POLITECNICO DI MILANO

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Elettronica, Informazione e Bioingegneria

ANALYSIS AND OPTIMIZATION OF AN EXPERIMENTAL APPARATUS TO TEST ACTIVE SAFETY SYSTEMS IN VEHICLES

Relatore: Prof. Alessandro COLOMBO

Tesi di Laurea di:
Andrea RIZZI, matricola 766468

Anno Accademico 2012-2013

Ai miei genitori

Ringraziamenti

Innanzitutto un grazie sentitissimo al Professor Alessandro Colombo e alla Professoressa Domitilla Del Vecchio, non solo per i preziosi suggerimenti ma anche per avermi regalato la fiducia e l'occasione di fare quest'esperienza ed arricchirmi così tanto dal punto di vista professionale e umano. Un abbraccio infinito va a Giulia che nel sopportare e supportare senza sosta il mio girovagare ha dimostrato di avere la pazienza di una santa. Alla fine di tutto farò in modo che ne sarà valsa la pena. Un'altro grazie immenso ai miei genitori, sempre pronti a rendermi la vita più facile e rasserenarmi a discapito di tutto. A tutti i miei parenti, mio fratello, nonni, zii e cugini che sono sicuro non hanno dubitato di me neanche per un secondo. A Barbalenzio, Casto e Daniele che grazie a Dio sembrano essere stati creati per non farmi studiare. Ai compagni di avventura svedesi ormai sparpagliati per l'Europa: Fabio, Ale, Gio, Loris, David. Alla mia squadra, per le battaglie in mezzo al campo e le risate in pizzeria negli ultimi 13 anni.

A tutti voi grazie. Non varrei la metà di quello che sono senza di voi.

Abstract

This work describes various solutions adopted for the creation of a laboratory environment suitable for experimental testing of algorithms for vehicle collision avoidance. Moreover, the results obtained from the test of an algorithm for robust multi-agent collision avoidance at intersections are reported. In the first place, a model of the vehicle available in the lab is developed. A through analysis of the disturbances affecting the modeled system is performed to identify and quantify the sources of noise. The environmental factors and the eventual errors in the parameters calibration are included in the model as additive disturbance terms, and a predictor based on this model that assumes constant motor input and constant disturbances is implemented. Secondly, various techniques are adopted to reduce the previously identified disturbances and to allow a less conservative prediction. Finally, the aforementioned algorithm is successfully tested in the laboratory with two computer-driven vehicles.

Sommario

Questo elaborato presenta il lavoro di analisi ed ottimizzazione di un apparato sperimentale per test di sicurezza attiva su veicoli. In particolare viene testato un algoritmo per la prevenzione di incidenti agli incroci basato su *scheduling* e caratterizzato da robustezza ai disturbi.

L'algoritmo preso in considerazione richiede di effettuare una predizione dello stato futuro dei veicoli. A questo proposito viene sviluppato un modello delle macchine disponibili in laboratorio che fornisce la base per la realizzazione del predittore (Capitolo 2). I fattori di disturbo agenti sul sistema vengono identificati ed incorporati nel modello come termini di disturbo additivi.

Diverse tecniche sono adottate con il fine di ridurre l'effetto di questi disturbi e rendere la predizione meno conservativa e compatibile con lo spazio limitato disponibile all'interno del laboratorio per l'esecuzione dell'esperimento (Capitolo 3). Il controllore dello sterzo è riprogettato con l'obiettivo di minimizzare l'impatto della dinamica laterale su quella longitudinale. Gli effetti dovuti alla capacità che filtra l'alimentazione e ai disturbi dipendenti dal percorso (inclinazione del *testbed* e angolo di curvatura) vengono ridotti da un compensatore applicato al segnale d'ingresso del motore elettrico. Infine, gli errori di misura su posizione e direzione dei veicoli sono limitati da opportuni filtri.

I risultati sperimentali derivanti dalla minimizzazione dei disturbi e il test dell'algoritmo sono illustrati nel Capitolo 4. L'esperimento con due veicoli è portato a termine con successo.

Si descrive inoltre l'architettura e il funzionamento di un simulatore realizzato durante lo svolgimento del progetto che virtualizza l'ambiente di laboratorio e che può essere utilizzato per l'identificazione dei disturbi e per velocizzare l'implementazione, il debug e la verifica delle soluzioni adottate in laboratorio (Capitolo 5).

Si raccolgono infine nel Capitolo 6 le informazioni che descrivono l'utilizzo pratico dei vari script implementati per la semplificazione di determinate procedure in laboratorio.

Contents

1 Introduction **1**

2 Predictor **3**

2.1 Laboratory environment . . . 3
2.2 Basic model . . . 4
2.3 Prediction . . . 6
2.4 Disturbance boundaries calibration . . . 6

3 Disturbances analysis and reduction **9**

3.1 Lateral and longitudinal dynamics coupling . . . 9
3.1.1 Disturbance estimation . . . 10
3.1.2 Steering model . . . 11
3.1.3 Steering controller design . . . 13
3.1.4 Controller calibration . . . 16
3.2 CPS measurement error . . . 16
3.2.1 Position linear correction . . . 16
3.2.2 Heading filtering . . . 17
3.3 Other disturbances . . . 19
3.3.1 Effect of steering and testbed slope . . . 19
3.3.2 Effect of the power filter capacitor . . . 19
3.3.3 Mathematical formulation of the engine compensator . 22
3.4 On-line estimation of the motor gain . . . 23

4 Experimental results **25**

4.1 Disturbances minimization . . . 25
4.2 Algorithm testing . . . 28

5 Simulator **29**

5.1 Architecture and features . . . 29
5.2 User guide . . . 30

6 Laboratory manual **33**

6.1 Software overview . . . 33
6.1.1 Code organization . . . 33

6.1.2 Summary of changes 33
6.2 Software configuration 35
6.3 Car software 36
6.4 Scripts 37
6.4.1 Debug information visualization 37
6.4.2 Model parameters identification 39
6.4.3 Path-dependent disturbance identification 41
6.4.4 Tools for quantitive estimation of residual disturbances 44
6.5 CPS linear correction configuration 45
6.6 Path specification syntax 46
6.6.1 Steering compensator directives 46
6.6.2 Path-dependent disturbance specification 46
6.7 Troubleshooting 47

7 Conclusions **49**

Bibliography **51**

List of Figures

2.1 Lab vehicle, paths and cameras 4
2.2 Block diagram representation of the longitudinal model. . . . 5
2.3 Example of bad disturbance boundaries 7

3.1 Causes of non-zero d_{proj}. 9
3.2 Coordinate system and example of trajectory 10
3.3 Lateral distance x of the car in time on path fig8CL with the old steering controller. 10
3.4 Steering model of the car. 11
3.5 Block diagram representation of the lateral model. 13
3.6 Error of the position measured by the CPS 17
3.7 Car speed response to PWM and steering step input 20
3.8 Effect of capacitor on the car speed dynamics 21
3.9 Path-dependent disturbance dependencies 22
3.10 Speed dynamics variation with battery charge level. 23

4.1 Heading filtering 26
4.2 Distance from path 26
4.3 Final disturbances on the car speed. 27
4.4 Compensator and path-dependent disturbances with and without engine compensator 27
4.5 Trajectory of the two cars 28

6.1 Example of how to choose the four points for CPS linear correction configuration. 44

List of Tables

3.1 Steering model identification 12
3.2 Steering model validation for car 2. 12
3.3 Steering model validation for car 3. 13

4.1 Summary of disturbances norms 25

Chapter 1

Introduction

The project aims to create a laboratory environment suitable for experimental testing of algorithms for vehicle collision avoidance at intersections. In particular, a robust multi-agent collision avoidance algorithm based on scheduling is implemented and tested [1]. This algorithm extends a technique presented in a previous work [2] and puts the system in an environment which interacts with vehicles and influences their dynamics. As a consequence, the method requires the prediction of the future state of human-driven vehicles in an environment affected by disturbances. The list of subgoals thus includes modeling of the system, analysis of noise sources, and disturbances minimization.

The thesis is organized as follows. Chapter 2 introduces the laboratory environment, defines the equations that model the system and shows the methods used for the prediction of vehicles state. Chapter 3 describes the analysis, identification and reduction of the system disturbances. Experimental results obtained from algorithm testing and disturbances minimization are shown in Chapter 4. During the whole process, a simulator of the laboratory environment has been used to speed up the identification of the problems and the development of solutions. The design and basic usage of such a simulator is described in Chapter 5. Finally, Chapter 6 presents a practical guide to the newly implemented features of the lab for future reference.

Chapter 2

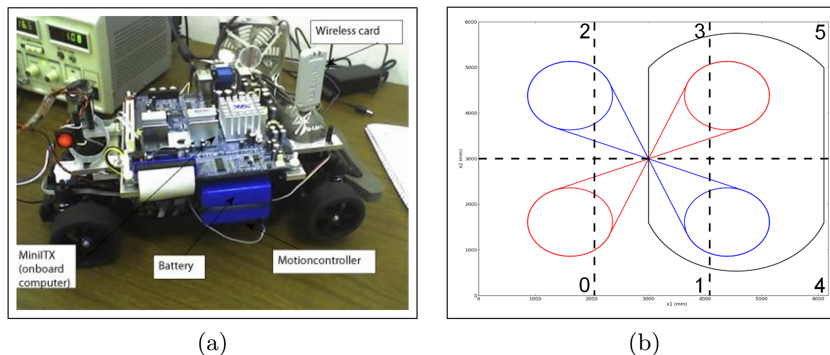
Predictor

The collision avoidance algorithm which is tested requires the estimation of the future state of the car and the time at which the vehicle enters and exits the intersection. To this end, I present a model-based predictor which is described in this chapter. In the first place, a description of the laboratory environment is given. Secondly, the car model is presented together with an overview of the disturbances caused by the environment and the model imperfections. Finally, the methods adopted for the prediction are explained.

2.1 Laboratory environment

The testbed consists of a $40m^2$ surface where the vehicles are free to move. This surface is covered by 6 cameras mounted on the ceiling which send the images to three computers for further processing. Positions and directions of the cars are extracted from the received frames with the help of computer vision and specific symbols mounted on the top of the vehicles. The whole system composed by cameras and computers will be referred in the following as Camera Positioning System (CPS). The information is then made available to the vehicles used in the experiment through a UDP/IP connection. Each camera has a number from 0 to 5 and it is responsible for a portion of the testbed (see Figure 2.1b).

The base of the vehicles available in the lab is a Tamiya scaled RC car chassis equipped with a DC motor. A microcontroller with 2 PWM channels is used to control the motor and the steering. A motherboard and a hard disk are installed on the top of the car and run Ubuntu or Fedora [4]. A C program is executed on the on-board computer which takes care of opening a connection and exchanging information with the CPS, and sends the desired input signal to the microcontroller. The speed of the car is measured by an encoder mounted on the rear axis. There are 6 vehicles in the laboratory, each of them numbered from 1 to 6.



(a) (b)

Figure 2.1: **(a)** The vehicle used in the laboratory and **(b)** the paths followed by each car. The path in blue is called "fig8CL", the red one is "fig8AL" and the black one is "fig0". There are six cameras in the lab that are numbered from 0 to 5. Each camera is responsible for the area labeled with the correspondent number in **(b)**.

The path followed by each car is specified in a file contained in the vehicle hard disk as a sequential list of points. Three paths are used for the experiment which are called "fig8AL", "fig8CL" and "fig0" (see Figure 2.1b).

The CPS and the vehicles involved in the experiment communicate through UDP/IP protocol. The CPS not only has to send to cars the information extracted from camera frames (e.g. position, direction, etc.), but it is also responsible of relying the information that is only available locally on the vehicles on-board computers. This includes the car speed which is measured by the encoder, the motor gain parameter which is estimated online (see Section 3.4), and the driver desired motor input.

The two flows of information (from CPS to car and from car to CPS) are asynchronous. The CPS keeps in memory the most recent states sent by the vehicles but if a long delay occurs in one of the car, the information sent to the others by the CPS becomes quickly out-of-date and the algorithm can encounter problems.

2.2 Basic model

The model of the vehicle is based on the decoupling of the lateral and longitudinal dynamics of the car with respect to the path. In other words, the model assumes the vehicle to follow exactly its path and it treats the effects of the lateral dynamics as a disturbance. Under this hypothesis, the state of the car can be described by its longitudinal components only (i.e. its unidimensional position and speed along the path). The car is represented

by the the system of equations

$$\begin{cases} \dot{y}(t) = v(t) + d_{proj}(t) \\ \dot{v}(t) = av(t) + b + fu(t) + d_{steer}(t) + d_{slope}(t) + d_{cap}(t) \\ y_m(t) = y(t) + d_{meas,y}(t) \\ v_m(t) = v(t) + d_{meas,v}(t) \end{cases} \tag{2.1}$$

where $y(t)$ is the unidimensional position along the path, $y_m(t)$ is its measurement, $v(t)$ is the car speed (the one measured by the encoder), and $u(t)$ is the PWM signal given as input to the DC motor. The parameter b accounts for friction, f is the gain of the motor while the $d_X(t)$ terms represent various sources of additive disturbance. In particular they account for

- d_{proj}: the fact that the vehicle does not follow perfectly its path (see Section 3.1). In other words, it describes the weaving of the car around its prescribed path.
- $d_{meas,y}$ and $d_{meas,v}$: the measurement error of the CPS and the encoder (see Section 3.2).
- d_{steer}: the effect of the steering on the speed (see Section 3.3.1).
- d_{slope}: the fact that the testbed is not perfectly flat (see Section 3.3.1).
- d_{cap}: the disturbance caused by the capacitor that filters the power (see Section 3.3.2).

Figure 2.2 gives a block diagram representation of the model. The magnitude of the disturbance terms was too high to allow an useful prediction with this model, thus a reduction of these disturbances was necessary. A more detailed explanation of these terms and the solutions adopted for their minimization are illustrated in Chapter 3.

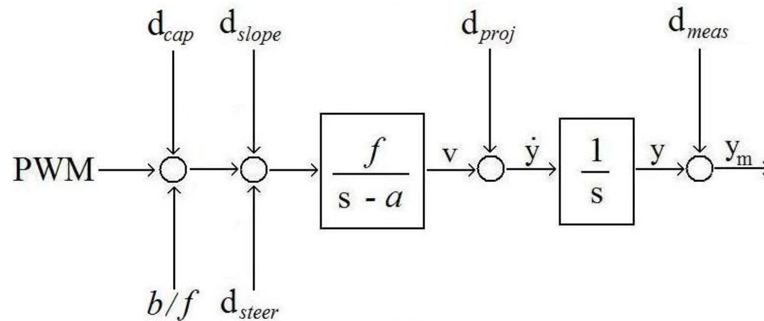


Figure 2.2: Block diagram representation of the longitudinal model.

2.3 Prediction

The algorithm requires two types of prediction: the car state in the next clock cycle, and the time at which the vehicle reaches a certain point of the path, in particular when the car enters and exits the intersection.

The first type of prediction is easily done by integrating the system of equations (2.1). Defining $d_y = d_{proj}$ and $d_v = d_{cap} + d_{slope} + d_{steer}$, given the current measured state $(y_m(k), v_m(k))$, the solution can be computed analytically

$$\begin{cases} y(k+1) = y_m(k) + \frac{av_m(k)+b+f\cdot u(k)+d_v}{a^2}(e^{a\tau}-1) - \frac{b+f\cdot u(k)+d_v}{a}\tau + d_y\tau \\ v(k+1) = v_m(k)e^{a\tau} + \frac{b+f\cdot u(k)+d_v}{a}(e^{a\tau}-1) \end{cases} \tag{2.2}$$

where $u(k)$ is the motor input, τ is the clock cycle time. Note that d_v and d_y are assumed to be constant terms, but their value is unknown at the time of prediction. To solve this problem, boundaries for disturbances $[d_{min}, d_{max}]$ are determined (see Section 2.4) and the prediction is performed in the worst and best case scenario obtaining an interval of states $[y_{min}, y_{max}]$ and $[v_{min}, v_{max}]$ rather than a single one.

The second problem can be formulated as finding the time $\bar{t}$ such that $\bar{y} = y(\bar{t})$, where $\bar{y}$ is monodimensional coordinate of the position that must be reached by the vehicle on its path. Let the motor input and the disturbances be constant. If $t_0 = 0$, then according to the model $\bar{t}$ is the solution to the equation

$$y(t) = y_m(0) + \frac{av_m(0) + b + f\bar{u} + d_v}{a^2}(e^{at} - 1) - \frac{b + f\bar{u} + d_v}{a}t + d_y t \tag{2.3}$$

which is mixed exponential in t and must be solved numerically. The Newton-Raphson method is used to this end. It should be noted that for this type of prediction not only we do not know the dynamics of disturbances $d_v(t)$ and $d_y(t)$, but also $u(t)$ in the time interval $[0, \bar{t}]$ is unknown. Indeed, the driver can potentially change motor input signal at each clock cycle. To simplify, the algorithm tackles the problem by assuming both input and disturbances to be bounded $u(t) \in [u_{min}, u_{max}], d(t) \in [d_{min}, d_{max}]$. These boundaries are then used together with Equation (2.3) to obtain an interval of time $[\bar{t}_{min}, \bar{t}_{max}]$ in which the vehicle may occupy the intersection [1].

2.4 Disturbance boundaries calibration

The upper and lower bounds for the disturbances as they are defined above are very high. $d_y(t)$ can reach $250 - 300\frac{mm}{s}$ while $d_u(t)$ can get up to $350\frac{mm}{s^2}$. These boundaries makes the prediction too conservative for the experiment to work on the testbed. Indeed, as soon as a car exits the intersection and makes a brand new prediction, the algorithm detects an inevitable collision,

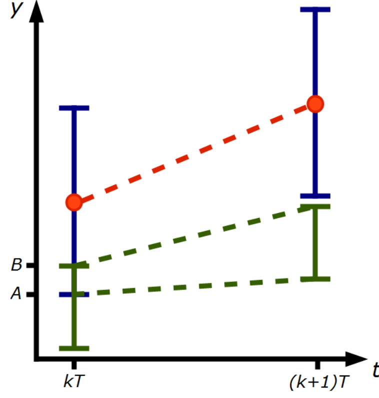


Figure 2.3: The red dot is the actual car position. The blue interval represents the robust interval of measured positions while the green one is the interval of positions predicted in the previous step. Boundaries that are not correctly defined may cause the intersection between the predicted and the measured interval to be empty.

consequently the safe input remains undefined and the code makes the cars stop. The confined space available for the experiment makes it difficult to solve the problem by changing the paths drastically. However, what we truly care is to be able to predict the time at which a car enters and exits the intersection. This fact can be exploited to determine less conservative boundaries.

Let t_f be the time at which the car enters (or exits) the intersection. We define $d_{y,min}$ and $d_{y,max}$ as the disturbances bounds such that

$$\int_{t_0}^{t_f} v(t) + d_{y,min} dt \leq \\ \leq y(t_f) = \int_{t_0}^{t_f} \dot{y}(t)dt = \int_{t_0}^{t_f} v(t) + d_y(t)dt \leq \\ \leq \int_{t_0}^{t_f} v(t) + d_{y,max} dt \quad (2.4)$$

for every t_0, state $x_0 = (y_0, v_0)$, input $u(t)$ and disturbance $d_y(t)$. One can define $d_{v,min}$ and $d_{v,max}$ similarly. It should be noted that defining the boundaries in this way allows the state to get temporarily outside its predicted boundaries in the interval $(0, t_f)$. In other words, we do not care what happens between t_0 and t_f as long as we know that the prediction will be correct at the intersection.

Conversely, the measurement disturbance boundaries are not affected by the same problem and they are defined as the actual minimum and maximum measurement error made by the CPS and the encoder for $y(t)$ and $v(t)$ respectively. However, with $d_y(t)$ and $d_v(t)$ boundaries defined as in Equation (2.4), the one-step prediction solved by the system (2.2) may be wrong

because the boundaries the predictor uses are not the actual upper and lower bounds of the system. As an example, consider Figure 2.3. The algorithm determines the interval of possible positions of the vehicle $[y_{min}, y_{max}]$ by comparing the prediction performed at the previous step (green interval) with the uncertain measurement given by the CPS (blue interval). The state of the car is assumed to be in the set obtained as the intersection between these two estimates (in the figure represented by $[A, B]$ at time kT). However, since the disturbances used for the prediction underestimate the actual maximum boundary, the actual vehicle position (the red circle) can change faster than expected by the algorithm. If the car position is too far from the predicted one, the intersection between the two intervals becomes empty. This is the case in Figure 2.3 at time $(k+1)T$. Assuming the distance from the predicted state and the actual state to be bounded, one can circumvent this problem by increasing the measurement uncertainty boundaries beyond the actual ones (i.e. by increasing the size of the blue interval).

After acknowledging the complexity of estimating quantitatively the boundaries by analyzing algorithmically the experimental data without overestimate them, I adopted a trial and error approach to determine the bounds with the help of the simulator.

Chapter 3

Disturbances analysis and reduction

The magnitude of the disturbances sketched in Section 2.2 is too high to allow an useful prediction. This chapter explains in details the sources of disturbance and the techniques adopted to reduce their effect. The results of this work are presented in Chapter 4.

3.1 Lateral and longitudinal dynamics coupling

The car steering controller is not refined enough for the car to follow the path perfectly. The main implication is $\dot{y}(t) \neq v(t)$. Indeed, in the model $\dot{y}(t) = v(t) + d_{proj}(t)$. In order to fully understand what gives rise to a non-zero $d_{proj}(t)$, one should consider that the path followed by cars is composed by a polygonal chain. For example, in Figure 3.1 the car goes through the trajectory A-B-C-D. While in sections A-B and C-D, the value of $\dot{y}$ is simply the projection of $v(t)$ on the path, in section B-C the vehicle position

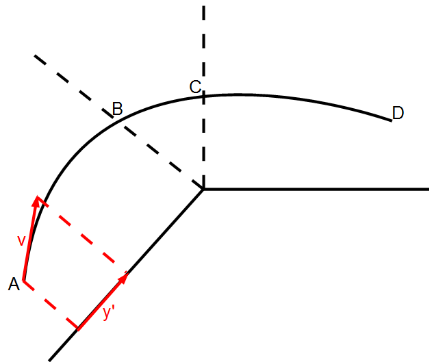
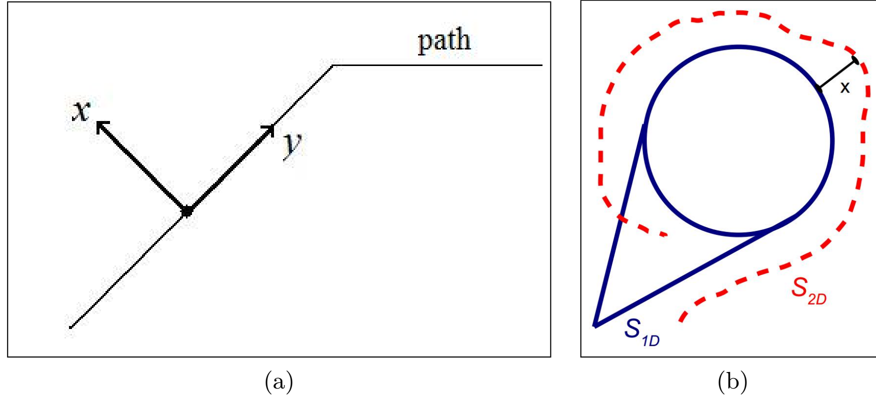


Figure 3.1: Causes of non-zero d_{proj}.



(a) (b)

Figure 3.2: **(a)** coordinate system and **(b)** example of trajectory on path fig8CL.

is projected on the path always in the same point P so $\dot{y} = 0$. Similar considerations lead to conclude that the unidimensional speed $\dot{y}$ reaches infinity when a car cuts a corner.

3.1.1 Disturbance estimation

In order to quantitatively estimate d_{proj}, let us consider the path in Figure 3.2b which is half of fig8CL and has a total length of $S_{1D} = 9465.53mm$. In the following the coordinate system will be relative to the path as represented in Figure 3.2a. In other words, $y(t)$ represents the position along the path while $x(t)$ the lateral distance from the path. If the car actually goes through S_{2D} millimeters, the average unidimensional speed from the beginning to the end of the path is given by

$$\bar{\dot{y}} = \bar{v} + \left(\frac{S_{1D}}{S_{2D}} - 1\right)\bar{v} = \bar{v} + \bar{d}_{proj}.$$

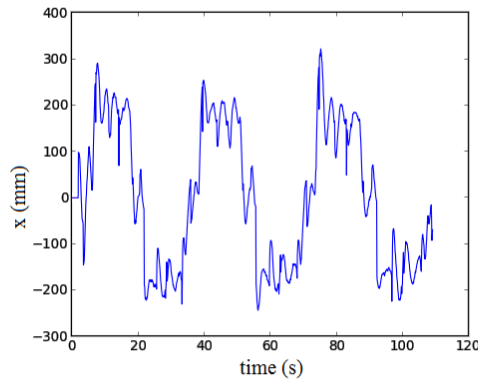


Figure 3.3: Lateral distance x of the car in time on path fig8CL with the old steering controller.

The first term of the sum is the actual average speed of the car as measured by the encoder, the second one can be considered as a disturbance term due to the projection on the path of the actual car speed. Assuming the car to maintain a lateral distance of $x = 20cm$, it is easy to compute a value for S_{2D} and obtain

$$\bar{d}_{proj} = 0.25\bar{v}.$$

In other words the disturbance on the unidimensional speed would be 25% of the actual car speed. It is clear that the lateral dynamics considerably affects the longitudinal components of the car state. As a matter of fact, the old steering controller keeps the car at an approximately constant lateral distance of about $20cm$ during the experiment (see Figure 3.3). Thus it becomes necessary to design a new and more effective controller.

3.1.2 Steering model

Let us consider the simplified model of the vehicle in Figure 3.4, where w is the wheelbase and δ is the steering angle. The dynamics is actually affected by some slip angles which is assumed to be negligible for the purpose of this model. We want to determine the curvature radius of the center of the car (i.e. R) because that is the position tracked by the CPS. We assume also that the steering angle $\delta(t)$ can change instantaneously and is proportional to the steering input or, in other words, that $\delta(t) = \alpha u(t)$, where α is refered to as steering factor and $u(t) \in [-100, 100]$ is the steering input. With some simple trigonometry one can obtain

$$R_r = \frac{w}{tan(\alpha u)}$$

which means that

$$R = \sqrt{R_r^2 + \frac{w^2}{4}} = \sqrt{\frac{w^2}{tan(\alpha u)} + \frac{w^2}{4}}. \tag{3.1}$$

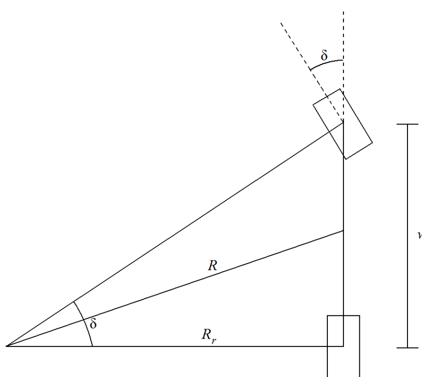


Figure 3.4: Steering model of the car.

Car 1				
Input	Diameter right	Diameter left	Error right	Error left
100	1360mm	1380mm	4.02mm	-5.98mm
90	1450mm	1420mm	39.18mm	54.18mm
80	1710mm	1680mm	8.90mm	23.90mm
70	1940mm	1880mm	21.56mm	51.56mm
60	2400mm	2290mm	-38.88mm	16.12mm
50	2980mm	2960mm	-92.27mm	-82.27mm

Table 3.1: Results of the experiments with car 1 for steering factor identification. All measures are reported in millimeters. Measured diameters for right and left turns is separated. The error in the table is the difference between the radius (not the diameter) predicted by the model and the real radius.

The wheelbase can be directly measured on car. The only unknown parameter of the model is the steering factor α.

In order to determine α, one can run car 1 on circles with constant steering input and manually measure the diameter of the circle it goes through. Results of this experiment are reported in the Table 3.1. The steering factor is then easily computed by minimizing the mean error. This leads to the value $\alpha = 0.2116466582$.

The model is validated with cars 2 and 3. Data is reported in Tables 3.2 and 3.3 which show an average radius error of respectively $70.57mm$ and $-58.17mm$.

It should be noted that the reported error refers to curvature radius not diameter. This means that when car 2 completes a semicircle, on average the model mispredicts its position by $7.057cm \cdot 2 = 14.114cm$. According to the partial data gathered, the misprediction can be up to $36cm$. Still, it is not required for the model to be perfect. Indeed the main idea behind the steering controller is to associate a reference steering input given by the identified model with a PD that corrects its error over time. Considering that it takes about 4 seconds for the car with steering input 50 and motor input 150 to complete the semicircle, the PD would have 4 seconds to correct an error of $36cm$.

Car 2				
Input	Diameter right	Diameter left	Error right	Error left
100	1340mm	1370mm	14.02mm	-0.98mm
90	1340mm	1370mm	94.18mm	79.18mm
80	1520mm	1600mm	103.90mm	63.90mm
50	2440mm	2730mm	177.73mm	32.73mm

Table 3.2: Steering model validation for car 2.

Car 3				
Input	Diameter right	Diameter left	Error right	Error left
100	1670mm	1560mm	-150.98mm	-95.98mm
90	1670mm	1560mm	-70.82mm	-15.82mm
80	1830mm	1720mm	-51.10mm	3.90mm
50	2730mm	3030mm	32.73mm	-117.27mm

Table 3.3: Steering model validation for car 3.

After some tests, it seems that the curvature radius is not affected by the motor input. The curvature radius of car 1 saturates for input above 100 (i.e. signals 110 and 120 gives exactly the same curvature radius). For cars 2 and 3 this happens even before, at input 90. However, the current diameter of the curved part of path fig8 is about 148cm. This means that car 1 and 2 can follow it, while car 3 cannot.

3.1.3 Steering controller design

The linear model used for the steering controller design is shown in the block diagram in Figure 3.5 where $U(s)$ is the desired lateral distance from the path (which is always 0), $C(s) = \frac{c}{s}$ is the constant compensation term of the controller computed by using the steering model presented above, $M(s) = \frac{m}{s}$ is a constant term that accounts for the error of the model used to determine the compensation, $\Phi(s)$ is a disturbance term that models the curve in the path, w is the wheelbase, α is the steering factor, $\delta \in [-100, 100]$ is steering input and $X(s)$ is controlled variable, the lateral distance from the path.

The model does not take into account the measurements disturbances and it considers disturbances associated with the actuator, projection on the path and slip angles negligible.

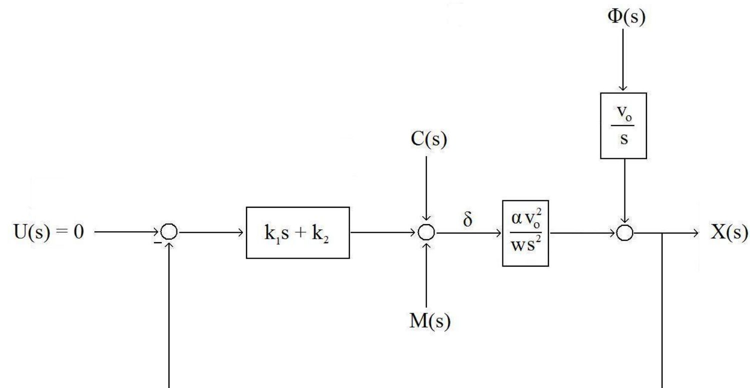


Figure 3.5: Block diagram representation of the lateral model.

Distance from the path

Defining β as the direction of the vehicle in radians, and given Equation (3.1) we have that

$$\dot{\beta}(t) = \frac{v(t)}{R(t)} = \frac{2v(t)tan(\alpha\delta(t))}{w\sqrt{4 + tan^2(\alpha\delta(t))}}.$$

The linear approximation of this function at the point $\overline{z} = (\overline{\delta} = 0, \overline{v} = v_0)$ is

$$\begin{aligned} d\dot{\beta} &= f(\overline{z}) + \frac{df}{dv}(\overline{z}) \cdot dv + \frac{df}{d\delta}(\overline{z}) \cdot d\delta \\ &= 0 + 0 + \frac{2v_0}{w}\frac{\alpha(1 + tan^2(0))\sqrt{4 + tan^2(0)} - 0}{4 + tan^2(0)}d\delta \\ &= \frac{\alpha v_0}{w}d\delta \end{aligned} \tag{3.2}$$

whose Laplace transform is

$$B(s) = \frac{\alpha v_0}{ws}\Delta(s).$$

Given the direction in radian of path $\phi(t)$, the car direction $\beta(t)$ and its speed $v(t)$, the distance from the path $x(t)$ follows

$$\dot{x}(t) = v(t)sin(\beta(t) - \phi(t))$$

whose linearization at the point $\overline{z} = (\overline{\beta} = 0, \overline{\phi} = 0, \overline{v} = v_0)$ is

$$\dot{x} = v_0 d\beta + v_0 d\phi \tag{3.3}$$

or

$$X(s) = \frac{v_0}{s}B(s) + \frac{v_0}{s}\Phi(s).$$

The controller

The controller is a PD in the form

$$\delta(t) = k_1\dot{e}(t) + k_2 e(t) = k_1(\dot{x}(t) - 0) + k_2(x(t) - 0) = k_1\dot{x}(t) + k_2 x(t).$$

Using Equation 3.3 we can write

$$\delta(t) = k_1 v_0(\beta(t) - \phi(t)) + k_2 x(t).$$

Static analysis

Putting it all together, one obtains the system represented in Figure 3.5. We want the distance from the path $x(t)$ to be 0 thus $U(s) = 0$. Calling the controller $K(s)$ and the system $G(s)$, we can write the transfer function

$$X(s) = -\frac{1}{1 + K(s)G(s)} \frac{v_0}{s} \Phi(s).$$

When the path does not have curves, its direction $\phi(t)$ is constant or in other words $\Phi(s) = \frac{q}{s}$, with q constant. In this case we obtain

$$X(s) = \frac{-qv_0 w}{ws^2 + \alpha v_0^2 k_1 s + \alpha v_0^2 k_2}$$

and for the final value theorem we can say that $\lim_{t\to\infty} x(t) = 0$. However, if there is a curve with constant curvature radius $\Phi(s) = \frac{q}{s^2}$ is a ramp. In this case

$$X(s) = \frac{-qv_0 w}{s(ws^2 + \alpha v_0^2 k_1 s + \alpha v_0^2 k_2)}$$

which means that

$$\lim_{t\to\infty} x(t) = -\frac{qv_o w}{\alpha v_0^2 k_2}.$$

To reduce the final error we can add a constant compensation term to the controller which assumes the form

$$\Delta(s) = (k_1 s + k_2)X(s) + \frac{c}{s}.$$

In this way we obtain

$$X(s) = \frac{-qv_0 w + \alpha v_0^2 c}{s(ws^2 + \alpha v_0^2 k_1 s + \alpha v_0^2 k_2)}.$$

Recalling that during a curve $\phi(t) = q \cdot t$, if the curve has curvature radius R, we can compute q as

$$q = \frac{\frac{\pi}{2}}{\frac{\pi R}{v_0}} = \frac{v_0}{2R}$$

so, in order to cancel the final error

$$c = \frac{w}{2R\alpha}.$$

If we consider the error in the model $M(s)$, the Laplace transform of the system becomes

$$X(s) = \frac{-qv_0 w + \alpha v_0^2 c + \alpha v_0^2 m}{s(ws^2 + \alpha v_0^2 k_1 s + \alpha v_0^2 k_2)}.$$

If we set c so that the effect of q is canceled, then the error is caused only by m. By applying the final value theorem one obtains

$$\lim_{t \to \infty} x(t) = \frac{m}{k_2}$$

thus the final error depends only on k_2 as reasonable to expect. Recalling that m must be a steering input, it is easy to compute its maximum value. By looking at the maximum error in the validation data in Table 3.2, I obtained $m = 13.3045$.

3.1.4 Controller calibration

The transfer function has a pole in 0 and a pair of complex poles for

$$|k_1| < \sqrt{\frac{4w}{\alpha v_0^2} k_2}.$$

Poles are

$$p_{1,2} = -\frac{\alpha v_0^2 k_1}{2w} \pm i \frac{\sqrt{4w\alpha v_0^2 k_2 - \alpha^2 v_0^4 k_1^2}}{2w}$$

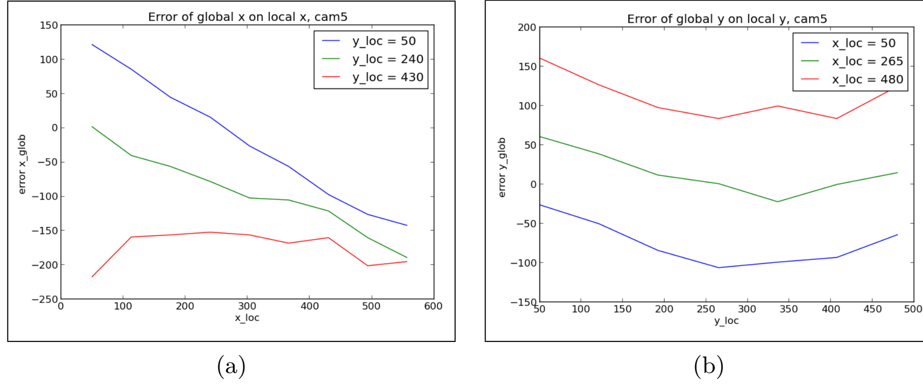
We want the final error e_f, the settling time t_s and the frequency of oscillation ω_d to be low or, in other words, we want to keep low the quantities $t_s = \frac{-4.6}{Re(p)}$ and $\omega_d = Im(p)$. With $k_1 = k_2 = 0.3$ and $v_0 = 800\frac{mm}{s}$ we obtain $e_f = 44.34mm$, $t_s = 3.37s$ and $\omega_d = 0.9314$.

3.2 CPS measurement error

3.2.1 Position linear correction

The position of the car on the testbed computed by the CPS is affected by a considerable error. I made some manual measurement of this error by finding the real position with the measuring tape and checking the computed position on the CPS and I found it to be up to $25cm$. Moreover, when the tracking of a car passes from a camera to another, the global coordinates "jump" because the position error in the transition point is different for the two cameras. From the experiments on path fig8, this leap can be up to $35cm$. This negatively affects the ability of the car to accurately follow a path which is a crucial aspect in limiting disturbances.

Figure 3.6 represents how the error varies when we move along a direction in camera 5. In order to understand the graphics, it must be said that a position on the testbed can be indicated in two different coordinate systems. The global coordinate system specifies a certain position with an absolute planar vector (x_{glob}, y_{glob}), where x_{glob} and y_{glob} are expressed in millimeters. On the other hand, a local coordinate system (x_{loc}, y_{loc}) can be associated to



(a) (b)

Figure 3.6: How the measurement error on x_{glob} changes when we set y_{loc} and we move along the x_{loc} axis on camera 5 **(a)**. How the measurement error on y_{glob} changes when we set x_{loc} and we move along the y_{loc} axis on camera 5 **(b)**. The notation x_{loc} indicates the x coordinate in pixels in the local coordinate system of the camera (i.e. the horizontal one), while x_{glob} is the x coordinate in millimeters in the global coordinate system.

the frames of each camera, and each vector represents the distance in pixels from the origin. Very similar trends where observed with camera 2.

To reduce the error, I applied a linear correction to the computed global coordinates error along both directions. The estimated error is given by

$$e = a \cdot x_{loc} \cdot y_{loc} + b \cdot x_{loc} + c \cdot y_{loc} + d.$$

The parameters a, b, c, d are computed by the CPS on startup by loading a file where the actual global coordinates of four points must be saved. The procedure that must be followed to configure this file is described in Section 6.5.

3.2.2 Heading filtering

Since the CPS extrapolates the car heading from two subsequent positions, the measurement error affects the computed direction too. In particular, when the car tracking passes from a camera to another and a jump occurs in a random direction, the measured heading is considerably compromised. This has a heavy negative effect on the steering controller ability to follow the path. I have thus implemented a model-based filter to eliminate heading jumps. The filter is applied in the car, not in the CPS because it needs the speed from the encoder. In short, the filter detects when a leap occurs and uses the model heading instead of the measurement when that happens.

First the measured heading subject to leaps is computed as

$$\gamma_{meas} = arctan\left(\frac{x_2(k\tau) - x_2((k-1)\tau)}{x_1(k\tau) - x_1((k-1)\tau)}\right)$$

Algorithm 1 Filter the car heading to remove jumps

```
if |γ_meas − γ_model| > k_1 then
  γ = γ_model
else if |γ_meas − γ_model| > k_2 and |dist_meas − dist_model| > h · dist_model then {This is the uncertain case}
  γ = γ_model
else
  γ = γ_meas
end if
```

where $k\tau$ is the current step and (x_1, x_2) represents the car position in the absolute coordinate system. The predicted direction for the current step is

$$\gamma_{model} = \frac{v((k-1)\tau)\tau}{R}$$

where R is the curvature radius which can be substitute by Equation (3.1) to obtain

$$\gamma_{model} = \frac{2v((k-1)\tau) \cdot tan(c \cdot \delta((k-1)\tau))\tau}{w\sqrt{4tan^2(c \cdot \delta((k-1))}}$$

where c is the steering factor and w is the wheelbase of the car. Then we compute the measured distance that the car has covered from the previous step

$$dist_{meas} = \sqrt{(x_1((k-1)\tau) - x_1(k\tau))^2 + (x_2((k-1)\tau) - x_2(k\tau))^2}$$

and the predicted covered distance

$$dist_{model} = v((k-1)\tau)\tau.$$

The algorithm works as shown in Algorithm 1.

Some leaps occur more or less in the direction of the car but they can be still detected because the car covers a much longer (or shorter) distance than it should. The second if-statement detects this kind of leaps. Thresholds k_1, k_2 and h have been determined empirically to be respectively $30degrees$, $20degrees$ and 0.35.

One way to interpret this filter is to consider it as a simple camera change detector that applies a Kalman filter with time-varying error covariance. The measurement error covariance is 0 when camera change is not detected (i.e. the Kalman filter keeps the measurement) and non-zero otherwise. On the contrary, the a priori estimate error covariance is 0 when camera change is detected (i.e. the Kalman filter keeps the model) and non-zero otherwise.

3.3 Other disturbances

In this section d_{cap}, d_{slope} and d_{steer} from Figure 2.2 are discussed and analyzed. These terms represent respectively the disturbances on the car speed caused by the power filter capacitor transient, the testbed inclination and the steering.

3.3.1 Effect of steering and testbed slope

I made car1 run on circles for 50 seconds with fixed PWM and steering input for a total of 16 runs. Every run was performed starting from the same battery voltage of $16.7V$. All the images below are obtained by filtering the encoder signal with a moving average window to discard the high frequency components of the noise.

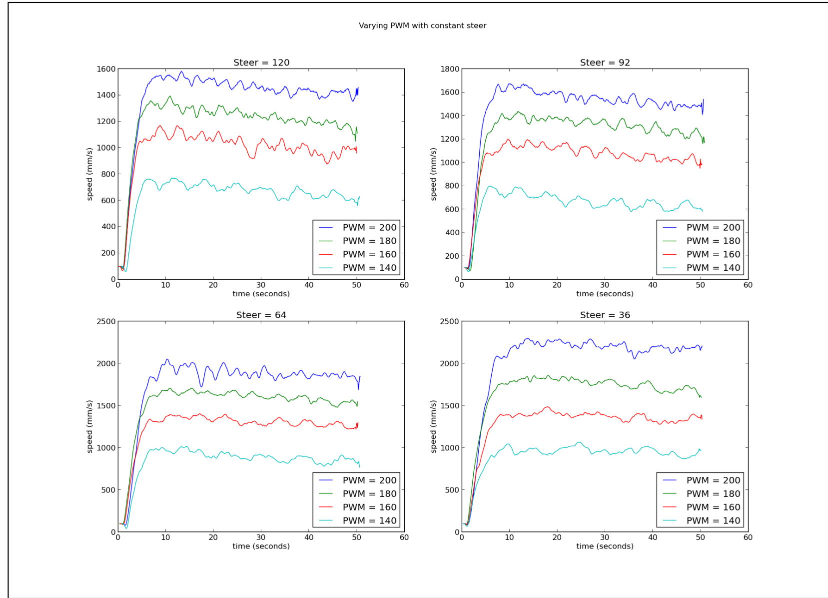
Figure 3.7 shows the speed response to PWM and steering step input. It is clear that the relationship between PWM and velocity is quite linear (Figure 3.7a). For some reason, when the steering input is high, the speed observed with PWM 140 is slightly lower than expected. The reason why the steady state speed slowly decreases in time will be clarified in Section 3.3.2. On the other hand, the steering effect on speed seems to be a little bit more complicated (Figure 3.7b). Velocities observed for steering 92 and 120 are always very close. This may be due to the fact that curvature radius for the two steering input are very similar. Still the speed is not linear with respect to the curvature radius because the discrepancy between the speed for steering 36 and 64 (which I measured to have respectively a curvature radius of roughly $200cm$ and $100cm$) tends to be reduced when the PWM decreases.

As a final observation, it should be noted the presence of an oscillation in the speed dynamics whose frequency increases with the speed. This oscillation is caused by the fact that the testbed is not completely flat. There are slight slopes that can be easily observed with a spirit level.

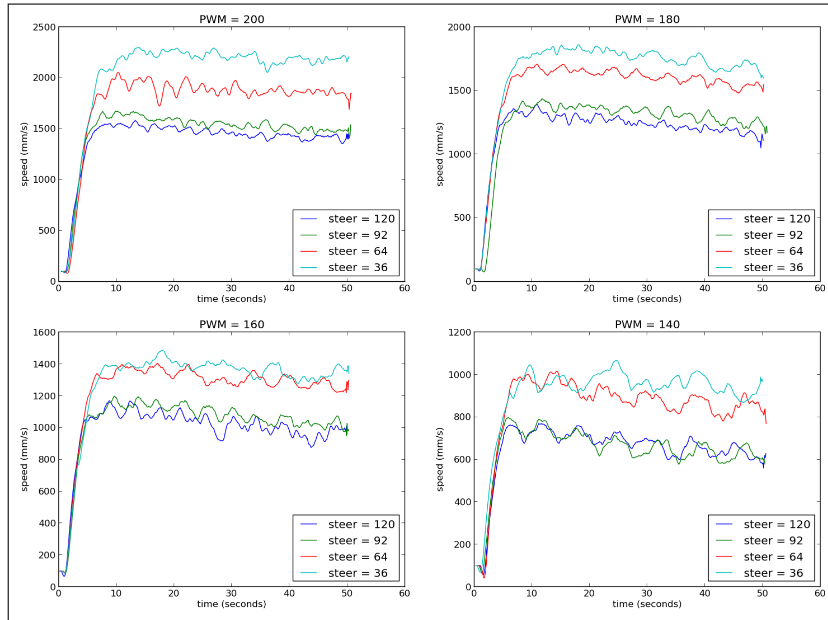
In conclusion, the car speed depends indeed linearly on the PWM input signal as it is modeled. Moreover the steering and the testbed slope introduce a complex dynamics that is not taken into account by the model and should be considered as disturbance.

3.3.2 Effect of the power filter capacitor

After gathering the first experimental data as described in Section 3.3.1, I further investigated the reason why the car speed slowly decreases in time. New data was acquired by running car 1 on a circle for a much longer time (between 200 and 250 seconds) with constant steering and engine input. The steering has always been set to -64. The starting position of the car has always been about $(3000, 3600)$ in global coordinates so that the effect

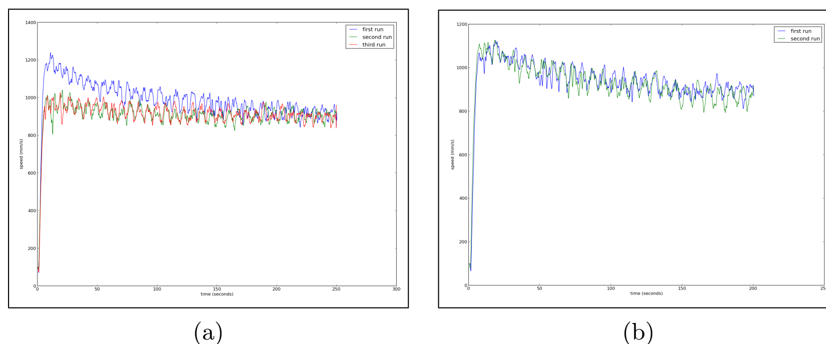


(a)



(b)

Figure 3.7: **(a)** speed-PWM relationship for different steering values and **(b)** speed-steering relationship for different PWM values.



(a) (b)

Figure 3.8: Speed dynamics of car 1 after being run 3 times in a row **(a)** or twice with a 200 seconds pause in between **(b)**.

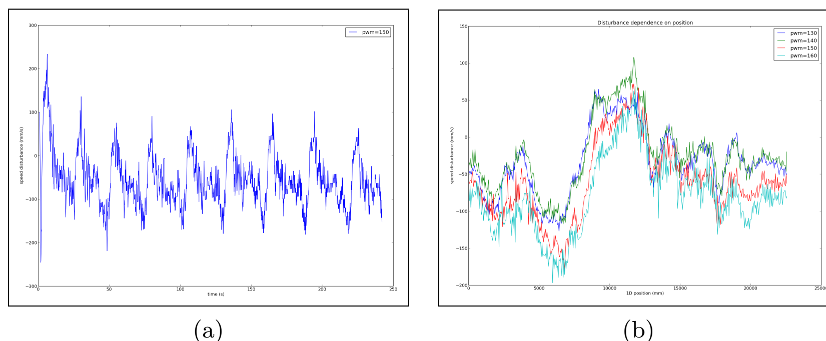
associated to the testbed inclination would be similar in all experiments. The battery voltage was measured both before turning the car on, and with the car on but just before it started moving. Various experiments with different configurations of battery voltage and PWM input were explored. The full data gathered and the detailed description of how the experiments were performed are available to the laboratory team.

Figure 3.8 summarizes the main results. In the first experiment (Figure 3.8a), car 1 was run 3 times in a row. It is very evident that in the first run the speed reaches a peak and slowly decreases to a steady state value. The speed recorded during the second and third run starts more or less from this steady state value. Figure 3.8b was obtained instead by running the vehicle for about 200 seconds, pausing it for another 200 seconds and then starting it again. Here the car speed has the same dynamics in both runs.

This behavior is likely caused by the capacitor that filters the power source. When the car is not running the circuit is in a steady state condition but when it runs the equilibrium moves and the capacitor slowly discharges. The same behavior was confirmed by experiments with both cars 2 and 3. Recalling Equation (2.1), the actual speed dynamics may thus be described by

$$\begin{aligned} \dot{v}(t) &= av(t) + b + fu(t) + d_{steer}(t) + d_{slope}(t) + d_{cap}(t) \\ &= av(t) + b + fu(t) + d_{steer}(t) + d_{slope}(t) + ge^{-\frac{t}{\tau}}u(t) \\ &= av(t) + b + (f + ge^{-\frac{t}{\tau}})u(t) + d_{steer}(t) + d_{slope}(t) \end{aligned} \tag{3.4}$$

where the motor gain decreases exponentially to reach the steady state value f with time constant τ, and g is a parameter defining the amplitude of the exponential decay. Further investigations showed that the PWM does not affect neither τ nor g.



(a) (b)

Figure 3.9: Dependency of path-dependent disturbance on time **(a)** and unidimensional position **(b)**.

3.3.3 Mathematical formulation of the engine compensator

I implemented an engine compensator which reduces the transient due to the capacitor, and the effects of steering and testbed inclination on the speed by introducing a compensation term in the PWM signal. Recalling Equation (3.4) we can write

$$\dot{v}(t) = av(t) + b + (f + ge^{-\frac{t}{\tau}})(u(t) + c(t, y)) + d_{path}(y) \tag{3.5}$$

where $d_{path}(y) = d_{steer} + d_{slope}$ and $c(t, y)$ is the compensation term. Note that $d_{path}(y)$ depends on the position along the path y. This comes from the consideration that the slope and the steering input are roughly the same at the same point of the path. For this reason, d_{path} is referred to as "path-dependent disturbance". Indeed, Figure 3.9b plots the integral path-dependent disturbance $D_{path}(y)$ (calculated as the subtraction between the speed measured by the encoder and the model) against the position along the path for different values of PWM. The dependency is clear. It should be noted that, even if the model neglects it, the disturbance seems to be slightly influenced by the PWM. Moreover, Figure 3.9a shows the trend of $D_{path}(y)$ in time. As one can see, the graphic exhibits roughly the same pattern whose period coincides with the completion of a lap.

By imposing the cancellation of the capacitor effect and $d_{path}(y)$ it is easy to obtain

$$c(t, y) = -\frac{ge^{-t/\tau} + d_{path}(y)}{ge^{-t/\tau} + f}.$$

An automatic procedure for the parameters calibration (i.e. g and τ) and the estimation of $d_{path}(y)$ as a linear piecewise function is available. Detailed instructions that explain how to use the script can be found in Sections 6.4.2 and 6.4.3.

It must be noted that the engine compensator assumes that the car starts with the capacitor in equilibrium. When the car is stopped, the capacitor

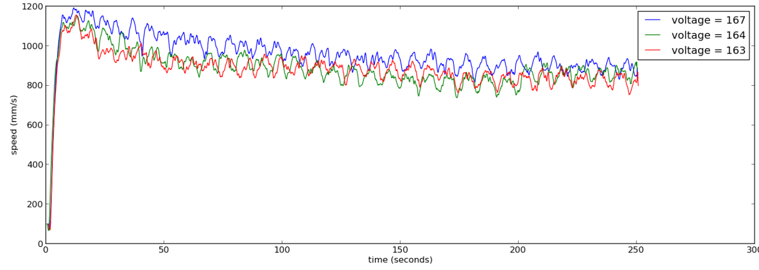


Figure 3.10: Speed dynamics variation with battery charge level.

starts recharging again. In order for the compensator to work, one needs to wait until this process is completed. It takes about 200 seconds for the transient to reach its steady state but if the car was run for a shorter time the await can be reduced. Plug the charger to speed up the process.

3.4 On-line estimation of the motor gain

The motor gain f is not a constant parameter of the model. In the first place, it depends on the battery charge level of the vehicle. Figure 3.10 shows how the speed dynamics change with the battery state. As reasonable, the motor performance decreases with the voltage supplied. In parallel, the performances of the motor can change from one day to another even if the battery charge level is the same. It is not clear why this phenomenon takes place but we speculated it depends on the status of the electric circuits which is subject to forces and tensions that can slightly alter its conditions when the car is moving. For this reasons, the motor gain is considered to be time-dependent and it is estimated online by using the adaptive control technique called "MIT rule" [3] which basically adopts gradient descent to minimize a cost function. The cost function is defined as:

$$J = \frac{1}{2}e(f)^2 = \frac{1}{2}(v_m - v_p(f))^2$$

where v_m is the measured speed, $v_p(f)$ is the speed predicted by the model (which depends on the gain f). Applying the gradient descent method gives the updating rule

$$f(k+1) = f(k) - \gamma e(k+1)\frac{de}{df}(k+1).$$

The parameter γ is set very low since the gain does not change rapidly.

Chapter 4

Experimental results

4.1 Disturbances minimization

Position correction

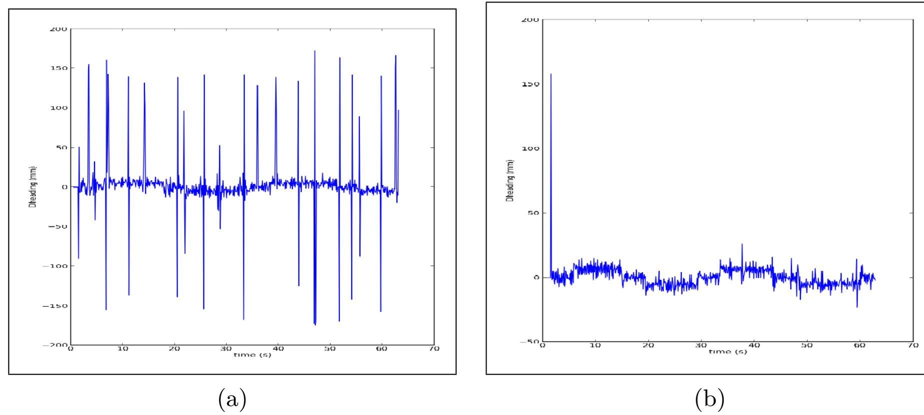
The first visible improvement given by the position linear error correction concerns the car path-following performance: the car stays much closer to the given path. Secondly, the error of the position in camera 5 and 2 is constantly below $10cm$ and rarely above $5cm$. Moreover, since the absolute position error is reduced, the "leaps" that occur when the tracking pass from a camera to another is reduced too. Indeed, I was not able to measure a leap above $20cm$ (against the $35cm$ without position correction).

Heading filtering

Figure 4.1 shows the heading difference in time defined as $d\gamma = \gamma_m((k+1)\tau) - \gamma_m(k\tau)$ where $\gamma_m(t)$ is the measured direction of the car at time t. As you can see, basically all the leaps, big and small, caused by the change of the tracking camera are removed by the filter (Figure 4.1b). It is easy to understand how important this is for path-following performances.

Input	Old error norm	New error norm
Position correction	$\lvert e\rvert_\infty \simeq 35cm$	$\lvert e\rvert_\infty \simeq 20cm$
Distance from path	$\lvert\bar{x}\rvert = 173mm$	$\lvert\bar{x}\rvert = 49mm$
Path-dependent disturbance	$\lvert d_{path}\rvert_\infty \simeq 200mm$	$\lvert\lvert d_{path}\rvert_\infty \simeq 50mm$

Table 4.1: Summary of disturbances norms before and after minimization. The average distance from the path $\bar{x}$ refers to a lobe of path fig8CL.

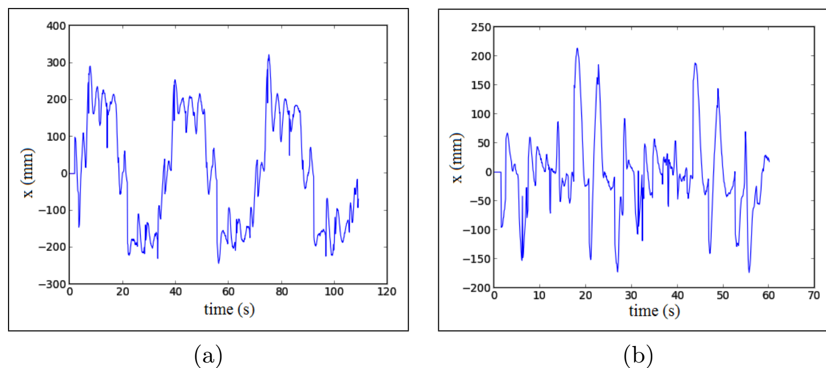


(a) (b)

Figure 4.1: Heading jumps with **(b)** and without **(a)** filtering.

New steering controller

Figure 4.2 shows the measured distance from the path $x(t)$ with the old and the new controllers. In both cases the dynamics of $x(t)$ is quasi periodic with each period coinciding with a lap completion, and each half-period coinciding with the end of a lobe on fig8CL. Notice that Figure 4.2a represents three whole periods while in the experiment represented in Figure 4.2b the vehicle was run for a shorter time. With the old controller the distance from the path increases at each curve, it oscillates around $200mm$ and then it gets closer to the path again in the straight section. The sinusoidal form is caused by the fact that the car needs to steer left in the first lobe and right in the second one. In this way the vehicle stays on the right side of the path in the first lobe and on the left one in the second lobe. On the other hand, with the new controller the car oscillates between the right and the left



(a) (b)

Figure 4.2: Distance from the path $x(t)$ with the old **(a)** and the new **(b)** steering controller of a car running on fig8CL. A negative distance means that the car is on the left of the path with respect to the path direction, while a positive distance represents a vehicle on the right side.

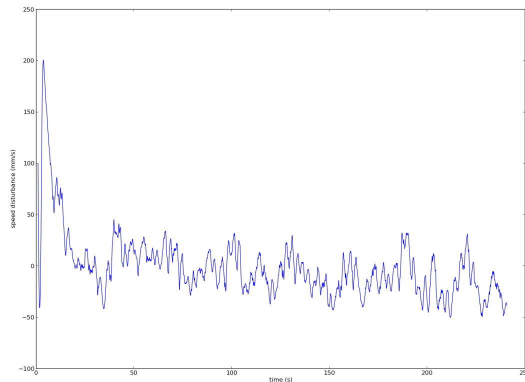
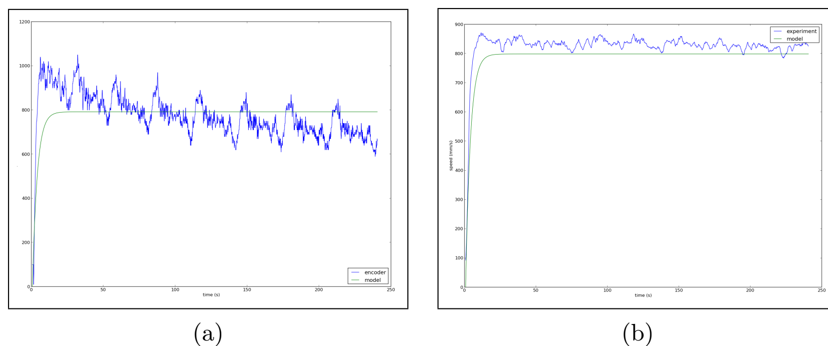


Figure 4.3: Final disturbances on the car speed.

side throughout the lobe of the path. This significantly reduces the average disturbance which is particularly advantageous when it is expressed in the form given by Equation (2.4). The average distance from the path in a lobe can be found in Table 4.1. Moreover, the new controller keeps $x(t)$ below $10cm$. The only times when this does not hold correspond to change of tracking camera when position leaps occur.

Compensator and path-dependent disturbances

Figure 4.4a compares the car speed dynamics predicted by the model with the one obtained by running car 2 on path fig8CL with constant motor input without the engine compensator described in Section 3.3.3. The superposition of the disturbance caused by the capacitor (i.e. the transient) with the path-dependent disturbance (i.e. the quasi periodic oscillation that coincides with a lap completion) is quite evident. When the engine compensator is activated the disturbances are greatly reduced and the car speed trend



(a) (b)

Figure 4.4: Compensator and path-dependent disturbances with **(b)** and without **(a)** engine compensator. In both cases the blue line represents the experimental data while the green one shows the trend predicted by the car model.

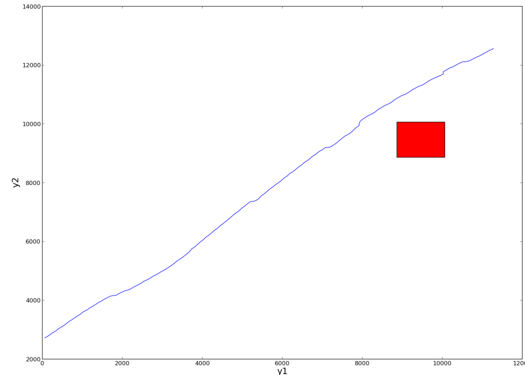


Figure 4.5: Trajectory of the two cars (blue line) and bad set (red rectangle). On the x-axis the position of car 1, on the y-axis the position of car 2.

assumes the shape given in Figure 4.4b. The reason why the speed is constantly above the model is due to an erroneous estimation of the motor gain parameter of the model. This is attenuated by the online estimation of the motor gain (see Section 3.4) whose effect is shown in Figure 4.3. The spike at the beginning is mainly due to a defect in the encoder measurement, but after that the speed does not get further than $50\frac{mm}{s}$ from the model.

4.2 Algorithm testing

The test of the collision avoidance algorithm with two cars was successful. The experiment lasted about 3 minutes and 30 seconds and the two vehicles were run with (different) constant desired PWM. During the experiment, the supervisor detected a collision and overwrote the desired input for 8.62% of the time. Figure 4.5 shows an instance of the trajectories of the two cars approaching the intersection. Notice how the slope of the curve slightly increases in the middle section and decreases again once the intersection is passed. Since the desired motor input is constant in both cars, the trajectory should appear almost as a straight line (constant motor input results in constant speed) but the supervisor corrects it to avoid the collision which is represented in the figure by the red rectangle.

On the other hand, I was not able to make the experiment work after adding a third vehicle. The problem is that the prediction is currently too conservative too find a schedule able to solve certain configurations of states of the three cars running together. This can be solved in two ways: by decreasing the boundaries of the disturbances used by the predictor and by lengthen the paths. Both approaches are currently being investigated.

Chapter 5

Simulator

The simulator emulates the laboratory environment by computing the dynamics of the vehicles, taking into account disturbances and noise. It allows to qualitatively and quantitatively study the effect of isolated disturbances on the system. Moreover it works as a first test bench for models and software solutions deployed to vehicles and the CPS, and it makes simpler the debugging process. This considerably sped up the software development. Finally, since this project is the result of a collaboration between Politecnico di Milano and MIT, it provides the ability to simulate experiments to those who do not have access to the laboratory situated in Cambridge, Massachusetts.

5.1 Architecture and features

The simulator is implemented in C. It works on Ubuntu Desktop 12.04 LTS and it should not have problems also with other distros of Linux but, since it uses the POSIX library for threading and network communication, it cannot work on Windows unless one uses some (untested) workarounds such as using a POSIX porting for Windows. However it should work on Mac since it is Unix-based.

The simulator communicates to the program executed on vehicle computer (called "`ca2`") exactly in the way CPS does. However full virtualization is not supported, `ca2` is paravirtualized. Indeed, the software on cars makes extensive use of a proprietary library that provide an API to communicate to the Brainstem microcontroller which would have taken more time than what was worthy to reimplement. The program can be easily compiled in simulator or car mode by setting the value of a macro.

Two different threads constitute the server and the client part of the simulator. The server receives messages from `ca2` that are supposed to be directed to the CPS and to the microcontroller. These include for example the fitness estimation (CPS) and the input for steering and motor (micro-

controller). This information is saved in variables shared between the server and the client. The latter takes care of simulating the dynamics of vehicles based on the input sent by cars themselves. This is accomplished by integrating a system of differential equations with the GNU Scientific Library. The result is then sent back to cars exactly like the CPS does.

The program includes accurately in the model the effects of the capacitor, the ones of the lateral dynamics and the position measurement error which includes the jumps of position caused by the change of tracking camera. It also simulates the effects of the steering and the testbed slope but it does not take into account the part of the disturbance which is unpredictable. Basically it assumes the path-dependent disturbances to be the one identified (see Section 6.4.3) which are the same that the engine compensator cancels. Moreover, the encoder speed measurement noise is not currently implemented.

Since the CPS code was Windows-specific, it was not possible to virtualize that part of the laboratory like it has been done for `ca2`. The algorithms are replicated and they must be updated every time the CPS is modified to keep the simulation consistent.

5.2 User guide

Here follows a small guide on how to compile and use the simulator. All files can be found in the main folder of the projects and all the paths below are relative to it.

To compile both `ca2` and the simulator I have implemented the shell script "`scripts/compile.sh`". First of all, check that the `CODE_IN_CAR` constant in "`src/car_src/const_car.h`" is set to 0. This allows to compile the code without the requirement of the Brainstem library and it configures the sockets to connect to localhost instead of the IP addresses of the CPS and the cars. From the main folder, just type in the console

```
cd scripts
./compile.sh
```

The two executables `ca2` and `simulator` should appear in your main folder. Other than the standard POSIX library, the simulator uses GSL (Gnu Scientific Library) so if you have problems compiling, make sure it is installed.

From the main folder, to run a simulated experiment first run the simulator.

```
./simulator
```

and then run the car code with the normal syntax on a different console.

```
./ca2 paths/figX_precise.txt
```

Where `X` can be `8AL`, `8CL` or `0`. The suffix "`_precise`" in the path specification file name differentiates it from the previous one which defines the path with a very small number of points. The path files I have used are contained in the folder `paths/`. Specifying a great number of points allows the car to follow its path more effectively. The simulator can handle experiments with more than a single car. You just have to compile different `ca2` executable (make sure that the constant `CAR_NUM` in `src/car_src/const_car.h` is different for each compiled `ca2` and that `NUM_CARS_EXP` in `src/util/constants.h` is set appropriately). Then run the simulator normally and run the car executables, each one on a different console and with its own path. Press the *Space* key in the `ca2` console to stop the car. The simulator automatically halts when all the cars stop.

In order to speed up the computation, the information displayed during the simulation is minimal. However both `ca2` and the simulator leave a debug file rich of data. I wrote a Python script called `plot_trajectory.py` that visualize graphically the information contained in these debug files. It can be used to speed up considerably the debugging of the experiment both simulated and on the test-bed. See Section 6.4.1 for more information about its usage.

Chapter 6

Laboratory manual

6.1 Software overview

6.1.1 Code organization

All the code is contained in the main folder of the project. All paths below are to be intended relative to it. The main folder contains 4 subfolders:

- `src/`: contains the C code divided in 3 folders
  - `src/car_src/`: contains the code specific for the car (i.e. the program `ca2`).
  - `src/sim_src/`: contains the code of the simulator.
  - `src/util/`: contains the code shared between `ca2` and the simulator.
- `scripts/`: contains all the Python scripts that can be used for example to compile the simulator, identify parameters and debug the experiment.
- `paths/`: contains the new paths definition files used by the cars which specify also the steering compensator directives and the path-dependent disturbances functions.
- `data/`: various data gathered during experiments on the test-bed.

6.1.2 Summary of changes

The new version of the CPS (called `AndreaCPS`) is not compatible to previous versions of `ca2`. Viceversa my `ca2` is not compatible with previous versions of CPS. This is because the communication protocol between CPS and ca2 is now different. In general I would suggest using `AndreaCPS` instead of `KevinCPS` or `LeoCPS` because they have a bug that involves the initial target

detection which is corrected in the new one. I did not check CPS versions that are older than `KevinCPS` but it is likely that those versions do not have this bug since computer 1 and 2 do not send their information to computer 0 before sending them to cars.

ca2

- Grouped utility functions in a separate folder that is shared with the simulator. This makes the code more modular.
- Reimplemented the target detection algorithm so that the car and CPS can work with paths defined by a large number of points.
- Steering controller has been reworked, the monodimensional speed has been substituted by the one read from the encoder, the PD has been calibrated and a path-dependent compensator has been added. The compensator can be configured through the path specification file (see Section 6.6.1).
- Implemented a new filter for the heading measure to cancel the effect of leaps (see Section 3.2.2).
- Set the limit of steering input to (-100,100) since the curvature radius saturates above it.
- Implemented the online estimation of the engine gain parameter (see Section 3.4).
- Implemented speed disturbances correction which uses a path-dependent disturbance function defined in the path specification file.

CPS

- Fixed a bug that affected the initial target dectection for cars that were not tracked by computer 0 and buffer overflow bug.
- Implemented linear error correction for the computation of camera position. This visibly improves the path following of cars (see Section 3.2).
- The computer that sends the data to cars is now computer 1 (not the slower computer 0 anymore). This is easily configurable through the constant `SENDER_COMP_NUM` in CPS.h.
- The sender computer now knows which camera is currently tracking which car and this information is sent to cars.

Communication protocol

- Now the car sends its personal data (encoder speed, estimated fitness, desired pwm, etc.) to CPS that forwards this data to all the cars. If the encoder of a car does not work you can set to 0 the constant `USE_ENCODER` in `car_src/const_car.h` and the CPS will estimate the car speed from differentials.

- Each car listens on a different port. This was required to make the simulator handle multiple cars and I extended it to the CPS for coherence.

Paths

- Modified the path specification file format in order to include the steering compensator directives and the definition of the speed disturbance function.

- Slightly narrowed the path fig0 since car3 used to go too close to the computers where the camera distortion is very high and the tracking was often lost.

6.2 Software configuration

There are 5 files which contain basically all the constants in the code. These should be the ones to check whenever you want to configure something.

- `car_src/const_car.h`: contains the compilation options that activate/deactivate the features of the code in the car and few other constants specific for the car code such as the car number, the initial pwm and when it has to stop.

- `car_src/supervisor.c`: contains few types that only the supervisor uses and the constants that define the boundaries for the disturbances.

- `sim_src/const_sim.h`: this contains very few simulator-specific constants which include the real fitness parameters of the cars, the paths followed by each car and whether to apply the measurement noise during the simulation.

- `sim_src/model.c`: contains the amplitudes of the noise signals that are applied to the physical quantities by the simulator.

- `util/constants.h`: this basically contains all the others. They range from network configuration to control parameters to physical constants.

The specific effect of each constant is documented in the code. Beside the types defined in `car_src/supervisor.c`, all the types definition (and their documentation) can be found in `util/types.h`.

6.3 Car software

Since the code of the car is split into modules, there are a couple of differences concerning the uploading and compilation of the source code.

Create your project

The folder on cars' computers that I have used is located at `/Desktop/brainstem/aProject/Andrea/`. The structure of the code is changed so I had to motify the makefiles to make the compilation working. As a consequence, if you want to create your own folder to expand my branch of the project you should start by copying mine.

Upload a new code version

The car source to compile and run needs both the folders `car_src/` and `util/`. If you write a new version of the code and you want to upload it on the cars with sftp, you must upload both those folders. Of course you can avoid updating both if the changes affect only the code in one of the two folders.

Compilation and execution

The compilation and the execution of the new code does not differ from the previous versions. Make sure that the constant `CODE_IN_CAR` in file `car_src/const_car.h` is set to 1 and then type on the console

```
cd /Desktop/brainstem/aProject/Andrea
make clean
make new
```

in case you do not want to recompile the whole project but only the changed files you can simply type `make` instead of `make clean` followed by `make new`. To run the code type

```
cd /Desktop/brainstem/aDebug/aUnix/i686
./ca2 figX_precise.txt
```

The path specification files I have used have the suffix "`_precise`" in their name in the same way of the simulator. As usual, you can find these files in the folder `/Desktop/brainstem/aDebug/aUnix/i686/`.

6.4 Scripts

The scripts are written in Python. At the beginning of the script, after imports there is always a “configuration” section (delimited by comments) which contains all the (documented) constant variables that must be set appropriately to make the script do exactly what you want.

Most of the scripts rely on the debug file that `ca2` leaves after execution. As you can imagine, the parsing of the debug file is strongly dependent on its format. This means that if you change the name of the attributes written in the file for example, the script might not work as expected. The parsing does not depend on the order of the attributes so it is safe to change it. All the scripts rely on the parser class `DebugData` in `scripts/util/parser.py`. This design makes it easy to make adjustments in case the file format changes. However, if you modify the internal data structure of `DebugData` (e.g. change the name of one of its member variables) you will have to modify the scripts that access to that data. Check the class documentation in the code for more information. Note that `DebugData` can be used to parse only the debug file produced by `ca2`, not the one written by the simulator.

The scripts often uses `matplotlib` to plot graphics and `numpy` and `scipy` for advanced numerical routines. If you have problems running the scripts, make sure these Python libraries are installed on your system.

6.4.1 Debug information visualization

The script `plot_trajectory.py` allows to analyze offline the data of an experiment. The program `ca2` saves a file in the same folder of the executable called “`data_carX.txt`” which contains lots of data describing position, speed, heading, etc. of the car in each car cycle. Similarly, the simulator produces a file called “`debug_sim.txt`”. The main difference between the two files is that the first one reports values of the physical quantities that are affected by measurement noise while the second one reports also the real state of the system. Moreover, the car can record the status of the internal variables which are unknown to the simulator.

The script normally parses only `data_carX.txt` and thus it can be used to debug both simulated and experimental data. You can have a graphical representation of the information contained in `debug_sim.txt` using the command line option “`-b`” (see below for full description). In this case the script integrates the information in the two files.

Configuration and usage

1. Run a simulation (or an experiment) to obtain the debug files. Do not stop the program with `CTRL+C` or the debug file will not be terminated correctly.

2. Open `plot_trajectory.py` and set the car number and the paths to debug files you want to analyze.

3. Run the script by typing "`python plot_trajectory.py`" on the console.

4. Focusing the console, use keys (lowercase) '`n`' to go to the next step, '`p`' to go to the previous one, '`q`' to quit.

The script displays four plots. On the top left corner there is the path and the trajectory of the car with some information. The little red cross represents the target point of the path. The other 3 graphics show the evolution of some attributes in time. You can configure which attribute to plot with the following precedure.

- Open `plot_trajectory.py`.
- Use your editor's search feature (usually `CTRL+F`) to find the string "TODO". You should find a code section titled "TODO configure which info to plot".
- Assign to variables `graphX_vals` the arrays containing the info you want to be plotted. The array will likely be a member variable of the class `DebugData` defined in `scripts/util/parser.py` or one of the `*_sim_vals` arrays defined inside `plot_trajectory.py` that represents the data extracted from the simulator debug file. Check the code documentation in `DebugData` and in the `plot_trajectory.py` code section called "SIMULATOR DEBUG VARIABLES" for details. Of course you can print any array-like structure as long as it has the same length of `lab_data.time`.

Red crosses show the value of the specified attribute at the current step.

Command line options

- `-b`: parse and integrate both `data_carX.txt` and `debug_sim.txt`. When this option is used, the trajectory shown and the information presented in the top left corner graphic are the real one, not the ones affected by measurement noise that cars perceive. This is useful to check that everything is going fine with the simulator. Note that the simulator and ca2 are asynchronous (the clock rate of the simulator is much higher than the one of the car), thus, in order to integrate the two files, the script must replicate/cut some data according to the desired clock cycle specified through the option `-c`.
- `-c` (default 0.1): the time between two represented steps. For example, if you call the script by typing `python plot_trajectory.py -c1.0`, pressing '`n`' will show the status of the car 1 second later.

- `-t` (default 0.0): the initial time shown by the script. For example, if you want to investigate an event that happened 40 seconds after the beginning of the experiment, you can invoke the script as `python plot_trajectory.py -t40.0` instead of starting from the beginning and press '`n`' until you reach that point.

- `-s` (default 20): the number of segments in the trail of the represented trajectory.

6.4.2 Model parameters identification

The script `identify_model.py` allows you to identify the parameters of Equation (3.5). Parameters a and b are used by the predictor; g and τ are needed to cancel the capacitor effect. Given the instability of the motor gain f, this will be estimated only for the purpose of the parameters identification but during the experiment it will be estimated online.

To speed up the process there is a script called "`identify_model.py`" in `scripts/` that estimates the parameters to fit the model to the experimental data by minimizing the mean average error. The script needs as input two debug files from `ca2` taken by running the car in circle for about 3-4 minutes at constant PWM. The two experiments must be run at different PWM with the same steering input. In order to ensure that the fitness parameter is the same in both the experiments, it is important for the battery charge level to be the same. The detailed procedure is below.

Run the car in circle: configuration

The parameters identification procedure requires to run several times the car in circle. This section explains how to configure `ca2` for this purpose.

1. Decide which constant PWM and steering signal to use for the experiment.

2. Turn on the car and connect to it. You do not need to turn on the CPS.

3. Move to the project folder (e.g. `cd /Desktop/brainstem/aProject/Andrea`).

4. Open the `const_car.h` file with an editor (e.g. `nano car_src/const_car.h`).

5. Set the constants `START_PWM` and `START_STEER` are set to the values you have decided at step 1.

6. Be sure that the following configuration constants are set as below

```
#define DURATION_EXP 2400 // This is flexible but consider
                          // that the transient given by
                          // the capacitor ends after
                          // about 200 seconds
#define CODE_IN_CAR 1
#define GET_POSITIONS 0
#define USE_ENCODER 1
#define SUPERVISOR 0 // This is not mandatory but it will
                     // save computational resources and
                     // avoid printing error messages on
                     // the console
#define PATHPLAN_ON 0
#define DEBUG 1
```

7. Compile the code with `make`.

8. Move to the executable folder (e.g. `cd ../../aDebug/aUnix/i686`).

9. Put the car on the testbed and run it. You can specify any path as argument of `ca2`, it will be ignored (e.g. `./ca2 fig8CL_precise.txt`).

10. During the whole experiment check the car. Sometimes the center of the circle that the car follows moves, and the car may hit the whole.

Parameters identification procedure

1. Unplug the charger, use the multimeter to record the voltage level of the battery and plug it in again.

2. Run the car in circle as described in Section 6.4.2. For the PWM and steering input I would suggest respectively 160 and -50 (negative values make the car steering left).

3. Connect through sftp and save the `data_carX.txt` file.

4. Plug the charger again and charge the battery to the same voltage measured before the first experiment.

5. Once the voltage is the same as before, run the car in circle for the first time. Keep the steering input at the same level (-50) but change the PWM (I suggest 140). It is very important to run this second experiment as soon as possible. Try to perform it the same day of the first one otherwise the fitness parameter might have changed.

6. Collect the new `data_carX.txt`. Pay attention to not overwrite the previous file or you will have to perform the first experiment again.

7. Open the script `identify_model.py` and set the configuration constants. You simply have to specify whose car you want to identify the parameters and the paths to the two debug files you just obtained.

8. Run the script `identify_model.py`. It will print on the console the five parameters you need and show you a graphic with the experimental data and the model fitting that you can use to check that everything went fine with the identification.

9. You now have to make the rest of the code (the scripts, `ca2` and the simulator) aware of the new parameters. Open `scripts/util/model.py` and set the variables `a`, `b`, `g`, `T` as documented in the code. Similarly, open `src/util/constants.h` and set the constants `ALPHA`, `BETA`, `GAMMA`, `TAU` which correspond to a, b, g and τ.

All the data used for the parameters identification are stored in `data/identification/`.

6.4.3 Path-dependent disturbance identification

The script `path_disturb.py` identifies the function $d_{path}(y)$ in Equation (3.5) that is used by the compensator to cancel the path-dependent disturbances. It takes two debug files from `ca2` and it prints on the console the disturbance function in a format that `ca2` understands (see Section 6.6.2). The first debug file is obtained by running the car in circle. This is needed in order to estimate the motor gain f. Indeed, since at this point we cannot assume the disturbances to have average 0, the fitness online estimation does not work. The second debug file must be obtained by running the car on the path with the engine compensator canceling the transient due to the capacitor. The detailed procedure is below. The data used for the disturbance function identification are stored in `data/path_disturb/`.

Disturbance identification procedure

1. If you did not do it yet, add to the path specification file the steering compensator directives (see Section 6.6.1).

2. If you did not do it yet, identify the model parameters as described in Section 6.4.2. If you have performed the model parameters identification today, you can skip to step 6 and use the debug file obtained from one of the two running in circle to estimate the fitness parameter.

3. Unplug the charger, use the multimeter to record the voltage level of the battery and plug it in again.

4. Run the car in circle as described in Section 6.4.2. For the input signal I suggest 140 PWM, -50 steering. The steering should be the same as the one used for the model parameters identification. The PWM can be different.

5. Plug the charger again, connect to the car with sftp save the file `data_carX.txt`.

6. Open `path_disturb.py` on your computer with a text editor and fill the configuration section. You must set `ONLY_FITNESS` to `True` and specify the car number, the path of the path specification file, the name of the debug file and its containing folder. It is not important to set `PATH_PATH_DATA` since you do not have that file yet.

7. Run `path_disturb.py`. It will output the estimated fitness value.

8. Go back to the car console and move to the project folder (e.g. `cd Desktop/brainstem/aProject/Andrea`)

9. Open `constants.h` with a text editor (e.g. `nano util/constants.h`). Set the correct element of the array `START_FIT` to the fitness value that the script output.

10. Open `const_car.h` with a text editor (e.g. `nano car_src/const_car.h` and set `START_PWM` to the same value used for the experiment in circle. Be sure that the following constants are set as below

```
#define START_STEER 0
#define DURATION_EXP 2100
#define CODE_IN_CAR 1
#define GET_POSITIONS 1
#define USE_ENCODER 1
#define SUPERVISOR 0
#define PATHPLAN_ON 1
#define STEER_COMPENSATOR 1
#define ENGINE_COMPENSATOR 2 // Set this to 2 makes the
                             // compensator cancel the
                             // capacitor effect while
                             // ignoring the online
                             // estimation and keeping the
                             // fitness value you just
                             // specified
#define DEBUG 1
```

11. Compile the code with `make`.

12. Move to the executable directory (e.g. `cd ../../aDebug/aUnix/i686`).

13. Open the path specification file and delete all the lines that start with a '*' character which represent the previous identified disturbance function.

14. Now turn off the car and let the battery charge level reach the same value as before the first experiment (if you have just identified the model parameters, charge it to the voltage used for the experiment in circle you performed for the identification).

15. When it is done, run the car on the path and save the `data_carX.txt` file (pay attention to not overwrite the previous one) and plug the charger.

16. On you computer, open `path_disturb.py`. Set `ONLY_FITNESS` to `False` and `PATH_PATH_DATA`. The other variables should be already correctly configured from the previous execution.

17. Run `path_disturb.py` saving the result on a file (e.g. `python path_disturb.py » temp.txt`).

18. Copy the content of the newly created file `temp.txt` into the path specification file in `paths/`, just below the compensator directives (the lines that start with '#').

19. Go back on the car console. Upload the new path specification file in the executable folder of the car (overwriting the previous one if you do not need it anymore).

20. Remember to set the constant `ENGINE_COMPENSATOR` in `car_src/const_car.h` back to 1 or the fitness estimation will not be considered.

21. Turn off the car.

22. Delete the file `temp.txt`

When to repeat the procedure

You will have to repeat the procedure every time you

- modify the path;
- modify the steering compensator directives;
- change the model parameters.

6.4.4 Tools for quantitive estimation of residual disturbances

The code in `scripts/util/predictor.py` is a reimplementation in Python of the supervisor algorithms in `src/car_src/supervisor.c`. It is very helpful for testing the supervisor offline instead of running the experiment on the test-bed over and over. This allows to analyze the very same experiment instead of a series of similar executions and it speeds up considerably the debugging of the supervisor. Of course, this script is useful only as long as it is kept updated with the changes of `src/car_src/supervisor.c`. The `Config` section of the script contains various constants that in the car code are defined in both `src/car_src/supervisor.c` and `src/util/constants.c`. Be sure that these values are updated too if you want to use this script.

In particular, the script implements two useful routines.

- `test_long_term_prediction()`: test that in each point the robust prediction (i.e. that include the effect of the disturbances) of the time at which the car enters and exits the intersection is correct. This is useful to check that the upper and lower bounds given to disturbances are effective for the prediction.

- `test_exact_supervisor()`: simulate the execution of the exact supervisor and print in detail the status of the internal variables. This is useful to debug the supervisor.

The script `test_supervisor.py` is a simple piece of code that calls these two functions. Open it, set the car number and the debug file path in the configuration section and run it on the console to see the result of the test.

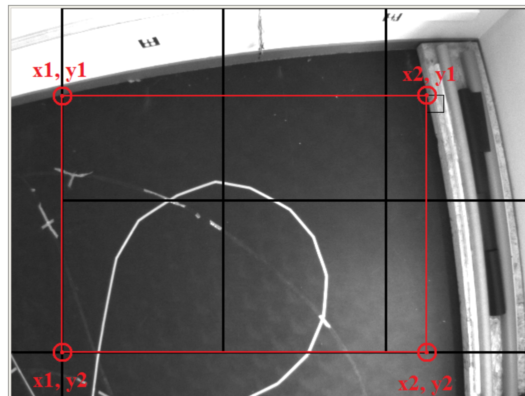


Figure 6.1: Example of how to choose the four points for CPS linear correction configuration.

6.5 CPS linear correction configuration

To reduce the CPS measurement error, I applied a linear correction to the computed global coordinates error along both directions. The estimated error is given by

$$e = ax_{loc}y_{loc} + bx_{loc} + cy_{loc} + d.$$

The parameters a, b, c, d are computed by the CPS on startup by loading a file where the actual global coordinates of four points must be saved. This is the procedure that must be followed to configure this file. This procedure must be repeated for each camera. It is calibration-independent, meaning that you do not have to repeat it if you have to perform a new instrinsic/extrinsic calibration, but if the camera is moved, the procedure must be repeated for that camera.

1. Set the variable `RECORD_OBJECT_DATA = 1` in the file `CPS.h` in the computer responsible for the camera that you want to configure and compile it.

2. Run `CPS.exe`. The `RECORD_OBJECT_DATA` mode was designed to take pictures of the cars symbols so it will ask you the car number and the section number, just put a negative number. The only thing you must insert correctly is the camera number that you want to configure.

3. Determine the four points to record. Considering the error trend, the four points should be chosen to form the broadest rectangle of interest, that is the rectangle with the biggest area contained in the camera view where the car can be tracked. To clarify, an example of how to choose the points is found in Figure 6.1. Walls, obstacles and the end of the sections limit the rectangle. The console gives you information about the pixel coordinates of the top left corner of the small squared boxes, in the figure it is located close to (x2, y2). You can move that box with keys '`a`', '`s`', '`d`' and '`w`'. Write down the pixel coordinates of those four points.

4. Now measure the global position of those four points with a measure tape and record them. You should now have written down 4 pixel coordinates (x1, y1, x2, y2) and 8 global coordinates, 2 for each point of the rectangle. Pay attention when writing down the global coordinates. Since the x and y axises of the local and the global coordinate system are inverted, it is easy to make confusion.

5. Open the folder `Desktop/camera_programs/AndreaCPS/calib_data`. There are 5 files called `error_camx.txt`, where '`x`' is the number of the camera. Open the one that refers to the camera you are configuring. The format of the file is very easy to understand and consistent with

this explanation. Write there the pixel coordinates of x1, y1, x2, y2 and the global coordinates of those points.

6. Now you can set `RECORD_OBJECT_DATA` to 0 and you are good to go.

6.6 Path specification syntax

The path specification file now does not contain only the points of the path but it also specifies other path-dependent quantities that are parsed by `ca2` at execution.

6.6.1 Steering compensator directives

In order to apply the steering compensation (see Section 3.1.3), the controller must know the curvature radius of the path. The syntax used to specify is the following.

```
# 1 648 750.07285
# 651 1298 -750.07285
```

These two lines mean that the curvature radius of the path from target point 1 to target 648 is $750.07285mm$, while from target 651 to 1298 is $-750.07285mm$. Note that the steering compensator directives lines must start with '`#`'. The curvature radius specified is positive if the car must turn right, negative to make the vehicle turn left.

6.6.2 Path-dependent disturbance specification

The disturbance $d_{path}(y)$ (see Section 3.3.3) is represented by a linear piecewise function defined in the path specification file. The syntax is the following.

```
* 50
* -1.48690195077
* -0.764317798416
* ...
* 28.9035206284]
```

Each line starts with a '`*`' character. The first line is the y step, all the others are the values of the function. In the example above the step is $50mm$ so $d_{path}(0) = -1.48690195077$, $d_{path}(50) = -0.764317798416$, $d_{path}(50n) = 28.9035206284$.

6.7 Troubleshooting

Simulator

The car does not follow the path correctly. Make sure that the paths specified in `src/sim_src/const_sim.h` by the constant `FILE_NAMES` are consistent with the experiment configuration.

ca2

The car does not follow the path correctly. Make sure that the constant `CAR_NUM` in `car_src/const_car.h` is set coherently with the symbol at the top of the car.

Chapter 7

Conclusions

In this work, a laboratory environment for experimental testing of algorithms for vehicle collision avoidance has been analyzed and configuration. In particular, the work has been aimed at testing a robust multi-agent algorithm for vehicle collision avoidance at intersections based on scheduling. The algorithm requires the prediction of the future state of human-driven vehicles in an environment affected by disturbances. To this end, a model of the cars available in the lab has been developed and a predictor based on this model has been implemented. A through analysis of the disturbances affecting the system has been performed to identify and quantify the sources of noise. The environmental factors and the eventual errors in the parameters calibration have been included in the model as additive disturbance terms.

Various techniques have been adopted to reduce the previously identified disturbances. The steering controller has been reworked to minimize the impact on the prediction performace deriving from the coupling between the lateral and the longitudinal dynamics. The effects of the power filter capacitor and the path-dependent disturbances (which are not directly modeled) have been corrected by means of a compensator applied to the motor input. Filters have been applied to refine the measurement of the vehicles position and direction. Finally, an online estimation method has been applied to compute the motor gain on the run. All these expedients have allowed to obtain a prediction less conservative and compatible with the limited space available in the laboratory to perform the experiment.

A simulator which virtualizes the laboratory environment has been implemented and used during the whole process to identify the elements of disturbance in the lab and speed up the verification of the solutions.

Finally, the aforementioned algorithm has been successfully tested in the laboratory with two computer driven vehicles. Various solutions to test the same algorithm with three cars are currently being explored.

Future work may include experiments with a mix of human-driven and

computer-driven vehicles, and the refinement of the tested algorithm for example by limiting further the number of assumptions (e.g. without knowing ahead which way the driver chooses at the intersection) or by decentralizing the decision process from an external supervisor to the set of computers on-board each vehicle.

Bibliography

[1] L. Bruni, A. Colombo, and D. Del Vecchio. Robust multi-agent collision avoidance through scheduling. In *Proc. 52nd IEEE Conference on Decision and Control*, pages 3944–3950, 2013.

[2] A. Colombo and D. Del Vecchio. Efficient algorithms for collision avoidance at intersections. In *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, 2012.

[3] H.P. H.P. Whitaker, J. Yamron, and A. Kezer. Design of model-reference adaptive control system for aircraft. Report r-164, Institution Lab, MIT, 1958.

[4] R. Verma, D. Del Vecchio, and H. Fathy. Development of a scaled vehicle with longitudinal dynamics of a hmmwv for an its testbed. *IEEE/ASME Transactions on Mechatronics*, 13(1):46–57, 2008.