge Muñoz, German Gutierrez, and Araceli Sanchis. "A Human-Like TORCS Controller for the Simulated Car Racing Championship". In: *Proceedings of the IEEE Conference on Computational Intelligence and Games*. 2010, pp. 473–480.

[23] Daniele Loiacono, Pier Luca Lanzi, Julian Togelius, Enrique Onieva, David A. Pelta, Martin V. Butz, Thies D. Lönneker, Luigi Cardamone, Diego Perez, Yago Saéz, Mike Preuss, and Jan Quadflieg. "The 2009 Simulated Car Racing Championship". In: *IEEE Transactions on Computational Intelligence and AI in Games* (2010), pp. 131–147.

[24] Albert Diosi and Lindsay Kleeman. "Laser Scan Matching in Polar Coordinates with Application to SLAM". In: *International Conference on Intelligent Robots and Systems*. 2005, pp. 3317–3322.

[25] Adam Coates. "Demystifying Unsupervised Feature Learning". PhD thesis. Stanford University, 2012.

[26] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. "Building High-Level Features Using Large Scale Unsupervised Learning". In: *Proceedings of the 29th International Conference on Machine learning*. 2012, pp. 81–88.

[27] Adam Coates, Honglak Lee, and Andrew Y. Ng. "An Analysis of Single-Layer Networks in Unsupervised Feature Learning". In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*. 2011, pp. 215–223.

[28] Pieter Abbeel, Varun Ganapathi, and Andrew Y. Ng. "Learning Vehicular Dynamics, with Application to Modeling Helicopters". In: *Advances in Neural Information Processing Systems 18*. 2006, pp. 1–8.

[29] Wikipedia. *Suzuka Circuit*. 2007. URL: `http://en.wikipedia.org/wiki/Suzuka_Circuit` (retrieved August 27, 2014).

[30] Sergey Levine and Vladlen Koltun. "Continuous Inverse Optimal Control with Locally Optimal Examples". In: *Proceedings of the 29th International Conference on Machine Learning*. 2012, pp. 41–48.

[31] Taesung Park and Sergey Levine. "Inverse Optimal Control for Humanoid Locomotion". In: *Workshop on Inverse Optimal Control & Robotic Learning from Demonstration at Robotics: Science and Systems (RSS)*. 2013.

[32] Felix Duvallet. *Lecture Notes: Inverse Optimal Control*. 2009. URL: `http://www.cs.cmu.edu/afs/cs/project/ACRL/www/LectureNotes/16899C_lecture08.felixd.pdf` (retrieved August 2014).

[33] Sergey Levine, Zoran Popovic, and Vladlen Koltun. "Nonlinear Inverse Reinforcement Learning with Gaussian Processes". In: *Advances in Neural Information Processing Systems 24*. 2011, pp. 19–27.

[34] Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. "Maximum Entropy Inverse Reinforcement Learning". In: *AAAI Conference on Artificial Intelligence*. 2008, pp. 1433–1438.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[36] Christopher G. Atkeson, Andrew Y. Moore, and Stefan Z. Schaal. "Locally Weighted Learning". In: *Artificial Intelligence Review* 11 (1997), pp. 11–73.

[37] Jan Peters. "Policy gradient methods". In: 5.11 (2010), p. 3698.