THE UNIVERSITY OF AUCKLAND

Path Tracking Control of Non-holonomic Car-like Robot with

Reinforcement Learning

by

Yuming Lin

Supervised by

Jacky Baltes

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

AUCKLAND, NEW ZEALAND

JULY, 1999

©Yuming Lin 1999
# Table of Contents

Table of Contents ................................................. i
List of Figures ...................................................... iv
List of Tables ......................................................... vi
List of Symbols ....................................................... vii
Abstract .................................................................. x
Acknowledgements .................................................. xii

1 INTRODUCTION .................................................. 1
   1.1 What is this Thesis about ? ................................. 1
   1.2 Center for IT and Robotics ................................. 2
   1.3 Environment ...................................................... 4
   1.4 The Car-like Mobile Robot ................................. 4
   1.5 The Aucklandianapolis Race Track ....................... 8
   1.6 Outline of the Thesis ......................................... 8

2 LITERATURE REVIEW .......................................... 10
   2.1 Kinematic Model for a Rear-wheel Driven Car ........... 12
   2.2 Formulation of the Path Tracking Problem ................ 13
   2.3 Existing Path-tracking Controllers ....................... 16
      2.3.1 Feedback Controller .................................... 17
      2.3.2 Sliding Mode Controller and Modified Sliding Mode
             Controller .................................................. 18
      2.3.3 Fuzzy Logic Controller ................................. 20
      2.3.4 Look-ahead Controller ................................. 21
3 REINFORCEMENT LEARNING 23
3.1 Introduction ........................................... 23
3.2 Q-Learning .............................................. 25
3.3 Q(\(\lambda\))-Learning .................................. 28
3.4 Eligibility Trace ....................................... 29
3.5 State Aliasing ......................................... 32

4 RL USING CASE-BASED FUNCTION APPROXIMATORS 33
4.1 Continuous State Representation ....................... 34
  4.1.1 Static Quantization ................................ 34
  4.1.2 Dynamic Quantization ............................. 36
4.2 Function Approximator ................................ 38
4.3 Instance-based Function Approximator ............... 39
  4.3.1 Function Evaluation .............................. 40
  4.3.2 Learning Update ................................. 40
4.4 Case-based Function Approximator .................... 41
4.5 The Algorithm ......................................... 42
  4.5.1 Function Evaluation .............................. 42
  4.5.2 Action Selection Strategy ......................... 48
  4.5.3 Learning Update ................................. 50
  4.5.4 Case Addition .................................. 51
4.6 Summary .................................................. 51

5 RL IN THE PATH-TRACKING PROBLEM 53
5.1 Variables ................................................ 53
  5.1.1 State ................................................ 53
  5.1.2 Action .............................................. 56
  5.1.3 Reward ........................................... 58
5.2 Reward Computation .................................. 61
5.3 Similarity Function ................................... 61
5.4 An Example Function Evaluation ..................... 62

6 EXPERIMENT 65
6.1 Learning to Track a Straight Line ..................... 66
  6.1.1 The Case Distribution ............................ 66
  6.1.2 The Learned Grid Chart ......................... 68
6.2 Learning to Track a Left Arc ........................ 73
  6.2.1 The Case Distribution ............................ 76
  6.2.2 The Learned Grid Chart ......................... 77
6.3 Learning to Track a Right Arc ........................ 78
  6.3.1 The Case Distribution ............................ 78
6.3.2 The Learned Grid Chart .............................. 78
6.4 Learning to Follow the Aucklandianapolis Race Track in Sim-
ulation ...................................................... 86
6.5 Performance in Practical Path Tracking .................. 97
6.6 Experiment with Different Weights in the Reward .......... 102
6.7 Shooting Yourself in the Foot ............................ 106
6.8 Summary .................................................. 113

7 CONCLUSION ............................................. 115
7.1 Conclusions .............................................. 115
7.2 Future Work ............................................ 116

Bibliography ................................................. 121
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>System Configuration</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>The Big Car (Length=40cm)</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>The Small Car (Length=18cm)</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>The Aucklandianapolis Race Track</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Kinematic Model of a Car-like Mobile Robot</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Relative Car Coordinates</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Different Action for Line and Left Arc with the same $\bar{y}$ and $\bar{\theta}$</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Look-ahead Controller Model</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Agent and Environment</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Reinforcement Learning Paradigm</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>The Learned Result of Fig 3.2</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Two Types of Eligibility Trace</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>The State Aliasing Problem</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>A Case of Non-MDT</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Memory-Based Function Approximator</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>The Nearest Neighbor Set $NN_q$</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>Different Scaling Factors for Kernel Function</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>Possible Skewing Functions</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Steering Angle Changes more often before Adding $\Delta U$ to $r$</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Steering Angle Changes less often after Adding $\Delta U$ to $r$</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Learning to Drive a Straight Line in Simulation</td>
<td>60</td>
</tr>
<tr>
<td>5.5</td>
<td>Example: two Cases in the $NN_q$</td>
<td>62</td>
</tr>
<tr>
<td>5.6</td>
<td>One of the Situations for the Previous $NN_q$</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Relative Position and Orientation of the Car and a Line</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>The Case Distribution without the Skewing Function for Straight Lines</td>
<td>69</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Fuzzy Set in the Fuzzy Logic Controller . . . . . . . . . . . . . . 21
4.1 Output Kernel Function for Different $\tau_k^u(N = 9)$ . . . . . . . . 46
5.1 Steering Angle Setting of the Small Car ($L = 12cm$) . . . . . . . . 57
5.2 Example Cases in the Nearest Neighbor set $NN_q$ . . . . . . . . . 63
6.1 Steering Angles Used in Simulation . . . . . . . . . . . . . . . . 66
6.2 Average Errors in Simulation for the Race Track . . . . . . . . . . 97
6.3 Average Error in Practical Driving for the Race Track . . . . . . . 99
6.4 Comparison of different Controllers in Practical Driving . . . . . 102
List of Symbols

\(x, y\): Current position of the car.
\(\theta\): Current orientation of the car.
\(\hat{x}, \hat{y}\): Point on the path that is closest to the car.
\(\hat{\theta}\): Tangent of the path at the point \(\hat{x}, \hat{y}\).
\(\tilde{y}\): Position error.
\(\tilde{\theta}\): Orientation error.
\(\phi\): Steering angle of the car.
\(L\): Axle length of the car.
\(R\): Minimum turn radius of the car.
\(v_1\): Velocity of the car (driving speed).
\(v_2\): Steering speed of the car.
\(X\): State in Reinforcement Learning.
\(U\): Action in Reinforcement Learning.
\(\Delta U\): Change between two consecutive actions.
\(\pi\): Control policy.
\(\pi^*\): the optimal control policy.
\(\arg\max_x\): the value of \(x\) that can maximize \(y\).
\(V^*\): maximum discounted cumulative reward.
\(r(X, U)\): the reward for taking action \(U\) from state \(X\).
\(r\): the reward function.
$w_1$: the weight for position error ($\tilde{t}$).

$w_2$: the weight for orientation error ($\tilde{\theta}$).

$w_3$: the weight for action change ($\Delta U$).

$\alpha$: the learning rate.

$\gamma$: the discounted factor.

$e$: eligibility trace.

$S_\gamma$: the skewing function of $\tilde{y}$.

$C_i$: case $i$ in the database.

$X_i$: the state of case $i$.

$U_i$: the action of case $i$.

$Q_i$: the Q value in case $i$.

$e_i$: the eligibility trace for case $i$.

$N$: number of total actions.

$M$: number of cases in the database.

$U_{ij}$: the $j$th action in case $i$.

$Q_{ij}$: the Q value for the $j$th action in case $i$.

$e_{ij}$: the eligibility trace for the $j$th action in case $i$.

$X_q$: the query state.

$U_q$: the query action.

$NN_q$: the nearest neighbor set for the query state.

$d^s_i$: state (or input) distance from case $i$ to the query state.

$d^a_i$: action (or output) distance from action $j$ to the query action for case $i$. 
\( \tau_k \): the smoothing factor.

\( \tau_d \): the case density parameter.

\( \rho \): the blending factor.

\( K^x \): input kernel function, defining how to scale the input distance.

\( K^u \): output kernel function, defining how to scale the output distance.

\( \text{sign}(x) \): sign of \( x \), i.e. 1 if \( x \geq 0 \), -1 if \( x < 0 \).
Abstract

This thesis describes the design and implementation of a Reinforcement Learning controller for the path tracking problem of car-like mobile robots. The goal of the research was a very practical one: that of driving a small toy car around a race track as fast as possible.

The path tracking problem is the problem of controlling a mobile robot around a path. This problem is very challenging. Practically, a number of factors, such as inaccuracies of actuators and the vision system, slipping and interference, may lead to failure of the control task without a good control algorithm.

A variety of control algorithms exist. Most of these controllers do well in simulation, but show poor performance in practice or require significant time for tuning. The motivation of this thesis is to learn the required control function through trial and error. The robot can learn to drive, by distinguishing and remembering good or bad actions. In the future, good actions are taken, while bad actions are avoided.

The controller in this thesis is based on reinforcement learning. During the learning, the agent is given penalty if it takes a bad action, or reward if it takes a good action.

The Reinforcement Learning controller performs well in simulation, but most importantly, it also exhibits very good performance in practice. The
controller was used to win the 1999 Aucklandianapolis competition.
Acknowledgements

First of all I would like to thank my supervisor Jacky Baltes for all his guidance, patience and encouragement during this period. Whenever I had a problem, he was always there and ready to help. He made many very useful suggestions to me for this project.

I also want to thank Nicholas Hildreth in our IAV lab for his help. He provided me with many useful hints when I was stuck.

The greatest thanks will go to my wife Yanting for all her patience for taking care of our naughty little daughter Amy so I could devote myself to this project.
Chapter 1

INTRODUCTION

1.1 What is this Thesis about?

The aim of this thesis is to design and implement a controller with reinforcement learning for the path tracking of car-like robots, and to compare its performance against some of the existing controllers.

The path tracking problem is the problem of controlling a mobile robot around a path. This problem is very challenging. Practically, a number of factors, such as inaccuracy of actuators and the vision system, slipping and interference, may lead to failure of the control task without a good control algorithm.

This thesis describes the design and implementation of one such path-tracking controller, both for simulation and for practical driving. Its performance is evaluated. It performs well in simulation, but most importantly, it also exhibits very good performance in practice. The controller was used to win the 1999 Aucklandianapolis competition.

The analysis of the experimental evaluation shows the influence of the different weights in the reward function.
CHAPTER 1. INTRODUCTION

This thesis also shows a problem which has not been mentioned in the literature on reinforcement learning but which may occur when learning to track circle. By selecting a suboptimal action, the learner prevents itself from revisiting this state and therefore of ever learning the correct action.

1.2 Center for IT and Robotics

The CITR at the University of Auckland has a mobile robotics lab, which hosts the Aucklandianapolis competition [Bal98]. The goal of the competition is to drive a car-like (non-holonomic) robot five laps around the given race track as quickly as possible. The car is guided by a global vision system. Since the robots used are cheap toy cars, their actuators are very coarse and the position and orientation feedback is noisy. A robust path-tracking controller with good performance is needed for such a task.

The robosoccer team of the lab, which takes part in the international Robocup robosoccer competition [rob], also needs a good controller. The movement of strikers and the goalie is one of the most important factors to win a game. In such a time-critical problem, a good path-tracking controller is of paramount importance. Because of this, a number of different controllers have been implemented at the CITR.

The goal of this project is to find a new controller with good robust performance, that is, a controller that can be used to finish tracking the given path with higher speed and less error than other controllers.
CHAPTER 1. INTRODUCTION

Figure 1.1: System Configuration
CHAPTER 1. INTRODUCTION

1.3 Environment

The setup of our mobile robot lab, which is used for variety of test (e.g., Robosoccer, racing, navigation), is shown in Fig 1.1. The cheap remote-controlled toy car is used as robot. The client-server model is used in our implementation. One obvious benefit is that the vision information generated by the vision server can be shared by many clients. Each of the clients can have its own task, but they need information about either one or all other objects. The vision server captures an image of the playing field, finds the car and other objects in it, converts their image coordinates into world coordinates and broadcasts them to clients via UDP packets. Two circles of different colors are placed on top of the car to simplify the vision task. Our path-tracking controller is running on the client machine. It controls the car by writing controlling commands to the device driver which actually communicates with the transmitter.

1.4 The Car-like Mobile Robot

The mobile robot, or car for simplicity, is a remote controlled toy car with proportional steering and speed control. The control is coarse through the use of low cost D/A converters and imprecise mechanical steering. A parallel port micro-controller based interface[NBM98] allows us to control the car from a computer. In theory, there are 65 steering angle settings( 32 left + 32 right + 1 straight) and 65 speed settings( 32 forward + 32 backward + 1 stop ). In practice, not all values can generate different steering angles. These steering angles are non-linear and asymmetric. We can also use PWM (pulse width modulation) setting to control the speed of the car. There are 16 PWM settings. The PWM setting is the time interval within which the motor of
the car is turned on. For example, if the PWM is 1, the motor is on one-sixteenth of the time. The 0 setting turns the motor on all the time (no PWM control). The addition of the PWM control was necessary since the cars were uncontrollable even at the slowest speed setting.

There are two types of such toy cars, big cars and small cars, as shown in Fig 1.2 and Fig 1.3. The steering angles for the big cars are better in linearity and symmetry. PWM can be used to control the speed of the car. But for the small cars, PWM almost has no effect in the speed control. The steering angles are completely nonlinear and asymmetric.

The controlling commands can be sent to the car via the micro-controller. Actually, the commands are written to the device driver via file I/Os. The device driver for the Linux system has been implemented by Jacky Baltes and is being used in all other mobile robot experiments.
CHAPTER 1. INTRODUCTION

Figure 1.3: The Small Car (Length=18cm)
Figure 1.4: The Aucklandianapolis Race Track
CHAPTER 1. INTRODUCTION

1.5 The Aucklandianapolis Race Track

A path can consist of any number of segments. These segments can be one of three types: straight line, left and right turning arc at the maximum steering angle of the car. These types of paths are returned by popular nonholonomic path planning algorithms, e.g. [BCS95]. As shown in Fig 1.4, the Aucklandianapolis race track has all three types of segment and can be tracked continuously. It is one of the suitable paths as a testbed for path-tracking controllers.

1.6 Outline of the Thesis

Chapter 2 is the literature review. The kinematic model for a rear-wheel driven car and the formulation for path tracking is introduced. Some existing control algorithms for path tracking are also described in the chapter.

Chapter 3 describes the standard reinforcement learning algorithm, $Q$-Learning. An extension to this algorithm, $Q(\lambda)$-Learning, is also introduced in this chapter. Our path tracking controller is based on this learning paradigm.

Chapter 4 describes the design and the implementation of our reinforcement learning algorithm for path tracking. A Case-based function approximator is combined with $Q(\lambda)$-learning algorithm in order to apply reinforcement learning to this problem.

Chapter 5 describes the formalisation of the path tracking problem as a reinforcement learning problem. For example, what are the states, actions and reward function in this specific problem? It describes issues in applying the reinforcement learning algorithm to path tracking, most importantly the representation of continuous states in path tracking.
CHAPTER 1. INTRODUCTION

Chapter 6 presents in detail the evaluation of the learning algorithm, both in simulation and practical driving, and an analysis of the experimental results.

Chapter 7 is the conclusion of this thesis, and shows direction for future research.
Chapter 2

LITERATURE REVIEW

Most approaches found in the literature on nonholonomic car-like robot control use variations of traditional feedback process control. As pointed out by Brockett[Bro83], it is impossible to stabilize a nonholonomic vehicle about a nonsingular configuration by any continuous time-invariant static feedback. However, the path following or trajectory tracking problem is simpler in principle than stabilizing to a point[BBBC96].

Luca et al[LOS96] present results in simulation on the application of different feedback control schemes to car-like vehicles. Performance and convergence can also be found from the simulation result. More details are shown in the following sections. In [LB93], Benedetto and Luca analyze the dynamic state feedback in controlling nonholonomic systems. Advantages and drawbacks with respect to the use of static state feedback laws are also discussed. Sordalen and Egeland[SE93] use non-smooth, time-varying feedback control for the goal of global, asymptotical stability with exponential convergence. Sarkar et al [SYV94] investigate a smooth nonlinear feedback control algorithm for trajectory tracking and path following.

Most implementations use the chained form to represent such nonholo-
nomic system. To demonstrate control algorithms, simulation results are presented in these papers but no practical results are presented.

Since the performance for traditional feedback control algorithms is not good for this specific problem, alternative approaches are pursued. The following sections will first introduce the kinematic model of a car-like robot and the formulation of the path-tracking problem. Then the feedback controller, sliding mode controller, fuzzy logic controller and the look-ahead controller are discussed in detail.
2.1 Kinematic Model for a Rear-wheel Driven Car

The mobile robot (in this case, our toy car) is a two-axle (front and rear) vehicle. There are two wheels on each axis. The front wheels can be steered, and the driving force comes from the rear wheels whose relative orientation to the car body is fixed (i.e., rear-wheel driving).

The generalized coordinates of such a robot are shown in Fig 2.1. It can be represented by a quadruple \( (x, y, \theta, \phi) \), where \( x, y \) are the Cartesian coordinates of the center of the rear wheel, \( \theta \) is the orientation of the car body with respect to the X-axis, and \( \phi \) is the steering angle.

The kinematic model comes from the assumption that wheeled mobile robot can only roll but never slip and that the steering angle is limited. Therefore the velocity orthogonal to the front and rear wheels must be zero. Based on this observation, the nonholonomic constraints can be represented by the following equations:

\[
\dot{x}_f \times \sin(\theta + \phi) - \dot{y}_f \times \cos(\theta + \phi) = 0 \tag{2.1}
\]

\[
\dot{x} \times \sin(\theta) - \dot{y} \times \cos(\theta) = 0 \tag{2.2}
\]

Equ 2.1 is for the front wheel, Equ 2.2 is for the rear wheel. The following equations can also be derived from Fig 2.1:

\[
x_f = x + L \times \cos(\theta)
\]
\[
y_f = y + L \times \sin(\theta)
\]

where \( L \) is the axle distance of the car (i.e., distance between the two axles of the car). Differentiating the first two equations, we get the followings:
\[ \dot{x}_f = \dot{x} - L \times \dot{\theta} \times \sin(\theta) \]
\[ \dot{y}_f = \dot{y} + L \times \dot{\theta} \times \cos(\theta) \]

Substitute \( \dot{x}_f \) and \( \dot{y}_f \) in Equ 2.1 by the two equations listed above, then we get:

\[ \dot{x} \times \sin(\theta + \phi) - \dot{y} \times \cos(\theta + \phi) - \dot{\theta} \times L \times \cos(\phi) = 0 \quad (2.3) \]

If the driving speed is \( v_1 \) and the steering speed is \( v_2 \), and expanding the above equation, the kinematic model of a rear wheel driving mobile can be derived, as shown below:

\[ \dot{x} = v_1 \times \cos(\theta) \]
\[ \dot{y} = v_1 \times \sin(\theta) \]
\[ \dot{\theta} = \frac{v_1 \times \tan(\phi)}{L} \]
\[ \dot{\phi} = v_2 \quad (2.4) \]

### 2.2 Formulation of the Path Tracking Problem

In order to design a controller applicable to all path, the absolute coordinates of the car cannot be used by the controller. What good is a controller that only works for one given path? Absolute coordinates must be transformed into relative coordinates. By using such relative coordinates, the controller becomes independent of the given path.

Another advantage of a relative coordinate system is that it can track the path without having to know the complete path. In some applications, such
as robosoccer, the path may change quickly. These are the prerequisite for the design of a controller for path tracking. Based on these, the following formulation (or representation model) is used in the implementation of our controller.

As shown in figure 2.2, at any time, the car is at position \((x,y)\) and is following a path. \((\hat{x},\hat{y})\) is the closest point on the path to the car. The position error \(\tilde{y}\) is the distance between points \((x,y)\) and \((\hat{x},\hat{y})\), that is,

\[
\tilde{y} = \sqrt{(x - \hat{x})^2 + (y - \hat{y})^2}
\]  

(2.5)

This position error shows how far the car is away from the path. Note that in Equ 2.5, \(\tilde{y}\) is always positive. This is insufficient, as we must allow a negative \(\tilde{y}\) to distinguish whether the car is on the left of the path or on the right of the path (or in the case of an arc, inside the arc or outside the arc).
CHAPTER 2. LITERATURE REVIEW

So the following equation is actually used to calculate $\tilde{y}$.

$$\tilde{y} = (y - \hat{y}) \times cos(\hat{\theta}) - (x - \hat{x}) \times sin(\hat{\theta}) \quad (2.6)$$

It is easy to show that the absolute values of $\tilde{y}$ in Equ 2.5 and Equ 2.6 are equal. This can be shown below:

$$|\tilde{y}| = |PK| = |PT| + |TK|$$
$$= (y - \hat{y}) \times cos(\hat{\theta}) + (x - \hat{x}) \times sin(\hat{\theta})$$
$$= -[(y - \hat{y}) \times cos(\hat{\theta}) - (x - \hat{x}) \times sin(\hat{\theta})] \quad (2.7)$$

$\hat{\theta}$ is the tangent of the path at the point $(\hat{x}, \hat{y})$, $\theta$ is the orientation of the car, $\tilde{\theta}$ is the orientation error of the car, that is,

$$\tilde{\theta} = \theta - \hat{\theta} \quad (2.8)$$

It must be noted that in such a model, the absolute position and the orientation of the car are not important. We are only interested in two errors, that is, $\tilde{y}$ and $\tilde{\theta}$, plus the curvature of the path $R$. They are the inputs to the controller. $R$ is important to the controller, as the controller must have different control function for different curvature. For example, for the same $\tilde{y}$ and $\tilde{\theta}$, there may be completely different action for a straight line and a left arc. As shown in Fig 2.3, even $\tilde{y}$ and $\tilde{\theta}$ are the same, the best action is a right-turn for a straight line, it is a left-turn for a left arc.

Another advantage of such a model is that it can greatly reduce the input space in reinforcement learning, thousands or more absolute positions have the same relative position to the path. This is of great importance to the
learning process. As pointed out in the following chapters, learning speed decreases exponentially with the increase in input space.

Because of this, this kinematic model is a good model in the sense that it makes the controller applicable even in time-critical applications. As detailed in Chapter 6, the value learned from simulation can be used in practical driving. The model defines the position and orientation error as the input instead of the absolute position and orientation. This is the same in all path tracking tasks where a path consists of some of the three types of segment. It can be deduced that the learned result from simulation in which the path consists of all three segment types can be used in all practical driving (for the same car).

2.3 Existing Path-tracking Controllers

Previous work on path tracking controllers include the following:

- Traditional feedback controller [LOS96]
CHAPTER 2. LITERATURE REVIEW

- Sliding mode controller[BBBC96]
- Modified sliding-mode controller
- Fuzzy logic controller[Ott98]
- Look-ahead path tracking controller[EHA98a]

All except the feedback controller have been implemented in the IAV lab. The feedback controller requires continuous second derivative for the path and our Bang-Bang path planner can not generate such path. The following sections will describe briefly each of these controllers.

2.3.1 Feedback Controller

The traditional feedback controller is based on control theory which can be used in all process control. It is not specific to path-tracking and thus can be used in point to point movement and trajectory tracking. We have not explored its performance in real world path tracking. Because it is not a specific controller for path-tracking, its performance is possibly sub-optimal.

In [LOS96], Luca et al presented and compared several feedback solutions for path following, as well as for point stabilization and trajectory following, such as smooth time-varying feedback and non-smooth time-varying feedback, approximate linearization and exact feedback linearization. A lot of simulation results can also be found here.

Luca et al use a (2,4) single-chain form for all their experiments. The representation of a (2,4) chained form is shown below:

\[ \dot{x}_1 = u_1 \]
CHAPTER 2. LITERATURE REVIEW

\[ \begin{align*}
\dot{x}_2 &= u_2 \\
\dot{x}_3 &= x_2 \times u_1 \\
\dot{x}_4 &= x_3 \times u_1
\end{align*} \] (2.9)

where \( u_1 \) and \( u_2 \) are two inputs. In our car-like robot, \( x_1 = x, x_2 = \frac{tan(\phi)}{L \cos(\theta)}, x_3 = tan(\theta), x_4 = y \). With the aid of the chained form, the path following can be implemented via input scaling by smooth time-varying feedback control.

As pointed out by Luca and Oriolo, when applying the proposed feedback controller to mobile robots in practice, the actual behavior (or performance) may be affected by such non-ideal conditions as uncertain kinematic parameters of the car, mechanical limitations (i.e. backlash at the wheels, limited range of steering angle), actuator saturation and dead-zones, noise and quantization errors. All these make it difficult to apply the feedback controller in practice and thus this controller is implemented in simulation only.

2.3.2 Sliding Mode Controller and Modified Sliding Mode Controller

The sliding-mode controller was used in our first trial of the Aucklandianapolis race. It has very good performance in simulation but performs very poorly in practical driving. It only uses two steering angles, namely, full left turn and full right turn, thus it is a Bang-Bang controller. Because of this, the car is only controllable at very low speeds. We modified this controller and used some smoothing over steering angles when close to the path and the performance was much better in the modified sliding-mode controller. Below is a brief introduction to the algorithm of the sliding mode controller.
Let $R$ be the minimum turning radius, $\hat{R}(s)$ be the radius of curvature of the path (or the current segment). The following control law is used:

$$\omega = \text{sign}(\hat{R}(t)) \times \text{sign}(\sigma) \times \frac{v_1}{R}$$

(2.10)

where

$$\sigma = -\frac{\hat{y}}{R} - \text{sign}(\hat{\theta}) \times (1 - \cos(\hat{\theta}))$$

(2.11)

From the kinematic model in Equ 2.4, we have

$$\omega = \hat{\theta} = v_1 \times \frac{\tan(\phi)}{L}$$

(2.12)

Then the steering angle is decided by:

$$\phi = \tan^{-1}\left(\frac{\hat{\theta} \times L}{v_1}\right) = \tan^{-1}\left(\frac{L}{v_1} \times \text{sign}(\hat{R}(t)) \times \text{sign}(\sigma) \times \frac{v_1}{R}\right)$$

$$= \tan^{-1}(\text{sign}(\hat{R}(t)) \times \text{sign}(\sigma) \times \frac{L}{R})$$

$$= \text{sign}(\hat{R}(t)) \times \text{sign}(\sigma) \times \tan^{-1}\left(\frac{L}{R}\right)$$

(2.13)

As shown in [LOS96], we can use $\tan(\phi) = \frac{L}{R}$ to compute the necessary steering angle $\phi$. That is, this controller only uses the maximum steering angle (either left or right) for path tracking control. Due to the delay from the actuator, this full steering angle cannot correct the error immediately and larger error is incurred. Another reason is that the full steering angle may lead to over-steering in case of very small error. As in our daily life, if we always drive with full steering, what a mess it will be! This is the main reason why it performs so poorly in practical driving.
2.3.3 Fuzzy Logic Controller

Because the performance of the sliding-mode controller is so poor in practical driving, even the modified sliding-mode controller is not good enough. Other approaches were pursued in the IAV lab. The fuzzy logic controller, implemented by Robin Otte in our robotics lab as a project with Jacky Baltes, has very good performance in practical driving for the big car. It can track the path twice as fast as the modified sliding-mode controller. The problem for this controller is, it needs a lot of rules (about 135 in Robin’s implementation), which must be hand-crafted.

This controller is a fairly standard fuzzy logic control process. Any fuzzy logic application will execute the following three steps:

- Fuzzify all variables that appear in the conditions of rules, or fuzzification.
- Apply a fuzzy rule set, or inference.
- Defuzzify outputs that appear in the conclusions of rules and apply them, or defuzzification.

The first step will partition all variables (input or output) into fuzzy sets. Although simple in theory, this step is of critical importance to the performance of the fuzzy logic controller. For different cars, these partition may be different. There is another implicit prerequisite. Since the fuzzy logic controller works by interpolating values of different fuzzy sets, these variables must be approximately linear, otherwise it may become even more difficult to design these fuzzy sets. Our big cars are good in this sense, but the small cars are problematic.

Tab 2.1 lists the number of fuzzy sets used in Robin’s implementation.
CHAPTER 2. LITERATURE REVIEW

<table>
<thead>
<tr>
<th></th>
<th>Input Variables</th>
<th>Output Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuzzy Set</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2.1: Fuzzy Set in the Fuzzy Logic Controller

The inference step is more tedious. It is merely a lot of statements consisting of if ... then. Ideally, the if statement should include all the possible permutation for input variables (in this case, it is 9x5x3=135). Robin’s implementation makes use of symmetry thus the total number of rules is less than this. These rules are the same for different cars.

The defuzzification step then calculates the output, usually by the 'center of mass' or 'centroid' defuzzification. More on fuzzy logic operations can be found in [Saf] and [RAK95].

2.3.4 Look-ahead Controller

The Look-ahead path tracking algorithm was implemented by some of Jacky Baltes’s students for the Aucklandianapolis race track and its performance was very good. It is similar to Egerstedt’s controller[EHA98a][EHA98b]. This controller models the system by two cars, one is the actual car being controlled, the other is a so-called virtual car. This can be shown in Fig 2.4.

The virtual car is on the path all the time and some distance ahead of the current position of the practical car (the length of —AB— in the figure). The desired steering angle of the practical car is decided by the difference between the orientation of the actual car ($\theta$) and that of the straight line from the real car to the virtual car ($\beta$). This model is very good for race track path tracking where time is critical, i.e. as fast as possible to finish the race while remaining controllable. But the position error is difficult to control in this
model. This limits its application in those areas where position errors are very important. For example, the controller for a striker may prefer small position errors rather than going faster when it gets close to the ball. On the other hand this controller is good for a striker if the ball is very far away and this striker needs to get close to the ball as quickly as possible in order to get the ball.
Chapter 3

REINFORCEMENT LEARNING

3.1 Introduction

This chapter introduces the foundations of reinforcement learning. Reinforcement learning focuses on the problem of how an autonomous agent, that can sense and act in its environment, learns to choose the optimal actions to achieve its goals [Mit97]. The environment presents at certain time a reward, that is feedback about the quality of previous moves. For example, in the case of playing chess, the agent may learn the best movement of each step in order to win the game. There is only one reward at the end of the game (win, lose or draw). The problem in reinforcement learning is to assign blame. For example, a bad move in chess will force a loss, even if the following actions are optimal. Such a problem can be solved by choosing those actions that can maximize the cumulative reward over time.

The relationship of an agent to its environment can be modelled in Fig 3.1. The environment has a set of possible states \( (X_0, X_1, X_2,...) \). From the initial
state $X_0$, the agent first takes an action $U_0$ and the environment possibly returns a reward $r_0$ as the immediate value of the transition of this state-action. This reward indicates how good the action was. After taking an action, the agent enters another state, takes another action and gets another reward from the environment. This sequence is also shown in Fig 3.1. That is, at time step $t$, $r_t = r(X_t, U_t)$ is the reward given by the environment for action $U_t$ in state $X_t$, $X_{t+1} = \delta(X_t, U_t)$ is the following state. The task for the agent is to learn a control policy that can maximize the expected reward in all the states.

A control policy can be expressed by a function, $\pi : X \rightarrow U$, that maps from the current state $X_t$ to an action $U_t$, that is, $U_t = \pi(X_t)$. This means that it selects the action only based on the current observed state. One policy selects actions by producing the maximum cumulative reward for all states. The cumulative reward for a state $X_t$ is defined by Eq. 3.1. $\gamma$ is a discount factor that determines the immediate reward versus delayed reward, $0 \leq \gamma < 1$. Smaller $\gamma$ (closer to 0) emphasizes the immediate reward, while $\gamma$ closer to 1 emphasizes the delayed reward.
CHAPTER 3. REINFORCEMENT LEARNING

$$V^*(X_t) = r_t + \gamma V(X_{t+1}) + \gamma^2 V^*(X_{t+2}) + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$ (3.1)

The optimal policy is one that maximizes $V^*(X)$ for all states, we use $\pi^*$ to denote such a policy, as defined by Equ 3.2.

$$\pi^* \equiv \arg\max_{\pi} V^*(X), \quad \forall X$$ (3.2)

The cumulative reward in Equ 3.1 for the optimal policy $\pi^*$ becomes $V^{**}(X)$, or $V^*(X)$ to simplify notation. That is, $V^*(X)$ is the maximum discounted cumulative reward that the agent can obtain from state $X$.

3.2 Q-Learning

The optimal control policy introduced in the last section can be learned if an optimal action in the current state is executed. The optimal action is defined by Equ 3.3.

$$\pi^*(X) = \arg\max_u [r(X, U) + \gamma V^*(\delta(X, U))]$$ (3.3)

Here $r(X, U)$ is the immediate reward for action $U$ in state $X$. $\delta(X, U)$ is the successor state after such an action. The optimal action is the one that maximizes the sum of the immediate reward and a discounted $V^*$ of the successor state.

To select the optimal action by Equ 3.3, the immediate reward $r$ for the action and the following state must be known (or must be predicted). This is difficult in many practical problems.

In order to find the feasible optimal action, we first define an evaluation function $Q(X, U)$, which represents the effective reward (immediate and
CHAPTER 3. REINFORCEMENT LEARNING

discounted) of the new state.

\[ Q(X, U) = r(X, U) + \gamma V^*(\delta(X, U)) \] (3.4)

For simplicity, the evaluation function is referred to as \textit{value function} or \textit{Q-value}. Then Equ 3.3 can be rewritten as below:

\[ \pi^*(X) = \text{argmax}_U Q(X, U) \] (3.5)

Equ 3.5 shows that if the \textit{Q}-value for each state-action pair \((X, U)\) is known (learned by the agent), the optimal action can be found. This optimal action is the one that has the highest \textit{Q}-value in the current state \(X\).

From the definition of \(V^*(X)\) and \(Q(X, U)\), it is easy to see that \(V^*(X) = \max_{U'} Q(X, U')\), Equ 3.4 can be changed to:

\[ Q(X, U) = r(X, U) + \gamma \max_{U'} Q(\delta(X, U), U') \] (3.6)

Based on Equ 3.6, we have the Q-learning Algorithm [Mit97], as shown in Algorithm 1.

\begin{algorithm}
\textbf{Algorithm 1} Reinforcement Learning Algorithm
\begin{algorithmic}
\For{each pair of <\(X, U\)>}
\State \(Q(X, U) \leftarrow 0\)
\EndFor
\State Observe the current state \(X\)
\Loop
\State 1. Select an action \(U\) and execute it
\State 2. Receive immediate reward \(r\)
\State 3. Observe the new state \(X'\)
\State 4. Update the table entry for \(Q(X, U)\), as follows:
\[ Q(X, U) = r + \gamma \max_{U'} Q(X', U') \]
\EndLoop
\end{algorithmic}
\end{algorithm}
Figure 3.2: Reinforcement Learning Paradigm

The function update in Eq 3.6 is radical, as the old value is not considered at all and new $Q$ is simply replaced by the expected total discounted return (or just return). An improvement is to use Sutton’s temporal difference, or TD error [Sut88], the difference between the return and the current $Q$-value of $(X, U)$, as defined by Eq 3.7:

$$\Delta Q = r(X, U) + \gamma \max_{U'} Q(\delta(X, U), U') - Q(X, U) \quad (3.7)$$

Thus the $Q$ update is modified, as:

$$Q(X, U) = Q(X, U) + \alpha \Delta Q$$

$$= (1 - \alpha)Q(X, U) + \alpha(r(X, U) + \gamma \max_{U'} Q(\delta(X, U), U')) \quad (3.8)$$

Where $0 \leq \alpha \leq 1$ is the learning-rate (or step-size) which determines how much to learn in each step. The $Q$-learning algorithm simply uses $\alpha = 1$.

An illustrative example of $Q$-learning is shown in Fig 3.2. The black
area is the obstacle. The agent can move either horizontally or vertically. The Goal is to move to the top right corner. Through Q-learning, the agent can find the best movement for each state even if it has no knowledge of the environment. As shown in the figure, only two actions have a reward of 1 (the left action of $X_{n-1}$ and up action of $X_m$), as these two actions can lead to final state $X_n$. All other actions have a reward of 0. Initially, All $Q$ values are zero, the agent moves randomly in the environment. If the agent goes to state $X_m$ and takes the up action, $r(X_m, UP) = 1$, so $Q(X_m, UP) = 1$. Next time if the agent goes to state $X_{m-1}$ and assuming the discount factor $\gamma=0.9$, then $Q(X_{m-1}, \text{RIGHT}) = 0 + 0.9 \times \max_{U'} Q(X_m, U') = 0 + 0.9 \times 1 = 0.9$. Given enough experimentation, the agent can eventually learn the best $Q$-values for each pair $(X, U)$. The $Q$-values can back-propagate from the goal state to the initial state step by step. The learned result is shown in Fig 3.3. In this Figure, the best movement from initial position to the goal can be found for any state by selecting those actions having the highest $Q$-values (the number by the arrow standing for an action). For example, the route marked by the dotted line is one such movement that will lead the agent to the goal.

For the $Q$-learning algorithm to converge (that is, the true $Q$-values are learned), one of the requirements is that the agent must visit every possible state-action pair infinitely often[Mit97].

### 3.3 $Q(\lambda)$-Learning

The algorithm described in the last section looks ahead just one step in the $Q$-value update and thus it is called one-step $Q$-learning. This can be extended to use a multi-step return, or TD($\lambda$) return[Sut88]. The $Q(\lambda)$-learning algorithm is based on this idea, it integrates the one-step $Q$-learning
algorithm and TD(\(\lambda\)) return and works significantly better than the one-step Q-learning\[PW96\]. The Q-learning is just a special case of Q(\(\lambda\))-learning with \(\lambda = 0\). This Q(\(\lambda\))-Learning algorithm is shown in Algorithm 2.

In Algorithm 2, step 5. and step 8. are used for one-step update, they are the same as that used in Q-learning update (the modified version, as shown in Equ 3.7 and Equ 3.8). Step 6. and 7. are used for TD(\(\lambda\)) update. \(\epsilon\) is the eligibility trace which is used to deal with the delayed reward, and it is described in next section.

### 3.4 Eligibility Trace

Eligibility traces\[BR83\] are a mechanism to handle the delayed reward. In most reinforcement learning problems, there exists the delay between an action and its effective reward. For example, in our path-tracking problem, the reward (or error to the path) may take several steps before it becomes obvious if the car is set to turn left fully from the current maximum steering
Algorithm 2 $Q(\lambda)$-Learning Algorithm

for each pair of $<x, u>$ do
    $Q(x, u) \leftarrow 0$
    $e(x, u) \leftarrow 0$
end for

loop
    1. Observe the current state $X_t$
    2. Select an action $U_t$ that maximizes $Q(X_t, a)$ for all $a$ and execute it
    3. Receive immediate reward $r_t$
    4. Observe the new state $X_{t+1}$
    5. $\Delta Q' = r_t + \gamma V^*_t(X_{t+1}) - Q_t(X_t, U_t)$
    6. $\Delta Q = r_t + \gamma V^*_t(X_{t+1}) - V^*_t(X_t)$
    7. For each pair of $<x, u>$
        (A) $e(x, u) = \gamma \lambda e(x, u)$
        (B) $Q_{t+1}(x, u) = Q_t(x, u) + \alpha e(x, u) \Delta Q$
    8. $Q_{t+1}(X_t, U_t) = Q_{t+1}(X_t, U_t) + \alpha \Delta Q'$
    9. $e(X_t, U_t) = e(X_t, U_t) + 1$
end loop

angle to the right. The delay can not even be eliminated by the $TD(\lambda)$ return. The above $Q(\lambda)$ algorithm thus adopts the strategy of eligibility traces.

The idea of an eligibility trace is somehow similar to our human memory system. When we come across something, we will remember it for sometime. We will forget it gradually if we have no chance to visit it again. Once it appears again, it can also refresh our memory. Similarly, in reinforcement learning, when a state is visited, it initiates a short-term memory process, a trace, and decays gradually over time. But if this state is visited again, the trace is updated or assigned another new value. Such a trace marks the learning eligibility of the state.

Eligibility traces decay gradually according to a given decay-rate, $\lambda$, $0 \leq \lambda \leq 1$, and a discount-rate, $\gamma$, which is the same as that used in Q-learning. There are two types of eligibility trace, as shown in Fig 3.4. The first one is
the conventional accumulating trace. Whenever a state is visited, a value is added to the old trace (accumulated). It can be defined by:

$$e_{t+1}(x) = \begin{cases} 
\gamma \lambda e_t(x) & \text{if } x \neq x_t \\
\gamma \lambda e_t(x) + 1 & \text{if } x = x_t 
\end{cases}$$

The second one is a replacing eligibility trace, as introduced by Singh and Sutton [SS96]. In this case, whenever a state is visited, the new value simply replaces the old value. It can be defined by:

$$e_{t+1}(x) = \begin{cases} 
\gamma \lambda e_t(x) & \text{if } x \neq x_t \\
1 & \text{if } x = x_t 
\end{cases}$$

The $Q(\lambda)$-learning algorithm described above utilises the conventional accumulating trace, while our implementation on path-tracking takes the other approach. As shown by Singh and Sutton, the replacing trace has a lower mean squared error for the $Q$-value in the long run and may be applicable more generally [SS96].
CHAPTER 3. REINFORCEMENT LEARNING

Figure 3.5: The State Aliasing Problem

$A, B, C$ are the same state. The action cannot be decided by state alone.

3.5 State Aliasing

If the action cannot be decided by the state alone in a reinforcement learning system, then it has the state aliasing problem. This can be demonstrated by Fig 3.5. Suppose the agent in such a system has only four sensors located at its front, rear, left and right side. These four sensors can detect the existence of the wall in front of it. Their inputs define the current state of the agent. So state $A$, $B$ and $C$ are the same state (with left and right sensor activated). But the best action cannot be decided for the state only. For example, at state $C$, the best action is to go down, while in state $A$ and $B$ the best action is to go upwards.

As shown in the previous chapter, the curvature of the path is necessary to avoid the state aliasing problem.
Chapter 4

RL USING CASE-BASED FUNCTION APPROXIMATORS

This chapter introduces two varieties of memory-based function approximator: instance-based function approximator and case-based function approximator. They are very similar. The difference in their storage structure is the main source for other differences during the learning process. This chapter first describes the representation of continuous states and function approximators, followed by a brief introduction to instance-based function approximators, whose presence here is only as an introduction to case-based function approximators, which is described in detail later. Finally, the algorithm for reinforcement learning using a case-based function approximator is presented.

A case used in a memory-based function approximator is a memory element that keeps a history of the agent during learning, such as what state and action it has experienced in the past, and most importantly, the associ-
CHAPTER 4.  RL USING CASE-BASED FUNCTION APPROXIMATORS

An initial reward (Q-value). These cases are used in the evaluation of the Q-value for new states and are updated during learning. Cases may be added to the memory if implicit information is missing from the case database, as described later.

4.1 Continuous State Representation

The two algorithms described in Chapter 3 assume a discrete-valued state where the learning algorithm can be implemented easily through a look-up table. In the problem of path-tracking, the input \( X = \langle \hat{y}, \hat{\theta}, R \rangle \) is continuous (both \( \hat{y} \) and \( \hat{\theta} \) are real-valued, \( R \) is discrete)\(^1\). So this is a continuous-state problem. There must be some kind of mechanism to 'discretize' the state before the learning algorithm can be applied to such a problem. Two approaches are discussed here.

4.1.1 Static Quantization

Static quantization is the direct approach. First the desired resolution for the input must be defined. The input is then discretized by this resolution. This works exactly in the same way as an analog-digital converter (ADC). After quantizing the input state into a finite number of cells, the discrete-state learning algorithm can be applied.

There are some problems with this approach. Firstly, it is difficult to get the optimum resolution. Too fine a resolution unnecessarily increases learning time, while not enough resolution leads to large control errors. For example, if the resolution for a position is 0.05\( m \), and for angles is 5°, and assume the range for the position error \( \hat{y} \) is \([-4m, 4m]\) (see Section 5.1.3),

\(^1\)A state is continuous if any variable in it is real-valued
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

then the cell number for all three types of segment would be:

$$\frac{8}{0.05} \times \frac{360}{5} \times 3 = 34560$$

This state size is much bigger than that in dynamic approach, as described in the next section.

The second problem is that it may create convergence problems because they form a non-Markovian representation of the dynamics of the system [JRAal][MA95]. The Q-learning only applies to Markovian Decision Task (or MDT) [Bel57], and it will create convergence problem when using the Q-learning in non-Markovian decision problem [Mit97]. This can be illustrated in Fig 4.1. In this figure, Cell C1 has the learned preferred action of left-turn, while C2 has a right-turn. What should be the action at input $X_\theta$? If the best action at $X_\theta$ is going straight, no matter how long the learning will be, the correct action for $X_\theta$ will never be learned. This problem can be solved through a finer resolution. But it is difficult to find the right resolution.
to overcome such problems while keeping the resolution not too fine for fast convergence.

4.1.2 Dynamic Quantization

This approach does not partition the state space into cells. The value function can be approximated through the use of function approximators. There are many types of function approximator (see next section). Memory-based function approximator [JRAal] is one type of them. The input space is not really discretized here, instead, it is just approximated by some existing memory elements. Each memory element stores information of some state-action pairs that the agent experienced before. Such memory element is also referred to as case. When the learning starts, there are no cases in the memory. The number of cases grows as the learning proceeds. Each case represents some input space, a circle with this case as the center, whose radius is a predefined density parameter \( \tau_d \) (shown in Fig 4.2).
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

During the learning, for a query point $X_q$, a nearest neighbor set $NN_q$ is generated. This $NN_q$ is defined by a circle with the query point $X_q$ as the center, and the radius is a predefined smoothing parameter $r_k$. These cases in $NN_q$ are similar to the query point. The value for the query point can be computed from all the cases in $NN_q$. Since some cases in $NN_q$ are nearer, or more similar, to the query point, they should contribute more to the computed value. This can be done easily by a weighting factor, whose value is based on the distance from the case to the query point. If the nearest case to the current query point is further than the density constant $r_d$, a new case is created for this query point.

As detailed in Chapter 6, our dynamic quantization approach creates just about 1100 cases for the same average position and angle error as that stated in the previous section. The size of the state space is much reduced, which reduces the time for learning greatly. There are three reasons why less cases are created by the dynamic quantization. Firstly, it never generates cases that do not occur in learning. For example, the case with $\tilde{y}=1m$ and $\tilde{\theta}=180^\circ$ is one of such cases. With this static quantization, the Q-value for each action in each cell must be learned independently, and cells in the neighbor will not be used as reference. The dynamic approach can overcome this problem, the Q-value for a action can even be approximated in a new state from its neighborhood. So the cell density can be reduced. This is the second reason. The third reason is that our implementation uses non-uniform resolution. So more cases are created for those states nearly to the path, fewer cases when far away from the path.

Memory-based function approximators include instance-based function and case-based function approximators. They are very similar. Their different case structure is the source of all other subtle differences. Based on the
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

A case structure, a case-based function approximator is used for the problem of path-tracking. More details are presented in the next chapter.

4.2 Function Approximator

When the state is continuous or its size is too big, the value function $Q$ can be approximated. That is to say, a function approximator is used to provide $Q$-value for state-action pairs for continuous state (or action).

Besides the memory-based function approximator, other types include neural networks\cite{Lin92}, statistical models\cite{TAH96} and state space self-partitioning\cite{Nak97}. Some of these approaches are very good. For example, Nakamura's self-partitioning algorithm can reduce the size of the state space efficiently, by recursively splitting continuous a state space into tentative states. If a tentative state is relevant by a statistical test, it is partitioned into finer states. As pointed out by Nakamura, if the dimension of the state space is higher, the computational time and memory requirements are enormous, and as a result the learning might not converge correctly.

A function approximator can be used as representation in continuous state (action) problem. The value function can even be accurately estimated for new states. Non-uniform resource allocation can be easily implemented by the use of a skewing function (see below).

Memory-based function approximator has a simple model. It is computationally efficient\cite{JRAal} and easy to implement, as compared with other function approximators. So it is used in our implementation. More details on other function approximators can be found in \cite{JRAal}.
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

4.3 Instance-based Function Approximator

So the definition of a case for an instance-based function approximator is as follows:

\[ C_i = (X_i, U_i, Q_i, \epsilon_i) \quad i = 1 \cdots M \]  \hspace{1cm} (4.1)

Where M is the number of existing cases (which grows when a new case is created).

That is, each case consists of one state-action pair \((X_i, U_i)\), the \(Q\)-value \((Q_i)\) for it, and the eligibility trace \(\epsilon_i\). This structure is very similar to the lookup table used for discrete states and actions (see Alg 2). The main difference is the way in which cases are used to compute the \(Q\)-value of query states. The former evaluates \(Q(X_q, U_q)\) by a set of cases in the nearest neighbor to the query point \(NN_q\) and a similarity function based on the distance from the case in \(NN_q\) to the query point is used to add up different \(Q\)-values to generate \(Q_{X_q, U_q}\). Standard reinforcement learning evaluates \(Q(X_q, U_q)\) just by searching the single cell for the query input \(X_q\) among all its associated actions. The neighboring cells will not be considered. So the instance-based function approximator can estimate the \(Q\)-values even for new states by looking at its nearest neighbors, while the discrete state and action learning algorithm cannot do this and simply takes a random action for the state. This makes the instance-based function approximator more robust in practice. On the other hand, the computational cost is increased due to the more complex evaluation.
4.3.1 Function Evaluation

There is only one similarity measure function (distance) used. This function is defined as the distance from the query point \((X_q, U_q)\) to the point \((X_i, U_i)\) in case \(C_i\). So this distance is a function of the state space and action space. There is only one kernel function \(K\) used for the function evaluation.

The first step is to search the memory to find the nearest neighbor set \(NN_q\) using the distance \(d_i\). The second step is to evaluate the \(Q\) value for the query point \((X_q, U_q)\), by the equation listed in Equ 4.2.

\[
Q(X_q, U_q) = \sum_{\forall C_i \in NN_q} \frac{K(d_i)}{\sum_j K(d_j)} Q_i
\]  \hspace{1cm} (4.2)

4.3.2 Learning Update

Equ 4.3 and Equ 4.4 are used here for the \(Q\) value update. Equ 4.5 is used to update \(e_i\) in a case.

\[
\Delta Q = r_t + \gamma \hat{Q}_{t+1} - \hat{Q}_t
\]  \hspace{1cm} (4.3)

\[
Q_i = Q_i + \alpha \Delta Q e_i
\]  \hspace{1cm} (4.4)

\[
e_i = \begin{cases} 
\frac{K(d_i)}{\sum_j K(d_j)} & C_i \in NN_q \\
\lambda \gamma e_i & otherwise
\end{cases}
\]  \hspace{1cm} (4.5)
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

4.4 Case-based Function Approximator

The case structure for a case-based function approximator is shown in Eq. 4.6.

\[ C_i = (X_i, Q_i, e_i, \{U_{ij}, Q_{ij}, e_{ij}\}) \quad i = 1 \cdots M, \quad j = 1 \cdots N \] (4.6)

where \( M \) is the number of existing cases, \( N \) is the number of actions.

A case-based function approximator is an extension of instance-based function approximator, that in addition to information about the state alone, maintains information about individual actions. In this case structure, there are two separate parts. The first part \((X_i, Q_i, e_i)\) is about the state space and has nothing to do with the actions. For the state represented by \( X_i \), there is a general \( Q\)-value \( Q_i \) for it and the eligibility trace \( e_i \) used for the learning process. The value of \( Q_i \) can be used to measure how good the state space \( X_i \) is. For example, a state \((\tilde{y}, \tilde{\theta}, R) = (1.0, 0, 1)\) has a higher \( Q_i \) than that in \((1.5, 0, 1)\) as the first state has less position error and thus is a better (more desirable) state.

The second part \( \{U_{ij}, Q_{ij}, e_{ij}\} \) is about the actions for the state space in the case. For the \( j \)th action \( U_{ij} \), there is a \( Q_{ij} \) value associated with it and the eligibility trace \( e_{ij} \). The value of \( Q_{ij} \) can be used to measure how good the action \( U_{ij} \) is for this specific state space \( X_i \).

Using this case structure, there are much fewer cases in the database than that in the instance-based function approximator, as in the latter, many cases (corresponding to the same state but with different actions) are merged into one case. As will be shown later in this chapter, the case-based function approximator requires more computation in the function evaluation.
4.5 The Algorithm

This section will focus on the application of reinforcement learning with a case-based function approximator. As stated earlier, the path-tracking problem has continuous states, and due to the drawbacks that may arise if the static quantization is used (see Sec 4.1.1), a case-based function approximator is especially suitable for such a problem.

The algorithm of \(Q(\lambda)\)-learning with a case-based function approximator is listed in Alg 3. This algorithm is a modified version of the \(Q(\lambda)\)-learning algorithm as listed in Alg 2, by adding the case-based function approximator mechanism. The following sections will explain this algorithm in more detail. The specifics of how to apply a case-based function approximator to path-tracking are shown in Chapter 5.

4.5.1 Function Evaluation

Whenever a query point \((X_q, U_q)\) needs the value function for the current input \(X_q\), the database is searched for those states that are similar to the query state. The distance \(d_x^q\) from an existing state \(X_i\) to the query state \(X_q\) can be used for the estimation of similarity. The nearest neighbor set \(NN_q\) consists of those cases whose state distance to \(X_q\) is less than a predefined threshold \(\tau_k\). That is

\[NN_q = \{C_i|d_x^q <= \tau_k\} \quad (4.7)\]

This step can be shown in Fig 4.3.

The distance from state \(X_i\) to the query state \(X_q\) is noted as \(d_x^q\) instead of \(d_{x_q}\) to avoid confusion \(^2\). As can be seen later, there are two types of

\(^2\)In the usual sense, two subscripts are used for distance.
Algorithm 3 $Q(\lambda)$-Learning with case-based function approximator

Define inputs $(\hat{y}, \hat{\theta}, R)$ and their ranges
Define output $(u)$ and its range
Set case number to 0

loop

Observe the current state $X_q$
Search memory to get nearest neighbor set $NN_q(\tau_k)$

for each action $U_q$ in all actions do
    for each case $i$ in $NN_q$ do
        Calculate the overall $Q$-value of this case for $U_q$ by:
        \[ Q_i(U_q) = (1 - \rho)Q_i + \rho \left( \sum_{\forall u_{ij} \in C_i} \frac{K^*(d^j)}{K^*(d^j)} Q_{ij} \right) \quad \forall C_i \in NN_q \]
    end for

    Calculate the $Q$-value for $(X_q, U_q)$ by:
    \[ Q(X_q, U_q) = \sum_{\forall C_i \in NN_q} \frac{K^*(d^j)}{K^*(d^j)} Q_i(U_q) \]
end for

Select the action $U_q$ that has the highest $Q(X_q, U_q)$, $Q_{t+1}$ and execute it
Receive immediate reward $r_t$
Calculate the change in $Q$ by:
\[ \Delta Q = r_t + \gamma Q_{t+1} - Q_t \]

for $i$ in all existing cases do
    $Q_i = Q_i + \alpha \Delta Q e_i$
    \[ e_i = \begin{cases} 
        (1 - \rho) \frac{K^*(d^j)}{K^*(d^j)} & C_i \in NN_q \\
        \lambda \gamma e_i & \text{otherwise}
    \end{cases} \]

    for $j$ in all actions do
        $Q_{ij} = Q_{ij} + \alpha \Delta Q e_{ij}$
        \[ e_{ij} = \begin{cases} 
            \rho \frac{K^*(d^j)}{K^*(d^j)} & C_i \in NN_q \\
            \lambda \gamma e_i & \text{otherwise}
        \end{cases} \]
    end for
end for

Create a new case if closest case is further than $\tau_d$.

end loop
Figure 4.3: The Nearest Neighbor Set $NN_q$
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

... distances defined here. One is the input distance, \( d_i^x \), defined by Equ 5.5. The superscript \( x \) stands for the input. The other is the output distance, \( d_{ij}^u \). This is the difference between two actions in case \( i \) (the given action \( j \) and the query action \( q \)). This is defined by Equ 4.8. The superscript \( u \) stands for the output (the action).

From \( NN_q \) in Equ (5.5), all existing cases in the database that are similar to the current input \( X_q \) can be found, thus the Q value for the query point \( <X_q, U_q> \) can be calculated by the following formulas:

\[
d_{ij}^u = |U_j - U_q|  \tag{4.8}
\]

\[
Q_i(U_q) = (1 - \rho)Q_i + \rho \left( \sum_{\forall u, j \in C_i} \frac{K^u(d_{ij}^u)}{\sum_j K^u(d_{ij}^u)} Q_{ij} \right) \quad \forall C_i \in NN_q  \tag{4.9}
\]

\[
Q(X_q, U_q) = \sum_{\forall C \in NN_q} \sum_j K^x(d_{ij}^x) Q_i(U_q)  \tag{4.10}
\]

In Equ 4.9, \( \rho \) is the blending factor. As there are two portions of Q values in a case, \( \rho \) is used to combine these two portions of Q values to get a single Q value for the query action for this case. \( K^u \) is the output kernel function, which is used to determine the relative contribution of an action \( j \) in case \( i \) to the query action \( U_q \). The \( K^u \) shown in Equ 4.11 is one possible function. The further the distance \( d_{ij}^u \) is, the smaller is the function value and thus this action contributes less to the overall Q value. Yet there is a problem when using Equ 4.11 in experimentation. The difference of computed output kernel value between different actions is not enough. This is shown in Tab 4.1. So the action distance \( d_{ij}^u \) is scaled by a factor \( \tau_i^u \) in Equ 4.11. This is shown in
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

<table>
<thead>
<tr>
<th>$d_{ij}^u$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^{-\left(\frac{d_{ij}^u}{N}\right)^2}$</td>
<td>1.000</td>
<td>0.988</td>
<td>0.952</td>
<td>0.895</td>
<td>0.821</td>
<td>0.734</td>
<td>0.641</td>
<td>0.546</td>
<td>0.454</td>
</tr>
<tr>
<td>$e^{-\left(\frac{d_{ij}^u \times 2}{N}\right)^2}$</td>
<td>1.000</td>
<td>0.952</td>
<td>0.821</td>
<td>0.641</td>
<td>0.454</td>
<td>0.291</td>
<td>0.169</td>
<td>0.089</td>
<td>0.042</td>
</tr>
<tr>
<td>$e^{-\left(\frac{d_{ij}^u \times 4}{N}\right)^2}$</td>
<td>1.000</td>
<td>0.821</td>
<td>0.454</td>
<td>0.169</td>
<td>0.042</td>
<td>0.007</td>
<td>0.001</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Table 4.1: Output Kernel Function for Different $\tau_k^u (N = 9)$

Equation 4.12 and this equation is actually used in our experiment (with $\tau_k^u = 4$). A larger $\tau_k^u$ emphasized more on the similar action (with less $d_{ij}^u$). The scaling effect can also be shown in Table 4.1 and in Figure 4.4.

$$K^u(d_{ij}^u) = e^{-\left(\frac{d_{ij}^u}{N}\right)^2}$$  \hspace{1cm} (4.11)

$$K^u(d_{ij}^u) = e^{-\left(\frac{d_{ij}^u \times 2}{N}\right)^2}$$  \hspace{1cm} (4.12)

where $N$ is the number of total actions.

Similarly, $K^x$ in Equation 4.10 is the input kernel function. This is used to determine the relative contribution of a case to the $Q$-value for the query state. This is based on the distance from this case to the query state, as defined by $d_{ij}^x$. The further this distance, the less this case will contribute to $Q(X_q, U_q)$. Equation 4.13 is one of such feasible functions. $\tau_k^x$ in this equation has the similar effect on the input kernel function.

$$K^x(d_{ij}^x) = e^{-\left(d_{ij}^x \times \tau_k^x\right)^2}$$  \hspace{1cm} (4.13)
Figure 4.4: Different Scaling Factors for Kernel Function

The kernel function is $e^{-\left(\frac{d_u}{\tau_k} \times \tau_k\right)^2}$
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

4.5.2 Action Selection Strategy

After computing \( Q(X_q, U_q) \), the \( Q \) value for all possible actions in the current state \( X_q \) is known. Usually the action having the highest \( Q(X_q, U_q) \) is used as the current action for the input \( X_q \).

However, if the highest \( Q \) value is always used, especially in the early stages of learning, when the agent explored just a few states, it will never have the chance to experience more states and is confined only to the existing states. So possibly better actions will never be explored and learned.

So some exploration is desired instead of just exploitation to avoid such premature convergence. Such exploration actions may be selected by some other mechanism instead of simply by the \( Q \) value. There are many suggested methods on selecting such actions, such as counters, recency and error based exploration[Thr92] or exploration bonus in dyna-Q[Sut90]. Three methods most often used in the literature are Boltzmann distribution, pseudo-stochastic choice and pseudo-exhaustive method[CD97].

- **Boltzmann distribution.** An action \( U_q \) has the following probability of being selected for the current state \( X_q \):

\[
p(U_q) = \frac{e^{\frac{Q(X_q, U_q)}{T}}}{\sum_{j=1}^{N} e^{\frac{Q(X_q, U_j)}{T}}} 
\]  

(4.14)

where \( N \) is the number of actions, \( T \) is a constant called *temperature*. It controls the degree of randomness in selecting an action and thus is used for the tradeoff between exploitation and exploration. Higher \( T \) emphasizes exploration, as it makes the \( Q \) value less important.

- **Pseudo-stochastic.** The action having the highest \( Q \) value has a predefined probability \( P_s \) of being selected. If this action (with the highest \( Q \})
is not selected, the agent will select an action randomly (with uniform distribution) in its action set, including the action with the highest $Q$
value.

- **Pseudo-exhaustive.** The action having the highest $Q$ value has an pre-defined probability $P_e$ of being selected. If this action (with the highest $Q$) is not selected, the agent will select the least recently used (LRU) action.

Among these three most often used strategies, *Boltzman distribution* has the worst performance [CD97]. The agent will have too many exploration actions far from the goal and have not enough near the goal. This is because of the non-linearity of Eqn 4.14. As shown by Caironi and Dorigo, two states $X1$ and $X2$ with the same ratio between $Q$-values in each state, if the average $Q$ value in $X1$ is higher, then the action with the highest $Q$ value in $X1$ will have a higher probability of being selected by Eqn 4.14. As average $Q$ values for those states near the goal are higher, the action with highest $Q$ value has a higher probability to be selected in these states, exploitation is much emphasized, and thus leading to not enough exploration.

Caironi and Dorigo also show in their experiments that *pseudo-exhaustive* and *pseudo-stochastic* perform much better than *Boltzman distribution*, and *pseudo-exhaustive* is a little bit better than *pseudo-stochastic* (and less computational overhead) [CD97]. However, for our problem of path-tracking, the *pseudo-stochastic* method is used. One reason is that there is just little difference in the performance for the *pseudo-stochastic* and *pseudo-exhaustive*. Another reason is that it is easier to implement *pseudo-stochastic* method, as there is no need to keep an activity record of each action in each state to know which action is the least recently used one. It is very difficult to argue which one of these two methods is superior. No theoretic proof has
been found to attempt to show which one is better. Some experiments use a pseudo-stochastic approach, others use pseudo-exhaustive.

4.5.3 Learning Update

All Q-values in the database must be updated after a new reward is returned from the environment for the given action. \( TD(\lambda) \) update is used here, as shown by Eqn 4.15. Eqn 4.16 is used to update the Q value for the state. \( \alpha \) is the learning rate or stepsizes for this update. Notice the influence of eligibility of state \( e_i \) on the change on \( Q_i \). If a state was visited recently, the \( e_i \) for this state is usually higher, so the \( Q \)-value for this state \( (Q_i) \) may change more. Replacing eligibility is used here, as shown by Eqn 4.17. If a case \( C_i \) is in the \( NN_q \), the \( e_i \) for this case is replaced by the amount \( (1 - \rho) \frac{K^e(q_e)}{\sum K^e(q_e)} \) instead of 1 as shown in Sec 3.4. This amount is directly related to the contribution of the \( Q \)-value of the state \( (Q_i) \) to the \( Q \)-value of the query point \( (Q(X_q)) \), as shown by Eqn 4.10 and Eqn 4.9. If a state is not selected to \( NN_q \), it contributes nothing to \( Q(X_q) \) and the eligibility of this state \( e_i \) is further discounted by a decaying factor \( \lambda \) (which is 0.6 in our implementation).

The update of the action-related information is very similar to the state-related information shown above, and Eqn 4.18 and Eqn 4.19 are used for this update.

\[
\Delta Q = r_t + \gamma \hat{Q}_{t+1} - \hat{Q}_t \quad (4.15)
\]

\[
Q_i = Q_i + \alpha \Delta Q e_i \quad (4.16)
\]
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

\[ e_i = \begin{cases} 
(1 - \rho) \frac{K^z(d^x_i)}{K^x(d^x_i)} & C_i \in \text{NN}_q \\
\lambda \gamma e_i & \text{otherwise} 
\end{cases} \quad (4.17) \]

\[ Q_{ij} = Q_{ij} + \alpha \Delta Qe_{ij} \quad (4.18) \]

\[ e_{ij} = \begin{cases} 
\rho \frac{K^x(d^x_{ij})}{K^x(d^x_{ij})} & C_i \in \text{NN}_q \\
\lambda \gamma e_i & \text{otherwise} 
\end{cases} \quad (4.19) \]

4.5.4 Case Addition

When the system starts, there are no cases in case database. More cases are added to the memory later as learning proceeds.

\( \tau_d \) is used to control the case density in state space. A new case is generated if the distance from the query state to the closest case \( (d^x_{min}) \) is still further than the predefined case density constant \( \tau_d \), as shown by Equ 4.20. Because \( \tau_d \) is predefined (static), cases are created evenly across the entire state space.\(^3\)

\[ d^x_{min} = \min_i \{d^x_i\} \quad (4.20) \]

4.6 Summary

This chapter presented the algorithm and all other details for the application of reinforcement learning using a case-based function approximator. Although the case-based function approximator is more complex and has a

\(^3\)Due to the existence of skewing factor, the actual density is not the same in state space, see 5.1.1
CHAPTER 4. RL USING CASE-BASED FUNCTION APPROXIMATORS

higher computational cost, it is used for our path-tracking problem instead of the instance-based function approximator. The main reason is that the case-based function approximator, separate distance and two kernel functions are used for the state space and the action space. As the state space and the action space are two completely different spaces, it is better to separate them for the similarity measurement as that case-based function approximator, and to avoid the problem of different resolutions for these two spaces. Another consideration is that in the case-based function approximator, the $Q$ value evaluation for a query action ($U_q$) may get a weighted reference from other actions within a state by the action distance. This is also a result from two separate distance function.

The following chapter will describe the adaption of the algorithm introduced in this chapter to path tracking problem. An example of function evaluation is also shown.
Chapter 5

RL IN THE PATH-TRACKING PROBLEM

This chapter describes the adaptation of reinforcement learning to the path-tracking problem, with a focus on variables used in reinforcement learning. More low level implementation details are presented in Chapter 4.

5.1 Variables

The most important concepts used in all reinforcement learning include state, action, and reward. What are they in this specific problem of path-tracking control? Some explanations for this are given in this section.

5.1.1 State

In Fig 3.1, the agent has two inputs: state and reward. These two inputs are needed for the $Q$-learning or $Q(\lambda)$-learning algorithm, as described in
Chapter 3. The reward is computed (synthesized) by the system. The sensor input decides the internal learning state. So without causing confusion, *input* and *state* are used interchangeably in the remaining part of the paper.

As mentioned in Chapter 2, The kinematic model defines the position error ($\tilde{y}$) and the orientation error ($\tilde{\theta}$). These two real-valued variables are the most important input to the controller. For different types of path, the best actions may not be the same. For example, for the same $\tilde{y}$ and $\tilde{\theta}$, there may be different best action. So to avoid the state aliasing problem, another input must be added to the state in this problem. The curvature $R$ of the segment being followed can be used for this purpose. It can distinguish different types of segment. Three types of segment are considered in our implementation. The discrete-valued $R$ may have either 0 (for left-turn arc segment), 1 (for straight line segment) or 2 (for right-turn arc segment). This representation can simplify the distance calculation, as will be explained in details in next chapter.

The 3-tuple $< \tilde{y}, \tilde{\theta}, R >$ can be used as the input to the controller. However, more cases are preferred when the car is almost on the path (that is, $\tilde{y} \approx 0$). Similarly, fewer cases are needed when the car is far away from the path (that is, $\tilde{y}$ has a large value). This can be done by a skewing function on $\tilde{y}$. Fig. 5.1 depicts three possible functions. A function similar to $x^{\frac{1}{2}}$ is used as the skewing function for $\tilde{y}$, as shown by Equ 5.1. So the input $< S_y, \tilde{\theta}, R >$ is the actual input to the controller. It defines the current state used in the reinforcement learning.

$$S_y = \begin{cases} \sqrt{\tilde{y}} & \text{if } \tilde{y} \geq 0 \\ -\sqrt{-\tilde{y}} & \text{if } \tilde{y} < 0 \end{cases}$$ (5.1)

It can be seen from the state definition, the agent only 'sees' the current
Figure 5.1: Possible Skewing Functions.
CHAPTER 5. RL IN THE PATH-TRACKING PROBLEM

segment being tracked. \( \hat{y}, \hat{\theta} \) and \( R \) are all defined by the current segment. The agent knows nothing about the next segment. Just like that in our daily driving, if we come to a curve and still drive at the 'high' speed as that in a straight line, it is not good at all. So information of next segment, including distance to the next segment and type of the segment, may be added to the model, especially when driving the car at high speed and if speed is added to the learning. This may help avoid overshooting the current segment. But at present, such information is not needed. There is no learning in this project for the speed, The car is assumed to go at slow speed. This is why the information of the next segment is not included in our experiment.

Furthermore, the definition of the state avoids any learning for the same type of segments in a path if they are at difference places. If absolute position and orientation information is used, the agent must learn for the same type of segment at different place. This will make the learning controller algorithm impossible to be applied in real-time path-tracking control, as for each dynamically generated path by the path planner, the agent may need to spend one day to learn to drive. In time-critical application such as robosoccer, no matter how well you learn to drive, the game would have finished after such a long time, what is the use of this controller?

5.1.2 Action

The action is the output of the controller. In path-tracking, it can be the steering angle or the speed of the car (or both). Only the steering angle is considered in our project. The car is assumed to go at a constant speed without any learning in speed control\(^1\). As described in Chapter 1, there

\(^1\)In practice, the speed must be controlled dynamically to keep it constant, as the battery level is changing all the time. However, it is not included in the learning
are 65 possible steering angles. If all of the steering angles are used for the learning, the state and action space \((X, U)\) is very big, as detailed in 4.1.1. This will lead to a very slow convergence. So only the most significant nine steering angles are used as the action for the learning. The steering angle is not linear, and it is asymmetric for the left-turning and right-turning. Details of a small car are listed in Table 5.1 as a result from measurements taken in a test (other small cars of the same model have similar characteristics. The calculation of the steering angle \(\phi\) is based on the following formula:

\[
\phi = \arctan\left(\frac{L}{R}\right)
\]

where \(L\) is the axle distance of the car (distance between front and rear axis), \(R\) is the maximum steering angle.

This data is not needed if the learned result in simulation is not used for practical driving. But it is still necessary to pre-define a certain steering angle for each action before learning. In such case it is only needed to assign a steering angle to each action number (see Chapter 6 for simulation angles). But they are of crucial importance if the learned function values are applied to practical driving. The values in the row Setting are used in the command to control the car. Notice in both cases the maximum steering angles (left and

<table>
<thead>
<tr>
<th>Setting</th>
<th>Left</th>
<th>Line</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radius(R) (cm)</td>
<td>31</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>Angle (\phi) (°)</td>
<td>28.6</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Used?</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Action Number</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.1: Steering Angle Setting of the Small Car \((L = 12\text{cm})\)
right) must be included as the radius of an arc segment is at the maximum steering angle of the car.

5.1.3 Reward

The reward has the most important impact on the behavior of the learner. Whether a learning system is successful or not depends largely on the assignment of the reward.

\[ r = -w_1 \left( \frac{\tilde{y}}{8} \right)^2 + w_2 \left( \frac{\tilde{\theta}}{2\pi} \right)^2 + w_3 \left( \frac{\Delta U}{9} \right)^2 \]  

(5.3)

\[ \Delta U = U_{t+1} - U_t \]  

(5.4)

Here \( \Delta U \) is the control work between two consecutive actions.

In our work, the reward for the current action is based on three factors: the position error(\( \tilde{y} \)), the orientation error(\( \tilde{\theta} \)) and the change of action (or control work needed for the action). Since they have different importance, each of them is scaled by a different weighting factor\( (w_1, w_2, w_3) \) before summing them up. Since the ranges for these three variables are different, they
are normalized first before adding them together. Here we assume the range for $\tilde{y}$ is $[-4m, 4m]$ which is enough in our experiment. If in some rare case $\tilde{y}$ is not in the range, it will be truncated.

The presence of $\tilde{y}$ and $\tilde{\theta}$ in Equ 5.3 is obvious. If the action leads to a larger error, the reward for it is smaller. The presence of $\Delta U$, on the other hand, is used to force the controller to learn a smooth control function. In other words, small changes in steering angle are preferred. This is very important for practical driving. The car may become uncontrollable at high speeds if the steering angle changes sharply (i.e., from full left to full right). This is one of the reasons why the car can only be controlled at very low speeds with the sliding-mode controller. This has been demonstrated in the experiment (both in simulation and practical driving), as shown in Fig 5.2 and Fig 5.3 when trying to track a straight line. As shown in Fig 5.2, before adding $\Delta U$ to the reward, the steering angle changes sharply from full right (0) to full left (8) more often than after $\Delta U$ is added.

The ratio among $w_1$, $w_2$ and $w_3$ is very important. For example, if $w_3$ is too large, the controller may learn not to change steering angle even when needed. This can be shown in Fig 5.4 for $w_1 = 1$, $w_2 = 1$ and $w_3 = 1$. In this figure the car is reluctant to change its steering angle, and it will receive heavy penalty if it does so. With such values, from Equ 5.3, assuming the three terms are equal, it can be inferred that 1 step change in action is equivalent to 45 cm in $\tilde{y}$, or 40° in $\tilde{\theta}$. This means the the controller will not change the steering angle by 1 step until there is more than 45 cm in position error (or 40° in orientation error). This is the exact reason why the car tracks a straight line in such an unusual way.

Usually a reward with larger value is better. So for larger $\tilde{y}$, $\tilde{\theta}$ or $\Delta U$, the reward should be worse (smaller). That is why a negative sign (−) is needed
Figure 5.4: Learning to Drive a Straight Line in Simulation
The car is reluctant to change its steering angle because of the penalty of a steering change is too large.
5.2 Reward Computation

After executing the selected action, the agent receives reward from the environment. Equ 5.3 is used to compute the reward in the path-tracking. To get better estimation of the reward, the rewards in the last five steps are used and averaged as the current reward. This can reduce the error in computing this reward, as there may be large errors for the position and orientation of the car for a single step from the vision server (and thus bigger error in $\hat{y}$ and $\hat{\theta}$). That does not mean using more previous rewards is a better approach to reduce such error further. If too many rewards from previous steps are used to compute the current reward, this reward will be affected heavily by previous actions and may not show correctly how good or bad the current action is.

5.3 Similarity Function

The similarity measure (or distance) for cases $d_{ij}^k$ introduced in Chapter 4 is defined as:

$$d_{ij}^k = \sqrt{\left(\frac{\hat{y}_i - \hat{y}_q}{8}\right)^2 + \left(\frac{\hat{\theta}_i - \hat{\theta}_q}{2 \pi}\right)^2 + \left(\frac{R_i - R_q}{2}\right)^2} \quad (5.5)$$

The distance is based on three parts: distance error difference ($\hat{y}_i - \hat{y}_q$), orientation error difference ($\hat{\theta}_i - \hat{\theta}_q$) and the curvature difference ($R_i - R_q$). As the three parts have difference ranges, before summing them up, they must by normalized first. The distance error is normalized to $-4.4$ meter, the orientation error to 360 degrees, and the curvature to 0..2. The range
for the position error from 4 to 4 meter is enough for the experiment. In rare case, if the car is 4 meters away from the path, the position error ($\tilde{y}$) is truncated either 4 or -4, depending on the car’s relative position to the path.

### 5.4 An Example Function Evaluation

In this section, an example is given to show how to evaluate the $Q$-value for an input $X_q = <0.5, 0, 1>$ and find the optimal action for it (without considering exploration here). To avoid tedious computation, two simplified kernel functions are used instead of those two shown by Equ. 4.12 and Equ. 4.13, and assume there are just three actions instead of nine (which are used in our implementation). These three actions are left-turn(0), go-straight(1) and right-turn(2).

Let $K^u(d_{ij}^u)$ be :

$$K^u(d_{ij}^u) = \begin{cases} 
1 & d_{ij}^u = 0 \\
0 & otherwise 
\end{cases}$$
<table>
<thead>
<tr>
<th>Case</th>
<th>$y_i$</th>
<th>$\theta_i$</th>
<th>$R_i$</th>
<th>$Q_i$</th>
<th>$Q_{i0}$</th>
<th>$Q_{ii}$</th>
<th>$Q_{i2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>0.9</td>
<td>0.3</td>
<td>1</td>
<td>-0.2</td>
<td>-0.8</td>
<td>-0.6</td>
<td>-0.3</td>
</tr>
<tr>
<td>$C_2$</td>
<td>0.1</td>
<td>-0.1</td>
<td>1</td>
<td>-0.1</td>
<td>-0.7</td>
<td>-0.1</td>
<td>-0.2</td>
</tr>
</tbody>
</table>

Table 5.2: Example Cases in the Nearest Neighbor set $NN_q$

instead of that one shown in Equ 4.12. Equ 4.9 becomes $Q_i(U_q) = (1 - \rho)Q_i + \rho Q_{\theta_i}$. This means that only the $Q$ value of the action that is the same as the query action is considered. Let $K^x(d^e_i) = d^e_i$ instead of the one shown in Equ 4.13, also let $\rho = 0.6$ in Equ (4.9),

Assume that after searching the database, only two cases are in $NN_q$, as shown in Figure 5.5. Table 5.2 shows details of the cases in $NN_q$.

All these changes and assumptions are to simplify computation but to maintain the intension behind case-based function approximator.

The first step is to compute the distance from a case to the query input $X_q$. It is also needed to compute $NN_q$. This is shown below by Equ 5.5:

$$d^e_1 = \sqrt{\left(\frac{0.9-0.5}{8}\right)^2 + \left(\frac{0.3-0}{2\pi}\right)^2 + \left(\frac{1}{2}\right)^2} = 0.069$$
$$d^e_2 = \sqrt{\left(\frac{0.1-0.5}{8}\right)^2 + \left(\frac{-0.1}{2\pi}\right)^2 + \left(\frac{1}{2}\right)^2} = 0.052$$

The distance from $C_2$ to $X_q$ is shorter than that from $C_1$ to $X_q$. $C_2$ will contribute more to the $Q$-value for the query point.

The second step is to compute (or synthesize) a unique $Q$ value for a specific action for $C_1$ and $C_2$, as shown below by Equ 4.9:

$Q_1(0) = (1 - \rho)Q_1 + \rho Q_{\theta} = (1 - 0.6) \times (-0.2) + 0.6 \times (-0.8) = -0.56$

$Q_1(1) = (1 - \rho)Q_1 + \rho Q_1 = (1 - 0.6) \times (-0.2) + 0.6 \times (-0.6) = -0.44$

$Q_1(2) = (1 - \rho)Q_1 + \rho Q_2 = (1 - 0.6) \times (-0.2) + 0.6 \times (-0.3) = -0.26$

Here $Q_1(2)$ has the highest value, so action 2(right-turn) is the preferred
(or best) action for case $C_1$.

\[
Q_2(0) = (1 - \rho)Q_2 + \rho Q_0 = (1 - 0.6) \cdot (-0.1) + 0.6 \cdot (-0.7) = -0.46
\]

\[
Q_2(1) = (1 - \rho)Q_2 + \rho Q_1 = (1 - 0.6) \cdot (-0.1) + 0.6 \cdot (-0.1) = -0.1
\]

\[
Q_2(2) = (1 - \rho)Q_2 + \rho Q_2 = (1 - 0.6) \cdot (-0.1) + 0.6 \cdot (-0.2) = -0.16
\]

Here $Q_2(1)$ has the highest value, so action 1 (go-straight) is the preferred (or best) action for case $C_2$.

The last step is to compute the average $Q$ value for each action from all the cases in $NN_q$ by Eq 4.10.

\[
Q(X_q, 0) = \frac{d^t}{d_1 + d_2}Q_1(0) + \frac{d^t}{d_1 + d_2}Q_2(0) = \frac{0.069}{0.121}(-0.56) + \frac{0.052}{0.121}(-0.46) = -0.52
\]

\[
Q(X_q, 1) = \frac{d^t}{d_1 + d_2}Q_1(1) + \frac{d^t}{d_1 + d_2}Q_2(1) = \frac{0.069}{0.121}(-0.44) + \frac{0.052}{0.121}(-0.1) = -0.29
\]

\[
Q(X_q, 2) = \frac{d^t}{d_1 + d_2}Q_1(2) + \frac{d^t}{d_1 + d_2}Q_2(2) = \frac{0.069}{0.121}(-0.26) + \frac{0.052}{0.121}(-0.16) = -0.22
\]

As the $Q(X_q, 2)$ has the highest value, the agent will take action 2, namely turn right when the input $X_q$ is $<0.5,0,1>$.
Chapter 6

EXPERIMENT

This chapter presents experimental results, from both simulation and practical driving. The quality of learning can be evaluated by checking the best action for any query state, which in turn can be examined in the learned grid value output. We will first introduce the learning of the three basic segments, including the states generated after learning and the grid value chart. Then the experiment with the Aucklandianapolis race track both in simulation and practical driving is presented. The last part presents experiment result with different weights in the reward function. Problems found during the experiment are also discussed in this chapter.

Due to the asymmetry of the steering angles of our car for the left-turn and the right-turn, the angles listed in Tab 5.1 were not used in the learning of the three basic segments in simulation. Instead, Tab 6.1 is actually used for this purpose. This makes the analysis easier, especially when analysing the grid value chart.

Unless otherwise specified, the default radius for the arc to be tracked is 20cm. The radius of the maximum turning circle is 15cm.
CHAPTER 6. EXPERIMENT

<table>
<thead>
<tr>
<th>Radius $R$ (cm)</th>
<th>Left</th>
<th>Line</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>20</td>
<td>32</td>
<td>65</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: Steering Angles Used in Simulation

6.1 Learning to Track a Straight Line

The learning of a straight line is the most basic, yet the most important one. This is because normally most paths consist of more line segments than arc segments. A line a more difficult to track than full-turn circles, where it is almost impossible to turn too far. Another reason is that the result from learning a straight line can be used to analyse the learning quality more easily, as the negative and positive position errors are symmetric, which is not the case for arcs.

The relative position and orientation of the car and the path is depicted by Fig 6.1. As can be seen from the figure, when the car is on the left of the path, the position error ($\tilde{y}$) is positive. Similarly, when the car is on the right of the path, $\tilde{y}$ is negative. If the car is facing too far to the left, its orientation error is positive.\(^1\)

6.1.1 The Case Distribution

Fig 6.2 shows the learned case database. The X-axis is $S_y$, as shown by Equ 5.1. Note that the actual unit of X-axis is neither $cm$ nor $m$. Internally, the unit of $\tilde{y}$ is $m$, it is converted to $cm$ in all the following charts for convenience. The unit of $S_y$ is $\sqrt{m}$. It is normalized by multiplying 100 in

\(^1\)This is the same representation as in Balluchi’s controller [BBBC96]
the chart to have the same magnitude as \( \tilde{y} \) whose unit is \( cm \) externally. For simplicity, it is referred to as \( cm \) in this figure and other figures if \( S_y \) is used.

As can be seen from the figure, cases are generated uniformly across the entire state space. The density is controlled by the predefined constant \( \tau_d \). The best action (with the highest \( Q \) value) in the case is plotted in the figure. Notice that for some cases the best action is not correct at all. For example, the case with \( S_y \approx 5, \tilde{\theta} \approx 10 \) in the first quadrant has the best action of 'slight-Left' instead of right-turn. One possible reason is that these cases are not visited often enough for them to converge to the correct value, and the \( Q \) values for those cases are usually small. But it is still good enough to track any given path. We can see later from the grid chart the best action is still correct in most of these regions (there are some wrong action though). This is because the best action for a query state is not based on a single case. But the cases in the neighbourhood are also considered and referenced when evaluating the \( Q \) value for this query state.

The effect of the skewing function can be seen from Fig 6.3. Here the X-axis is the actual position error (\( \tilde{y} \)). Although cases are distributed evenly
across the whole state space, due to the effect of the skewing function, case
density near $\tilde{y} = 0$ is much higher. The case density is much lower for larger
position error. Because of this property, when the car is near the path, more
cases will be referenced to evaluate the $Q$ value for the the best action and
thus a finer grained control function. While the car is far away from the
path, fewer cases are needed.

6.1.2 The Learned Grid Chart

The grid value chart can be used to estimate the performance of the learned
controller. The behaviour of the controller is exactly based on this grid chart.
Fig 6.4 is the grid chart with full range orientation error $[-180^\circ, 180^\circ]$, and
the position error from $-200cm$ to $200cm$. From this figure, we can see that
in the first quadrant, most actions are full-right. This is correct as in this
region the car is on the left of the path and points away from the path,
so it should make the full-right turn. Similarly, in the third quadrant, most
actions are full-left turn. The change is roughly along the main diagonal line.
Most of the actions are correct, but the agent does learn some wrong actions.
For example, the case for state $\tilde{y} = 0, \tilde{\theta} = 0$, or just $(0,0)$ for simplicity, has
the wrong action (turning left instead of going straight). From the state
distribution diagram in Fig 6.2, most actions near this important point are
not good or optimal, some are even wrong, they all contribute to the best
action at this point and thus lead to a wrong action.

Fig 6.5 zooms in part of Fig 6.4 that is of more importance to the perform-
ance of the controller. It shows regions with small position and orientation
errors. From this figure, it is more obvious that actions near $(0,0)$ are not
optimal. For example, let us have a look at those points on the X-axis. These
states have no orientation error. At $\tilde{y} = 2cm$ or $4cm$, the best action is a
Figure 6.2: The Case Distribution without the Skewing Function for Straight Lines

Cases are distributed evenly over the state space.
Figure 6.3: The Case Distribution with the Skewing Function for Straight Line

The effect of the skewing function is it changes the case density, with more cases near the path.
Figure 6.4: The Grid Chart for Straight Line
The central region of interest is magnified in Fig 6.5.
Figure 6.5: Magnified Grid Chart for Straight Lines
Points on the X-axis are used to analyze the learning quality.
full-right turn. The optimal action should be a slight-right turn. The best action full-left turn at \( \hat{y} = 6cm \) and 8cm is also wrong. On the right side of the Y-axis, for \( \hat{y} \) from 0 to \(-10cm\), the best action is the same left turn, while from \(-12cm\) to \(-28cm\), the best action is a slight-left turn. The change doesn’t happen gradually, as expected. The optimal action for small position errors without orientation error should be a slight turn, either left or right. These non-optimal actions make the practical driving with the simulation result unsatisfactory, and often the car overshoots the path when moving fast, as shown in Fig 6.6. This overshooting limits the performance in practical driving. This overshooting problem is not a problem in simulation. This is because in simulation the car moves at constant speed, and there is no delay in the sensors and the actuators. The learned control function assumes the real car goes at this speed too. This is hard to achieve in practical driving, it is impossible to maintain a constant speed during practical driving.

Despite the non-optimal actions or even some wrong actions locally, the overall action is correct and it is able to learn to track any given path. The tracking of a straight line in simulation is shown in Fig 6.7. The initial position is randomised in the range \([-200cm, 200cm]\), and the orientation is randomised in the range \([-180^\circ, 180^\circ]\). Because of this random initial position and orientation, all states can be explored by the car.

### 6.2 Learning to Track a Left Arc

The relative position and orientation of the car and the path can be depicted by Fig 6.8. When the car is inside the arc, the position error \( \tilde{y} \) is positive. It is negative if it is outside of the arc. The orientation error \( \tilde{\theta} \) is positive if the car is pointing inside the arc, compared with the tangent line at the point
Figure 6.6: Overshooting the Path in Practical Driving
Figure 6.7: Tracking the Line in Simulation after 2000 Trials
Despite its initial state so far away from the path, the car can approach and track the path well.
on the arc that is the closest to the car, otherwise it is negative.

### 6.2.1 The Case Distribution

Fig 6.9 shows the learned cases. The maximum $\tilde{y}$ is the radius of the arc, which is 0.2m in our setup. As $S_y$ is the X-axis, so the maximum value for it is about 0.45, or 45 after normalisation, as can be seen in the figure.

Most learned best actions near (0, 0) are left turns, which is roughly correct. For example, let us look at the point $S_y \approx 10, \tilde{\theta} \approx 0$. The equivalent $\tilde{y}$ is about 1cm. This is almost on the path. The learned best action is a sharp-left turn, which is the optimal action. The point $S_y \approx 0, \tilde{\theta} \approx 25^\circ$ has the best learned action of a right turn. In such cases, the car is on the path, but pointing inward, so the action right turn is good. In contrast, the point $(S_y \approx 22, \tilde{\theta} \approx 0)$ has a $\tilde{y}$ of 5cm inside of the arc and the learned best action is a full-left turn. This is not the optimal action, a left turn or even straight is a solution.

The effect of the skewing function can be seen in Fig 6.10. As can be seen
from this figure, the maximum position error $\tilde{y}$ 20cm is the radius of the arc. This is due to our problem formulation which is based on the closest point on the path to the car. Clearly, there can be no point inside of a circle that is further than the radius from the closest point.

6.2.2 The Learned Grid Chart

The learned grid chart is shown in Fig 6.11. Because there is no state for $\tilde{y}$ larger than 20cm, most area on the right side of the Y-axis has no best action. Due to the effect of the nearest neighbour set, there is still a best action in the region from [20cm, 40cm] for $\tilde{y}$.

The zoomed in region of interest is shown in Fig 6.12. Most best actions in the third quadrant are full left turns which is correct. In this quadrant, the car is outside the arc and points away from the circle, so the car should make full-left turn to get back to the path. Notice the gradual change of action in the second quadrant. The best actions change from full left in (0,0) to sharp left, then to left, and to slight left. In the case of arc, it is hard to see which action is the optimal one. From this magnified grid chart, no obviously wrong best action can be found. The learning quality is better than that of a straight line. This may be due to the fact the full turns are easier to ?? the straight line.

The best action at (0,0) is non-optimal, which should be a sharp left. Though it is not optimal, combined with other actions for those states near (0,0), it is good enough to track the given left arc.

Fig 6.13 is the trajectory for tracking a left arc with the center at (100,20) and a radius of 20cm. Despite the fact that the initial position is more than one meter away from the path, the car can still approach and track the path nicely. The trajectory is quite smooth too, due to the penalty from a change
CHAPTER 6. EXPERIMENT

of action.

6.3 Learning to Track a Right Arc

The relative position and orientation of the car and the path is shown in Fig 6.14. Here when the car is outside of the arc, the position error is positive, contrary to the case of a left arc. So is the orientation error. It is positive if the car is facing outward.

6.3.1 The Case Distribution

Fig 6.15 is the direct output of learned case. Here the minimum $S_y$ is about -45. It is interesting to see that this figure is more or less symmetric with Fig 6.9 about point (0,0). For example, in the first quadrant of this figure, most best actions are full right turns, while in the third quadrant of Fig 6.9, most best actions are full left turns. This will be discussed in more detail in the following section.

The case distribution with $(\tilde{y}, \tilde{\theta})$ as input is shown in Fig 6.16. The minimum position error $\tilde{y}$ is -20cm, which is the radius of the arc.

6.3.2 The Learned Grid Chart

The learned grid chart is shown in Fig 6.17. As shown in the last section, the state distribution chart for the right arc is almost symmetric with the left arc about (0,0). Here we would look into this in more details.

Compare Fig 6.17 with Fig 6.11, it is easy to see from these two figures that the best actions in the first quadrant of one figure are almost the mirrored action of that in the third quadrant of another figure, so are the second
Figure 6.9: The Case Distribution without the Skewing Function for Left Arc

There is no case when $S_y > 50$. 
Figure 6.10: The Case Distribution with the Skewing Function for Left Arc
There is no case for \( \tilde{y} > 20 \).
Figure 6.11: The Grid Chart for a Left Arc ($R = 20cm$)
States with $\tilde{y} > 50$ are not explored (such states don’t exist).
Figure 6.12: Magnified Grid Chart for a Left Arc
The best action at (0,0) should be sharp left instead of full left. Most actions are correct.
Figure 6.13: Tracking the Left Arc in Simulation after 2000 Trials
Despite its big initial position error, the car can approach and track the circle.

Figure 6.14: Relative Position and Orientation of the Car and Right Arcs
Figure 6.15: The Case Distribution without the Skewing Function for a Right Arc

There is no case if $S_g$. 
Figure 6.16: The Case Distribution with the Skewing Function for a Right Arc

No state exists if \( \tilde{y} < -20 \).
quadrant and fourth quadrant. For example, *full right* turn is mirrored to
*full left* turn, *sharp right* to *sharp left*, and so on.

This is what is expected. The states for the left arc and the right arc
should be symmetric, except the actions are opposite for the same position
error and orientation error. The learned grid charts shown in Fig 6.11 and
Fig 6.17 are symmetric about the origin, since the sign of the position and
orientation errors are inverted (see Fig 6.8 and Fig 6.14). In the case of left
arc, if the car in inside the arc, the position error $\tilde{y}$ is positive, but it is
negative for right arc. Same is the orientation error.

A closer inspection shows that due to differences in case distribution and
exploration, Fig 6.17 and Fig 6.11 are not exactly symmetric. For example,
the best action (*full right*) at (10,-10) in Fig 6.17 is not the opposite of the
best one (*left*) in Fig 6.11.

Because of the symmetry, it is not necessary to learn right and left arcs
separately. It is better to just learn and create cases for the left arc and
mirror the best action in the case of a right arc (and invert the sign of error
for them), or vice versa.

Fig 6.18 is the magnified region of interest. Fig 6.19 shows the tracking
of right arc (actually a circle defined as a right arc). This simulation result
is quite satisfactory. Despite the great error in position and orientation, the
arc can still be approached and tracked.

6.4 Learning to Follow the Aucklandianapolis
Race Track in Simulation

Learning to track any given path consisting of the three types of segment can
start from scratch (i.e. empty case database), just as the learning for each
Figure 6.17: The Grid Chart for a Right Arc
States with $\hat{y} < -50$ are not explored, these states don't exist. The central region of interest is magnified in in Fig 6.18.
Figure 6.18: Zoomed in Grid Chart for Right Arc
The symmetry can be easily seen when this is compared with Fig 6.12.
Figure 6.19: Tracking a Right Arc in Simulation after 2000 Trials
basic segment. The result in simulation for following the Aucklandianapolis race track is shown in Fig 6.23.

Due to the current kinematic model and the reward function, each segment is independent of any other segment (even the following segment). The distance for different types of segments (see Eq 5.5) is so far that cases for different types of segments will never be included in the nearest neighbor set $NN_q$ for the current segment being tracked. For example, the distance from left arc (with $R_i = 0$) to straight line (with $R_i = 1$) is 4 meters of the equivalent position error, or 180° orientation error. The threshold $\tau_k$ for the $NN_q$ is much smaller than this.

Because of this, it is possible to learn the cases for all three basic types of segments first. Then these learned cases can be applied to the driving of any path consisting of the three basic types of segments (for simplicity, we call this segment-first learning). This is more efficient than learning the path as a single segment (or full-path learning). More importantly, all states can be experienced in the learning of the basic type of segment, by the great randomization of the initial state. In the case of full-path learning, only the states for the first segment can be effectively fully explored, while the states for the other two basic segments can not be fully explored. This is because for a successful trial, the exploration ratio can not be too high (usually 0.1 or 0.2 at most). After the successful learning of the first segment (otherwise no chance to switch to the following segment), the car is in a relatively good state (i.e. with small position and orientation error) when it gets to the next segment (of a different type). Even with a small ratio of exploration, it is not enough to fully explore all the states for this segment. Actually, the exploration ratio is 1 for the first segment because of the random initial state.

These two approaches have been verified in simulation and the learned
CHAPTER 6. EXPERIMENT

results are very similar. Fig 6.23 is the result of the segment-first learning. Fig 6.24 is the result of the full-path learning. Obviously, the cases generated in segment-first learning can be applied to all path tracking. But the cases generated from full-path learning for a path consisting of all the three basic types of segment can still be applied to other path tracking problem, even there may be some states unexplored in this approach (as long as the new path starts with the same segment type as the learned path, these states may not be visited probably). Fig 6.25 is the tracking result in simulation with the cases learned in the Aucklandianapolis race track in the segment-first trial. As can be seen from the figure, the path can be tracked nicely.

Fig 6.20, 6.21, 6.22 are the case distribution charts in the full-path learning for the Aucklandianapolis race track (5000 trials with 20 percent exploration, followed by 1000 trials of no exploration). Most states can be explored with enough trials, yet some states remain unvisited, as can be found easily from Fig 6.22 where there is white space instead of cases (i.e. \( S_y = -20, \tilde{\theta} = 150 \)).

The average error for simulation is shown in Tab 6.2 for the Aucklandianapolis after the first 3000 trials and another 100 trials are run to compute the average error. To estimate the average error, the initial random state has been limited, with \( \tilde{y} \) from \(-20\) cm to \(20\) cm, and \( \tilde{\theta} \) from \(-\pi/8\) to \(\pi/8\). Notice that the average error is also dependent on the speed of the car and the step interval. Higher speed or longer step intervals lead to a higher error. The car may be uncontrollable if the speed is too high and the step interval is not short enough. Here the speed is 0.4 m/s and the interval is 0.1s.
Figure 6.20: Learned Cases for a Left Arc in the Race Track

In full-path learning, the distribution of generated cases is almost the same as that in segment-only learning. But there are still some unexplored states (e.g. region near (0, -160)).
Figure 6.21: Learned Cases for a Line in the Race Track
Since the first segment is a line, due to random initial position and orientation, all states can be explored.
Figure 6.22: Learned Cases for Right Arc in the Race Track
There are more states unexplored, compared with left arcs and lines. In the
path, only one segment is a right arc. The number of trials is still not
enough to explore all states for a right arc.
Figure 6.23: Tracking the Path with Basic Segments Learned first
Average $\bar{y} = 2.65\text{cm}$, Average $\bar{\theta} = 5.8^\circ$. 
Figure 6.24: Tracking the Path with Learning from Scratch after 3000 trials
Average $\bar{y} = 2.95\text{cm}$, Average $\bar{\theta} = 5.6^\circ$. 
### 6.5 Performance in Practical Path Tracking

The path tracking with reinforcement learning has shown good performance in simulation. It is impractical to train the car in real world driving. Some main reasons are:

- It is infeasible to do the learning in practice from scratch. It may require at least 3000 trials to train the car. Most of all, the amount of manual labor to reset the car into its desired initial state when the current trial failed is very large.

- There are many learning parameters that need to be tuned, it is hard to find the best ones for practical driving. This is much easier in simulation. You change one parameter and restart one experiment with several thousand trials, and you can see the effect of this change. However, the best parameters for simulation may not be the best for practical driving. For example, in the reward function of Equ 5.3 best weight for an action change may be completely different for simulation and practical driving. It does not matter if the controller changes its action often in simulation, but it is crucial in practical driving. This is the main reason why the sliding mode controller is so bad in practice,

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Trials</th>
<th>Average ( \hat{y}(m) )</th>
<th>Average ( \theta(\text{radius}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>400</td>
<td>0.2684</td>
<td>0.3202</td>
</tr>
<tr>
<td>2</td>
<td>800</td>
<td>0.2126</td>
<td>0.2802</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>0.1334</td>
<td>0.1381</td>
</tr>
<tr>
<td>4</td>
<td>1500</td>
<td>0.1062</td>
<td>0.1043</td>
</tr>
<tr>
<td>5</td>
<td>2000</td>
<td>0.0859</td>
<td>0.1033</td>
</tr>
<tr>
<td>6</td>
<td>3000</td>
<td>0.0577</td>
<td>0.0943</td>
</tr>
</tbody>
</table>

Table 6.2: Average Errors in Simulation for the Race Track
CHAPTER 6. EXPERIMENT

whereas it is good in simulation. It is impossible to drive the car at high speeds with only full steering angles.

- The speed is not constant, which results in the learned best action possibly being suboptimal.

Despite these problems, it is still possible to apply a reinforcement learning controller algorithm to practical driving. This is done by using the learned results from simulation. Now instead of using Tab 6.1 for simulation, Tab 5.1 must be used to provide the translation from simulation steering angle to actual setting.

It is possible to continue the learning in practice and modify the values learned from simulation, as long as there are enough trials and the further learning exploration rate is high enough. However, due to the limit in our experiment field and other factors, this has not been done.

The result from practical driving is shown in Fig 6.26. As can be seen from this figure, the tracking is good even at the speed of about 0.4m/s. The average error in practice is shown in Tab 6.3. Each row in the table was gathered and averaged from a single lap (or one trial) instead of a number trials. Due to more sources of error in practical driving (i.e. interference, error from vision server), the average errors can not be as stable as that of a simulation. As can be seen from this table, they change greatly in different trials. The average orientation error is almost twice as large as that in simulation, while the average position error is almost the same. There may be two possible reasons for this. Firstly, the angle measurement from the vision server is not as accurate as the position measurement. The position is the average of two points, the orientation is not. If there is an error in the coordinate of just one point, there may not be much error in the position of the car, but it may lead to a big error in the orientation. Secondly, it may
Table 6.3: Average Error in Practical Driving for the Race Track
Data in each row is from the average error in a trial.

<table>
<thead>
<tr>
<th>reward</th>
<th>avg $\hat{y}$</th>
<th>avg $\theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3.3000</td>
<td>0.0839</td>
<td>0.4290</td>
</tr>
<tr>
<td>-3.2950</td>
<td>0.0953</td>
<td>0.4020</td>
</tr>
<tr>
<td>-4.2687</td>
<td>0.1196</td>
<td>0.3649</td>
</tr>
<tr>
<td>-0.9768</td>
<td>0.0728</td>
<td>0.2021</td>
</tr>
<tr>
<td>-1.2079</td>
<td>0.0482</td>
<td>0.2779</td>
</tr>
<tr>
<td>-1.1816</td>
<td>0.0527</td>
<td>0.2865</td>
</tr>
</tbody>
</table>

mean that the same weight for the orientation error for practical driving and for simulation is non-optimal.

The performance in practical driving for different controllers is shown in Tab 6.4. Only the reinforcement learning controller has been implemented for both the big car and the small car, since it does not require any fine turning and is flexible enough to adapt to different cars. The big car is much easier to control. Its steering angle is almost linear. Its speed is more stable for different segments and easier to control. For the small car, even with the same setting its speed may change when tracking different types of segment (i.e. tracking an arc is slower than tracking a line). Due to the nonlinearity of the steering angle of the small car, the fuzzy logic controller does very poorly and cannot follow the race track successfully.

The reinforcement learning controller has the same performance as the fuzzy logic controller for the big car, they are about twice as good as the modified sliding mode controller, or roughly three times better than the sliding mode controller. But the reinforcement learning controller is worse than the look-ahead controller.
Figure 6.25: The Learned Result can be Applied to Different Paths in Simulation

This path can also be tracked in practical driving.
Figure 6.26: Practical Driving Using the Learned Cases from Simulation
Table 6.4: Comparison of different Controllers in Practical Driving

<table>
<thead>
<tr>
<th>Controller</th>
<th>Big Car (C=900cm)</th>
<th>Small Car (C=720cm)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Speed(cm/s)</td>
</tr>
<tr>
<td>Sliding Mode</td>
<td>72</td>
<td>12.5</td>
</tr>
<tr>
<td>Modified Sliding Mode</td>
<td>46</td>
<td>19.5</td>
</tr>
<tr>
<td>Fuzzy Logic</td>
<td>27</td>
<td>33.3</td>
</tr>
<tr>
<td>Look-Ahead</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Reinforcement Learning</td>
<td>28</td>
<td>32.1</td>
</tr>
</tbody>
</table>

Time shows how many seconds to complete a lap, whose circumstance is 900cm in the case for a big car, and 720cm for a small car.

6.6 Experiment with Different Weights in the Reward

This section investigates the influence of different weights in the reward function on the performance of the reinforcement learning controller in simulation. These experiment are all done on a straight line, as it is a most challenging problem. As mentioned previously, the reward function is of great importance to the learning behavior. In our path tracking problem, this in turn is the problem of assigning weights to the position error($w_1$), orientation error($w_2$) and the change between two consecutive actions($w_3$).

High $w_1$ will force the car to go to the path quickly. For example, if $w_1 = 1$ and the other two are zero, the outcome will be that the car goes straight to the path. This may sometimes result in the car following the path in the opposite direction as no orientation information is given in the reward function. This scenario is shown in Fig 6.27. As shown in this figure, the car approaches the path quickly even if it has a large initial position and orientation errors. On the other hand, higher $w_2$ will force the car to just go parallel to the path and it will never approach the path. Fig 6.28 is one
example for this with \( w_2 = 1 \) and the other weights set to zero. The car will just learn to go parallel to the line. However, if \( w_1 \) is not zero, the car will get close to the line, but some distance remains, depending on whether reducing this distance error can result in a better reward than the current orientation error. The equivalence of \( \tilde{y}, \tilde{\theta} \) and \( \Delta U \) is based on the following estimation. From the reward function Eq 5.3, assume that the three portions are equal, then we have:

\[
\sqrt{w_1} \times \frac{\tilde{y}}{8} = \sqrt{w_2} \times \frac{\tilde{\theta}}{2 \times \pi} = \sqrt{w_3} \times \frac{\Delta U}{9}
\]

For example, if \( w_1 = 1, w_2 = 1 \) and \( w_3 = 0.0025 \) then we have

\[
\frac{\tilde{y}}{\tilde{\theta}} = \frac{2}{\pi} = \frac{1}{90^\circ} \quad \frac{\tilde{y}}{\Delta U} = 0.022
\]

which means that every 1 meter of position error is equivalent to 90° of orientation error, or 10cm to 9°. And every 2.2cm of position error is the equivalent of 1 step change between two consecutive actions. This implies that the car will not try to get close to the path if the position error is less than 10cm and if there is no orientation error. The car won’t change the current steering angle if the position error is less than 2.2cm. This is relatively reasonable. So most experiments shown previously were conducted with these default weights in the reward function.

It is interesting to look at the grid chart for different weights in the reward function. For example, What influence does a change in \( \Delta U \) have? Fig 6.29 and Fig 6.30 are the grid output for \( w_1 = w_2 = 1 \), but \( w_3 = 0.025 \) for the former, and \( w_3 = 0.25 \) for the latter. Actions in Fig 6.30 change less often than that in Fig 6.29, and actions change most often in Fig 6.5 with \( w_3 = 0.0025 \). Because \( \sqrt{0.025} \times \frac{1}{9} = 0.07 \), and \( \sqrt{0.25} \times \frac{1}{9} = 0.22 \), so along
Figure 6.27: Tracking the Line with only $\hat{y}$ in $r$

The car tracks the path in the opposite direction due to no $\hat{\theta}$ in the reward function.
Figure 6.28: Tracking the Line with only $\dot{\theta}$ in $r$

The car goes parallel to the path and will not get close to it.
the $\tilde{y}$-axis (with constant $\tilde{\theta}$), each action may last 7cm in Fig 6.29, and 22cm in Fig 6.30 for the point (0,0). At this point the actions should change most frequently. This is approximately true in these figures.

In Fig 6.30, due to the heavy penalty for changing actions, the current steering angle will not be changed unless the position error (or orientation error) increases drastically. This behavior is shown in Fig 6.32. Notice that in the worst case, the car will not try to get close to the line even if it is 22cm away from the path and there is no orientation error.

Fig 6.31 is the track for $w_3 = 0.025$. The car oscillates along the path. The current steering angle will not be changed until the car passes the line and keeps going forward for some distance to have a large enough position error or orientation error before the current action changes. This is very similar to the overshooting problem as shown in Fig 6.6.

Because these problems, it is necessary to introduce dynamic weights for the reward function. When the car is far away from the path, a large $w_1$ is desired so the car can go close to the path quickly, the other two factors are not so important. When the car is near the path, higher $w_2$ and $w_3$ are better so the car can so can keep the current orientation and action relatively stable.

6.7 Shooting Yourself in the Foot

During our experiments in simulation, a strange phenomenon was encountered. When tracking a left arc, the car could not approach the arc correctly even after a large number of trials. As shown in Fig 6.33, the car would track slightly outside of the path.

A closer inspection showed the reason for this anomaly. The difference
Figure 6.29: Grid Chart for $w_1 = 1, w_2 = 1, w_3 = 0.025$
Figure 6.30: Grid Chart for $w_1 = 1, w_2 = 1, w_3 = 0.25$
Figure 6.31: Tracking the Line with $w_1 = 1, w_2 = 1, w_3 = 0.025$

The car will overshoot the path before $\hat{y}$ or $\hat{\theta}$ are large enough to change the current action.
Figure 6.32: Tracking the Line with $w_1 = 1, w_2 = 1, w_3 = 0.25$
The heavy penalty for action changes results in stiff steering.
Figure 6.33: Anomaly when Learning to Tracking a Left Arc
When a case close to the path but with the wrong action is created, it is difficult to visit this state frequently enough to learn the correct value.
in learned database between this experiment and others is that this scenario created a special case for the state \((0,0)\) and that the learned action for this case was wrong (slight-left instead of sharp-left). The probability for this happening is quite low. This is because when the car tries to approach the path from somewhere away, a state near \((0,0)\) is explored first, a case is created for this state and a case for state \((0,0)\) will never be created, due to the case density parameter.

When such a state is created, it is very difficult for the car to get onto the path. This is because it is extremely difficult to visit this state frequently enough, since the current suggested slight-left turn action, even when combined with other suggested actions, moves the car away from the path. This problem is similar to the exploration and exploitation problem. However, it can not simply be solved by modifying the exploration parameter. The problem is that from the current position, the car will have to make a number of moves (for example, \(n\)) to get to the path. Assuming that the probability (rate) for exploration is \(e\), then the probability of reaching the path decreases exponentially with the value of \(e^n\). If the exploration rate is low, outer states will be visited first and locally optimal actions are learned and taken, which will lead away from the problem state. If the exploration rate is high, it is also difficult to get onto the path (so \(\hat{y} = 0\) and \(\hat{\theta} = 0\)). It is difficult to find the ultimate solution for such a problem.

As can be seen from the figure, the circle is tracked by a number of segments. This steering angle is stiff. This implies that the weight on the action change is slightly too high. The learned best action is a slight-left turn. This action has a turning radius of 65cm, which is too big for the circle. The current action will not be changed until the position or orientation error is large enough.
CHAPTER 6. EXPERIMENT

To verify the above argument, an experiment was conducted. The case for state \( (0,0) \) was deleted. The car could approach the path very well.

Another experiment was conducted in order to further investigate this problem. Every trial starts with the car on the path. To eliminate the influence of the weight on the action change, \( w_3 \) is set to zero. After about 1000 trials, the car can track the circle very well and it can always keep moving on the path.

6.8 Summary

This chapter presents a lot of data obtained from experiments, both from simulation and practical driving. The learned grid chart is the most important tool to analyze the learning behavior and the quality of the learned control function.

Although there are non-optimal actions or even wrong actions in some small regions of the state space, the overall actions are good enough to track any given path consisting of the three basic segments. It is also possible to apply the learned control function from simulation to practical driving, by seeding the controller through simulation first.

Our experiment shows that the left arc and the right arc are actually symmetric, so there is no need to learn both from scratch.

Learning any given path can be done by learning the path as a whole segment (full-path learning) or by learning the basic segments first (segment-first learning). Once all three basic segments are learned, the learned result can be applied to track any given path.

The weights on the reward function are an important factor for the learning behavior. Incorrect weights may lead to failure of path tracking. We
suggest dynamic weights on the reward function.
Chapter 7

CONCLUSION

7.1 Conclusions

This thesis describes the design and implementation of a reinforcement learning controller for the path tracking problem of car-like robot. It solves the problem of continuous states through the use of a case-based function approximator. States are generated dynamically. Such a controller does very well in simulation. But most importantly, the learned control function can be applied to practical driving. The performance in practical driving is good and much better than that of the sliding-mode controller and comparable to that of the fuzzy logic controller on a big car. The robustness of this controller is another advantage.

Although it is worse than the look-ahead controller, however, this reinforcement learning controller can minimize the control error. This is very important in order to apply it to some tasks like robosoccer, where sometimes less position error is better than faster speed, for example, missing the ball in a goal shot.

Since left arcs and right arcs are symmetric, the agent just needs to learn
CHAPTER 7. CONCLUSION

two types of segments. Since in most path tracking tasks, a path consists of the basic three segments, it is possible to learn the control function for these basic types first and then store the learned result. When a controller is needed, these stored learned result can be applied, lest to learn from scratch. This makes the reinforcement learning controller of practical use.

The current implementation is rudimentary. The performance in practical driving is not as good as it could be. It is possible to improve further its performance through future work, which is listed below.

7.2 Future Work

There are a number of possible directions for future research in this area. These include:

*Dynamic weights to the reward function.* The reward function is very important to the performance of the controller. Currently the weights for the three portions in the reward function are static. This is not the optimal approach. It is desirable to have more weight on the position error when the car is far away from the path, or more weight on the orientation error if the car is near the path. This thesis conducted experiments to identify the need for dynamic weights.

*Other reward functions.* Other alternatives for the reward function may be even better than the current one, which is the sum of the three portions. This is the simplest reward. What will be the effect if these three portions are multiplied? Is it necessary to include any other information in the reward? Is it better to use the difference of the orientation of the virtual car and the real car in the reward, instead of the orientation error? The answers are needed for these questions.
CHAPTER 7. CONCLUSION

*Next segment information.* Lack of such information leads to overshooting the current segment. When it is about to switch to the next segment, the action will be different from those action for the current segment.

*Learn to control the speed of the car.* There are two reasons why it is important to learn to control the speed of the car. Firstly, learn the speed setting to keep the car going at a constant speed. It is necessary. The speed setting must be changed if the battery level changes. Going at a constant speed can alleviate overshooting in practical driving. Secondly, learn to complete the path tracking task with the lest time. For example, to go faster (at higher constant speed) when tracking a straight line will reduce the time while maintaining controllability.
Bibliography


BIBLIOGRAPHY


